CUCS-29-82

The NON-VON Supercomputer[1]

David Elliot Shaw

Department of Computer Science
Columbia University

August 1982

Abstract

NON-VON is a highly parallel, non-von Neumann "supercomputer", portions of which are now being implemented in the Computer Science Department at Columbia University. The machine is intended to support the extremely rapid execution of large scale data manipulation tasks, including relational database operations and many other functions relevant to commercial data processing.

The NON-VON architecture includes a tree-structured Primary Processing Subsystem (PPS), which we are implementing using custom nMOS VLSI circuits, along with a Secondary Processing Subsystem (SPS) based on a bank of intelligent disk drives. A high-bandwidth parallel interface provides for rapid data transfer between the two subsystems. This paper describes the organization of the NON-VON machine, with particular emphasis on the structure and function of the PPS. Some of the most important NON-VON programming techniques are then outlined, and their application to typical data processing applications illustrated with simple examples.

[1] This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-82-C-0427.

Table of Contents

List of Figures

Acknowledgements

The efforts of a number of individuals are reflected in the research reported in this paper. In particular, the author wishes to acknowledge the contributions of his faculty co-investigators, Professors Salvatore J. Stolfo, Zvi M. Kedem, and Michael Lebowitz, and of the eight gifted and zealous Ph.D. students who form the central core of the NON-VON Project. Specifically, much of the design and VLSI layout of the PPS processing element is due to efforts of Hussein Ibrahim and Dan Miranker, aided by the critical insights of Sanjiv Sharma. Dan Miranker and Dayton Clark were responsible for a large part of the implementation of the NON-VON simulator, while Bruce Hillyer made significant contributions involving fault tolerance and testing, record allocation, high-level languages, and parallel algorithms. Steve Taylor has played a major role in both hardware design and translator development, while Dong Choi and Yoram Eisenstadter have been recent participants in the implementation of database management software for the NON-VON machine.

The "real-world" expertise of our project engineers, Ted Sabety and Shun Ueda, has been critical to the success of our integrated circuit and system design efforts. Substantial contributions to the theory, architecture, design, implementation, simulation, and programming of NON-VON have also been made by Bob Floyd, Don Knuth, Gio Wiederhold and Terry Winograd, all of the Stanford Computer Science Department, and by Dave Bacon, Peter Brajak, Lincoln Hu, Kevin Kalajan, Stuart Kreitman, Ted Markowitz, Reynaldo Newman, Terry Newton, Alessandro Piol, Arthur Sun, Danny Sykora, and Michael Weisberg, at Columbia. Finally, Jerry Wiener deserves special recognition for his role as administrator of the NON-VON project. The contributions of each of these individuals are gratefully acknowledged.

1 Introduction

Two observations regarding the evolution of computer systems have, during the past decade or so, become so commonplace as to require little discussion. First, the cost of digital hardware has dropped to the point where, in many applications, processors need no longer be considered a scarce resource. Second, the cost of computer software is increasing, both in absolute terms, and even more dramatically, by comparison with that of the hardware on which it executes.

The design of highly parallel machines is more commonly associated with the first of these observations than the second. Indeed, the availability of large numbers of inexpensive processing elements quite naturally suggests the possibility of constructing highly concurrent systems capable of very rapid execution. The NON-VON machine, which incorporates a large number (between 100,000 and 1,000,000, within the target time frame) of unusually simple processors, is one of the most ambitious proposals to date for the realization of very large scale parallelism using current integrated circuit technology. It should be emphasized, however, that issues related to software are as central to the goals of the NON-VON project as is the achievement of unprecedented processing power.

NON-VON was designed to apply computational parallelism on a rather massive scale to a large share of the information processing functions now performed by digital computers. In particular, highly efficient support is provided for the kinds of operations which seem to characterize much of the workload involved in commercial database management and data processing applications. This paper describes the architecture of the NON-VON machine and illustrates the manner in which it achieves such a high degree of parallelism.

The paper is divided into three sections. The current section briefly reviews the history and current status of the NON-VON Project, and provides an informal comparison between the essential elements of a conventional computer system and the analogous components of the NON-VON machine. In the second section, NON-VON's physical organization is described at several levels. The final section describes the instruction set of the NON-VON Processing Element (PE), and introduces some of the most important paradigms for the implementation of NON-VON software.

1.1 Project History and Current Status

The theoretical basis for the NON-VON machine was established in the course of a doctoral research project at Stanford University [16], [17]. Asymptotic improvements in the evaluation of a number of relational database operations were reported. These results employed a highly general technique known as hash partitioning, by which many large-scale data processing operations having O(n log n) time complexity on a von Neumann machine may be implemented in linear time on a different type of machine which has the same hardware complexity. The interested reader is referred to these earlier results for a rigorous analysis of the complexity of algorithms to which the current paper will make only casual reference.

Detailed design of the NON-VON hardware began in the latter part of 1981, and has gained momentum since that time. Major funding for the NON-VON project has recently been obtained from the Defense Advanced Research Projects Agency, supporting the implementation at Columbia University of certain key elements of an initial prototype, which we have come to call NON-VON 1. These elements employ custom-designed nMOS VLSI circuits, which are to be fabricated remotely using DARPA's "silicon brokerage" system, MOSIS. As of August, 1982, a preliminary data path for the NON-VON 1 Processing Element (to be described shortly) has been laid out in nMOS VLSI, simulated and debugged at the logic level, and mechanically checked for design rule violations. Portions of these designs have recently been submitted for fabrication.

The development of software for the NON-VON machine has proceeded in parallel with our hardware implementation efforts. A simulator for the NON-VON instruction set was implemented in the fall of 1981, and has since been enhanced to provide a user-convenient vehicle for the development of NON-VON software. Higher-level linguistic constructs have been implemented as part of an evolving LISP-based programming environment, and compilers for two parallel languages, modelled after Pascal and APL, are now under development. About twenty individuals have thus far written NON-VON programs, and have tested their execution using the instruction set simulator. While no large-scale applications have yet been implemented, our experience with this modest corpus of simple NON-VON programs has already led to several minor refinements of the architecture and instruction set.

1.2 Comparison with von Neumann Machines

If pressed to identify a single principle underlying the essential "philosophy" of the NON-VON architecture, we would probably choose to highlight the strategy of extensively intermingling processing and storage resources. This strategy is employed at several levels within the NON-VON machine, and is perhaps best appreciated by contrast with the organization of a conventional computer system.

In an ordinary von Neumann machine, a single (often quite powerful) central processing unit is connected to a single (often quite large) random access memory, which is used for the storage of both programs and data. The CPU and RAM communicate in a serial (or at best, weakly parallel) fashion through a narrow conduit which Backus [2] has called the "von Neumann bottleneck".

Moreover, the limitations of this organization are becoming more serious as technological progress increases both the potential power of processing hardware and the realizable size of computer memories.

In the NON-VON Primary Processing Subsystem (PPS), on the other hand, a large number of very simple, highly area-efficient processing elements (PE's) are, in effect, distributed throughout the memory. In particular, each integrated circuit in the PPS contains a number of PE's (eight, in our planned prototype version, which is based on typical 1982 nMOS device dimensions and die sizes). Each PE is associated with a small amount of locally accessible random access memory (64 bytes, in NON-VON 1). The potential processor/memory bandwidth in NON-VON is thus many orders of magnitude higher than in conventional machines.

In practice, many or all of these tiny PE's are often able to operate concurrently on data stored in their respective local memories, supporting effective execution speeds far exceeding those of today's fastest supercomputers. Because of their small size, however, the PPS is expected to be scarcely more expensive than an equivalent amount of ordinary random access memory. (Specifically, we estimate that a NON-VON PE might occupy as little as twice the area that would be required for the amount of RAM it would incorporate.) From the viewpoint of performance, the PPS may thus be regarded as an ultra-high-speed parallel processing ensemble; from a cost perspective, though, it is better viewed as a (slightly overpriced) random access memory unit.

A similar comparison between the mass storage facilities of a conventional computer system and the analogous subsystem within the NON-VON machine may also prove instructive. In the typical large-scale data processing system, a large bank of disk drives is charged with the task of responding "mindlessly" to a sequence of requests for data posed by the CPU. In practice, most of this data in fact proves irrelevant to the task at hand. The secondary storage subsystem -- a husky and obedient, but rather dim-witted brute -- is generally incapable of separating wheat from chaff, and must pass both along to its more intelligent master.

As in the case of the von Neumann bottleneck, the pathway between the "thinking part" and the "remembering part" of such a system is a relatively narrow one, even in the most sophisticated contemporary systems. While a modest degree of parallelism is sometimes achieved in the disk-to-computer interface, the process of transferring data between primary and secondary processing hardware remains, for the most part, an essentially sequential function.

In the NON-VON Secondary Processing Subsystem (SPS), on the other hand, a small amount of processing hardware is associated with each disk head. This hardware allows records to be inspected "on the fly" to determine whether a given record is relevant to the operation at hand. The NON-VON SPS is thus able to be more discriminating in the data it passes along to the primary processing hardware. Furthermore, the topology of the PPS supports a massively parallel interface between primary and secondary storage, allowing data transfers between the subsystems to keep pace with the greatly accelerated execution possible within the PPS. In short, the SPS is able to "filter" data before it is sent to the PPS, and to transfer the "filtrate" in a highly parallel manner.

2 Organization of the NON-VON Machine

In this section, we describe the physical structure of the NON-VON machine. The top-level organization of the system is outlined in the first subsection. Our principal concern in this paper, however, will be with the Primary Processing Subsystem, which is described in more detail in the second subsection. In the third subsection, we discuss certain topological considerations that influenced the design of the PPS. The section concludes with a detailed description of the individual processing elements from which the NON-VON PPS is constructed.

2.1 System Organization

The top-level organization of the NON-VON machine is illustrated in Figure 1.

The PPS is configured as a binary tree of processing elements. By dynamically altering certain switch settings within the PE's, however, the subsystem can be reconfigured to provide for linear, tree-structured or global bus communication. With the exception of minor differences in the "leaf nodes", each PE is laid out identically, and comprises a small random access memory, a modest amount of processing logic, and an I/O switch supporting the various modes of inter-PE communication.

At the root of the tree is a von Neumann machine called the Control Processor (CP), which is responsible for coordinating various activities within the PPS. In a production version of the NON-VON machine, the CP would in fact be specialized in several respects to optimize its performance as a controller for the PPS. In the context of this paper, however, the CP may be thought of as a conventional single instruction stream, single data stream (SISD) computer. While certain sequences of instructions are executed sequentially within the CP, it is also capable of broadcasting instructions to be

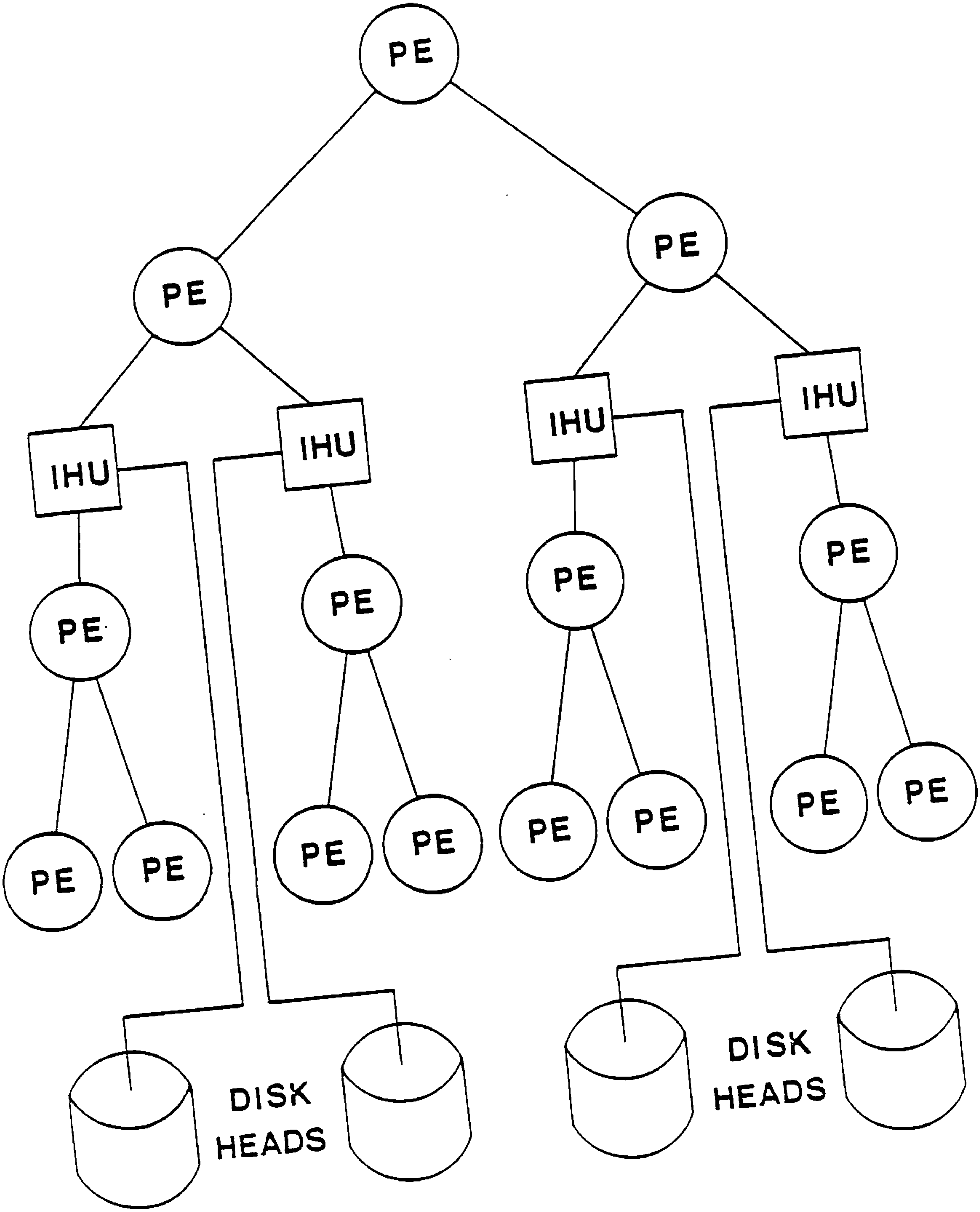

Figure 1: Organization of the NON-VON Machine

simultaneously executed by all enabled PE's in the tree on a single instruction stream, multiple data stream (SIMD) basis [6].

The SPS is based on a number of rotating storage devices, which might in practice be realized using either slightly modified multiple-head disk drives or unmodified single-head drives. Associated with each disk head in the SPS is a separate sense amplifier and a small amount of logic capable of dynamically examining the data passing beneath it. These Intelligent Head Units (IHU's) are also capable of performing simple computations (hash coding, for example), and of serving a control function similar to the role played by the CP.

Assuming that the number of intelligent disk heads is equal to 2^k, for some integer k, the k-th level of the PPS tree (where the root is considered to be at level zero) is used to interface the PPS and SPS. Specifically, each of the k internal PPS nodes at this level is associated with a different IHU. Physically, this connection is made by interposing the IHU between the interface-level PE and its parent PE, as illustrated in Figure 1.1. In its passive state, the IHU acts as a simple bus, passing information in both directions without change.

In certain algorithms, though, each IHU serves as an active control processor for the subtree it roots, allowing independent, asynchronous computation within the various interface-rooted subtrees. (NON-VON is thus not, strictly speaking, a SIMD machine; in practice, however, it often functions as either a single SIMD machine or a collection of such machines.) The most common application of this capability is in the concurrent loading of each interface-rooted subtree from its respective disk drive. Such parallel transfers between SPS and PPS account for the unusually high effective I/O bandwidth achieved in a wide range of applications. Other algorithms make use of the "top part" of the PPS tree -- more precisely, the portion consisting of all

PE's lying above the interface level. Among other things, this portion of the tree can be used for the efficient synchronization of interface-rooted subtrees following asynchronous operation.

A more thorough discussion of the SPS, its interface to the PPS, and the kinds of algorithms that make explicit use of the upper and lower portions of the tree is, unfortunately, beyond the scope of this paper. The reader may, however, find the discussion of hash partitioning presented in [18] to be useful in gaining some appreciation for the way NON-VON-like architectures provide support for at least one important family of highly parallel algorithms involving large amounts of data.

2.2 The Primary Processing Subsystem

Although physically structured as a binary tree, the NON-VON PPS can be dynamically reconfigured to support communication patterns characteristic of two other topologies in a highly efficient manner. In this subsection, we describe the physical organization of the NON-VON PPS and discuss the three modes of communication it supports.

The PPS is implemented using a number of identical PPS chips. Our use of a single circuit is made possible by the adoption of a tree-partitioning scheme first suggested by Leiserson [12]. This approach embeds both a complete subtree (containing 2^c-1 constituent PE's, for some c depending on device dimensions) and a single interior node on each chip. Four nine-bit busses (eight bits for data, and one for a control function, which will not be discussed in this paper) enter the chip. One, called the T connection, leads to the root of the chip's subtree, while the other three, called the F, L, and R connections, attach the single interior node to its father, left child and right child, respectively, within the tree.

A simple recursive procedure allows the construction of a complete binary tree of arbitrary size using only chips of this type. This construction is illustrated for the case of two chips in Figure 2. Note that the resulting circuit consists of a larger complete binary subtree (in this case rooted by the interior node of the chip on the left side of Figure 2), together with a single unconnected interior node (the interior node of the chip on the right). This circuit has the same four external connections — T, F, L and R — as did a single chip.

The interconnection scheme shown in Figure 2 may be easily extended to allow the construction of a simple, planar printed circuit board layout (also due to Leiserson), which is illustrated in Figure 3. The regularity of this PC board layout scheme has greatly simplified the task of designing the NON-VON PPS. Furthermore, the area required for routing wires within the PC board is strictly proportional to the number of chips, allowing the efficient implementation of boards of arbitrary size.

The PPS is simply a collection of these PC boards, interconnected in precisely the same manner as are the constituent PPS chips. This scheme is suitable for the construction of a PPS comprising 2^b-1 PE's, for arbitrarily large b, and leaves only a single interior PE unused.

The subtree incorporated within each PPS chip is configured geometrically according to a "hyper-H" embedding [3], as illustrated in Figure 4. This construction is highly regular, is area-optimal (in the sense that the amount of silicon area occupied by the tree is proportional to the number of PE's), and is easily extended to incorporate larger numbers of PE's as device dimensions scale downward.

The tree structured inter-PE bus structure supports three distinct modes of communication:

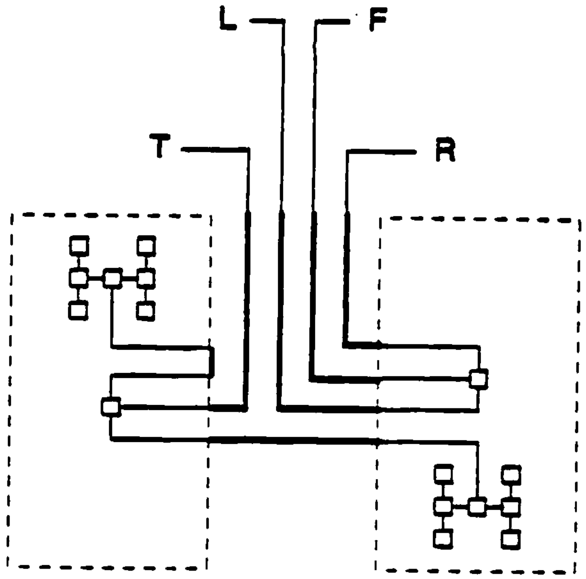

Figure 2: Interconnection of Two Leiserson Chips

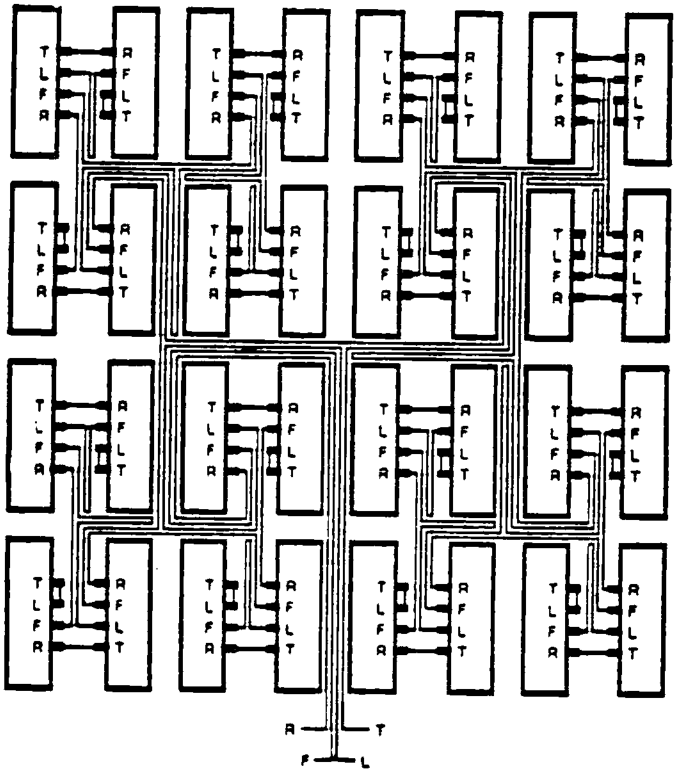

Figure 3: The PPS Printed Circuit Board (Leiserson Layout)

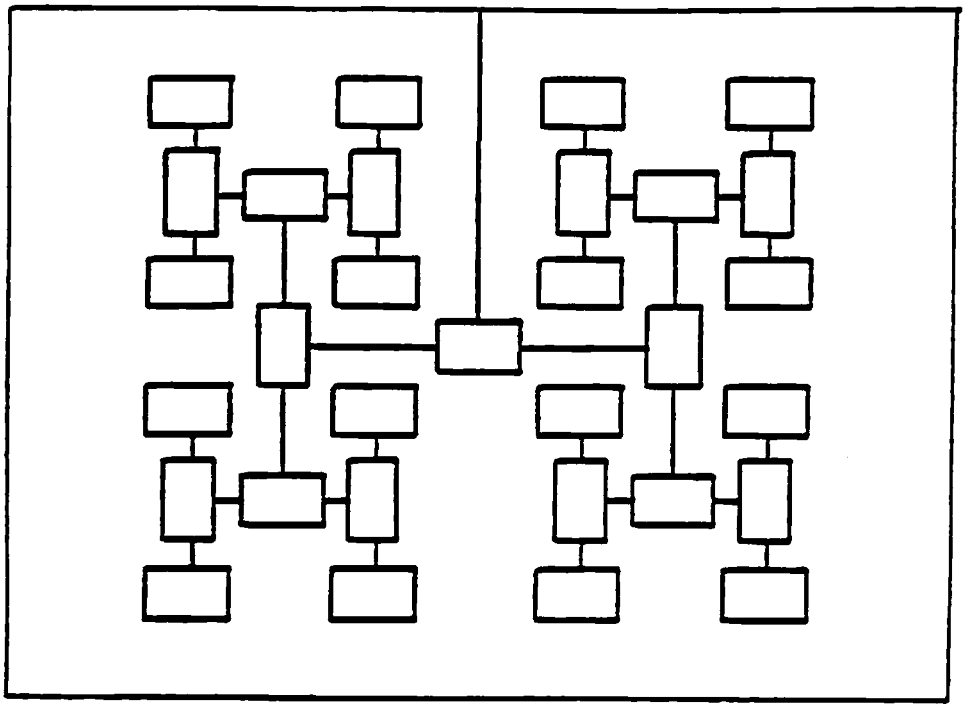

Figure 4: Hyper-H Embedding of the Binary Tree

1. Global bus communication, supporting both broadcast by the CP to all PE's in the PPS and data transfers from a single selected PE to the CP.

2. Physically adjacent (tree) communication to the Parent (P), Left Child (LC) and Right Child (RC) PE within the physical PPS tree.

3. Linearly adjacent neighbor communication to the Left Neighbor (LN) or Right Neighbor (RN) PE in a particular logical linear sequence.

The global broadcast function supports the rapid parallel communication of instructions and data from the CP to the individual PE's, as required for SIMD execution. As will be seen in Section 3, it is also possible for a selected PE to send data to the CP. Using the CP as an intermediary, any PE can thus send data to any other PE. No communication concurrency is achieved, however, when data is passed from one PE to another using the global mode primitives.

The physically and linearly adjacent communication modes, on the other hand, support fully parallel communication. The former is used in many tree-based algorithms. (Parallel sorting and the logarithmic-time addition of n numbers are two examples). The linear mode is used in algorithms in which many PE's simultaneously exchange data or control information with their immediate predecessor or successor PE's in some predefined total ordering. Several mappings between the linear logical sequence and the tree-structured physical topology of the PPS are possible; these alternatives are discussed in the following subsection. By way of summary, each PE can communicate with five other PE's, which are referred to within its own local context as P, RC, LC, RN and LN.

2.3 Topological Considerations

The choice of a tree-structured topology for the PPS was based on considerations involving such factors as the efficient use of silicon area, favorable pinout properties, and suitability for the rapid broadcasting of data. Another important factor was the ability to efficiently emulate a linear array (a sequence of PE's, each connected only to its immediate predecessor and successor), which, among other things, plays a central role in one of our techniques for manipulating records too large to fit within a single PE.

First, we observe that each PPS chip has exactly four external connections (each nine bits wide, in NON-VON 1), regardless of the number of PE's contained within its subtree. Because of its fixed pinout requirements, independent of the size of the embedded subtree, the realizable capacity of the PPS chip will increase quadratically with decreases in minimum feature width. This will permit dramatic increases in the computational power of the NON-VON PPS unit as device dimensions are scaled downward with continuing advances in VLSI technology. (During the target time frame for a production version of a NON-VON-like machine, a c value of 7 or 8, corresponding to several hundred processing elements per PPS chip, would seem feasible.)

It is worth mentioning that, with the notable exceptions of linear arrays and such closely related architectures as simple rings, most topologies proposed for parallel computation in VLSI do not share the area and pinout properties we have just outlined. A homogeneous implementation of the orthogonal and hexagonal mesh-connected topologies proposed for the implementation of systolic arrays [10], for example, would require a number of pins proportional to the square root of the number of PE's embedded within a chip. This is also true of such "nearly equivalent" architectures as toroidal meshes [7] and the chordal ring [1]. In the absence of a breakthrough in packaging technology allowing a dramatic increase in the number of pins per chip, such architectures will thus become progressively more "I/O-bound" as device dimensions continue to scale downward.

A large family of closely interrelated architectures exemplified by the shuffle-exchange [11] and cube-connected cycles [13] networks are even more limited in this regard. The pinout requirements of this family of architectures grow considerably faster than those of the two and three-dimensional meshes. Furthermore, area proportional to $n^2/\log^2 n$ is (provably) required to embed n PE's within a single chip using such schemes [19]. Thus, such architectures are subject to quickly decreasing returns to scale as improvements are made in logic densities.

Another topological consideration in designing a machine having as many processing elements as is envisioned for NON-VON is the manner in which global communication is handled. If a "processor density" comparable to that of the NON-VON machine is to be achieved, only a very small amount of local memory can be associated with each PE. The extremely fine "granularity" of such a massively parallel machine is thus inconsistent in principle with the replication of substantial programs within each PE. For this reason, the realization of very high processor densities would seem to be inextricably tied to the efficient global broadcasting of instructions.

What are the implications of this requirement for rapid global broadcasting capabilities? First, we note that the "bounded valence assumption" (the restriction that no "node" be connected to more than a fixed maximum number of "wires"), which is central to all contemporary models of computation in VLSI, precludes the possibility of broadcasting in time less than logarithmic in the number of recipients. While this lower bound is realized by members of the tree-structured and shuffle-based families, most other topologies do not share this property. The two-dimensional meshes, for example, are incapable of broadcasting in time less than proportional to the square root of the number of recipients. In the linear array, broadcast requires linear time. The same is true of the ring network, which may be considered "almost equivalent" to the linear array in the context of these concerns.

In the NON-VON PPS, broadcast communication is effected not only in asymptotically optimal time, but with extremely small constants as well. Specifically, information that is broadcast is not buffered at each level of the tree according to a sequential discipline, but is instead propagated in an unclocked manner, passing through a very small amount of combinational logic at each level. NON-VON thus provides highly efficient support for the global broadcasting of instructions and data to all processing elements.

By way of summary, the meshes are as area-efficient as the binary tree, but would increasingly suffer from pinout limitations and broadcast inefficiencies if used in high-density applications of the sort with which we are concerned. Such architectures as the shuffle-exchange network and cube-connected cycles, while matching the optimal broadcast time of the tree, have area complexity and pinout characteristics that would be incompatible with this degree of parallelism. Of the architectures we have considered, only the linear array and the tree may be considered indefinitely scalable, in the sense that their pinout is fixed, and their area proportional to the number of embedded processors.

There are two reasons for our selection of the tree, and not the linear array, as the topology for the NON-VON PPS. First, a strictly linear interconnection network requires time proportional to the number of processors for broadcast. Second, the NON-VON PPS tree is in fact capable of dynamically reconfiguring to emulate the behavior of a linear array with only a minor constant-factor degradation in speed, as shown below. (It should be clear that the converse is not true.) Thus, we are in fact giving up very little by choosing the tree over the linear array.

There are several ways in which a binary tree can be used to emulate the behavior of a linear array. The most obvious possibility is to map the nodes of the tree onto a linear sequence according to a standard preorder, inorder or postorder traversal scheme [9]. The nodes of the tree shown in Figure 5, for example, are mapped onto those of a linear array by inorder enumeration.

Let us now consider what data would have to pass along each tree edge in order to simultaneously transfer a single data element along the path from each tree node to its successor in the linear sequence. These paths are indicated in Figure 5 by arrows extending from each node to its linear successor, in general passing through intermediate nodes along the way. It should be noted that since every other element in the inorder sequence is a leaf node, half of these arrows (which we have colored black) originate in internal nodes and terminate in leaf nodes, while the other half (colored white) extend from leaf nodes to internal nodes. Note further that each tree edge is associated with exactly one black and one white arrow. If the communication cycle is divided into separate phases for communication to and from leaf nodes, all nodes in the tree can thus communicate with their respective successors within a single communication cycle.

The inorder embedding scheme, however, has the property that the maximum number of physical tree edges between two nodes that are adjacent in the linear logical sequence grows logarithmically with the size of the tree. This drawback is present in the preorder and postorder enumeration schemes as well, since both mappings contain paths extending from root to leaf. Since each phase of the communication cycle must be at least as long as the maximum time required for communication between any two linearly adjacent neighbors, it is worth investigating whether a linear array can be embedded in the binary tree in such a way that the maximum path between linearly adjacent nodes is bounded by a constant.

As it happens, we have found a way to configure NON-VON's simple I/O switches so that the longest path between linear neighbors is exactly three. Based on a mathematical result first reported by Sehanina [14], our scheme requires

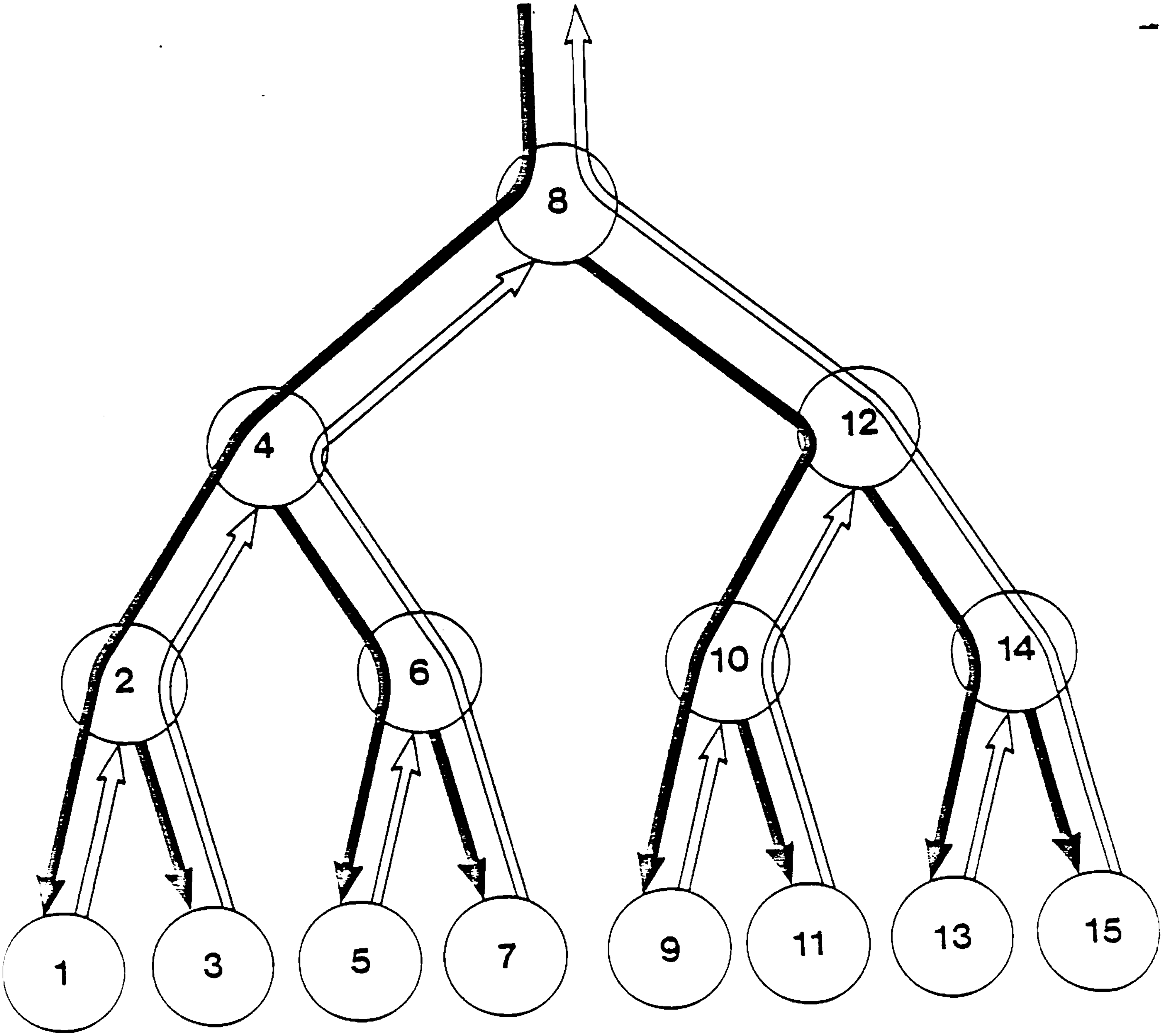

Figure 5: Inorder Embedding of the Linear Array

that the I/O switch settings at successive levels of the tree alternate between those that would be employed in a preorder configuration and those that would be used for a postorder mapping. This "bounded neighborhood" embedding is illustrated in Figure 6.

In practice, however, the relative advantage of bounded neighborhood embedding over inorder mapping is not so great as it might first appear. The reason has to do with the fact that the delay between physically adjacent PE's is not in fact constant throughout the PPS tree. In particular, while most pairs of physically adjacent PE's reside on the same chip, many such pairs are located on different chips, some on different printed circuit boards, and (in a large-scale system) a few in different cabinets. In a realistic large-scale system, the delays encountered between chips, boards and cabinets would typically be considerably larger than those experienced within a given chip. Because the speed of the communication cycle is limited by the slowest data transfer between linearly adjacent neighbors, each communication phase must be slow enough to allow for the transfer of data between cabinets.

Rough calculations based on estimates of intra-chip, inter-chip, inter-board and inter-cabinet delays suggest that the relative advantage of the bounded neighborhood mapping over a simple inorder embedding, while not negligible, is not overwhelming for PPS trees of the sizes likely to be encountered in practice. In the interest of simplicity, we have thus decided to adopt the inorder embedding for use in the NON-VON 1 prototype. Later versions of NON-VON will probably be capable of supporting any of the four orderings discussed above, and of dynamically switching among these orderings.

Having argued strongly for the adoption of a tree-structured physical topology in systems exhibiting parallelism on the massive scale attempted in NON-VON, it must be emphasized that the alternative architectures discussed in this subsection may in fact prove well suited to applications amenable to coarser

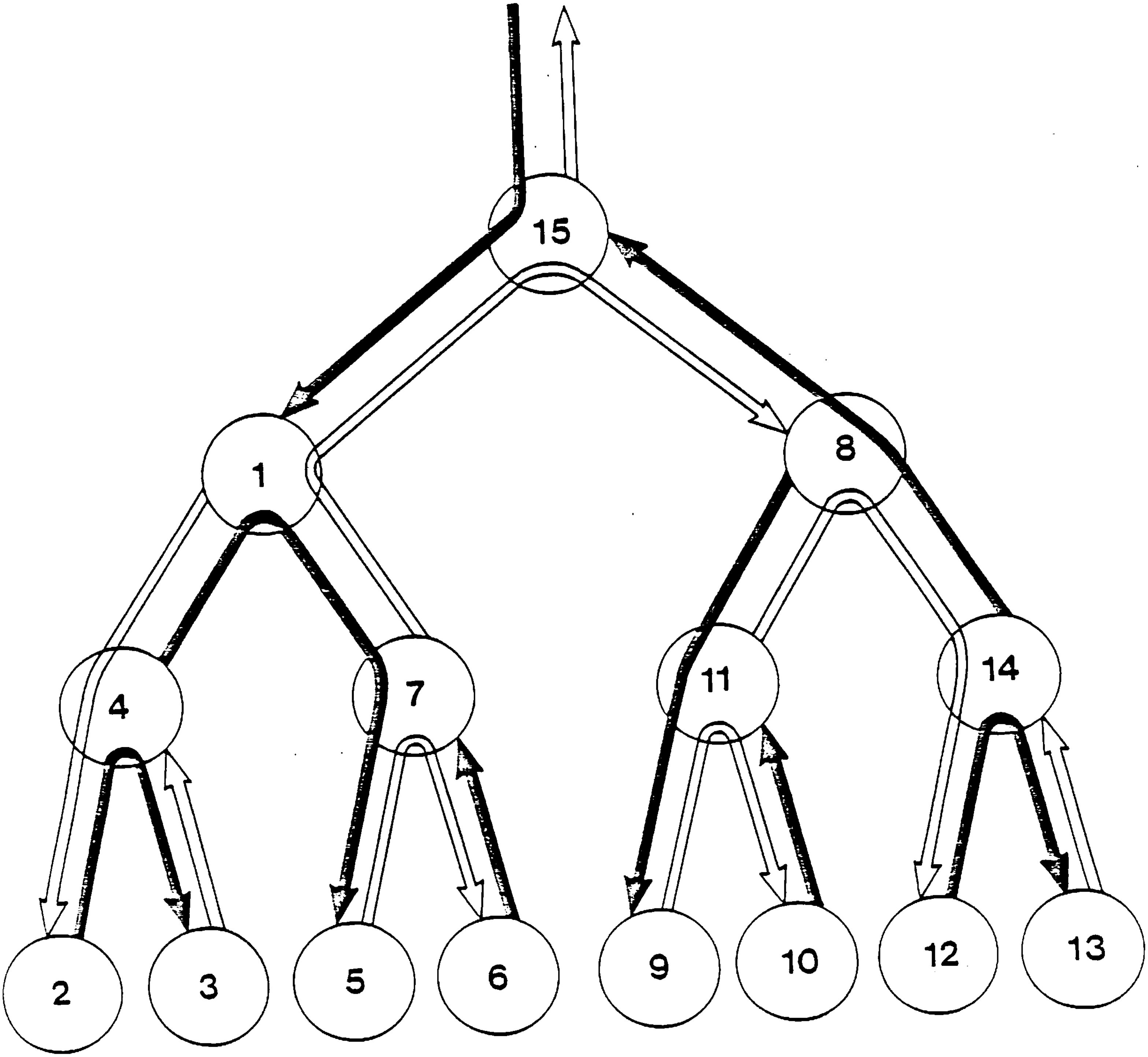

Figure 6: Bounded Neighborhood Embedding of the Linear Array

granularities, especially in the short term. In particular, the superficially compelling asymptotic arguments advanced above must be considered in the context of Larry Snyder's well-phrased reminder that "we don't live in Asymptopia". On the other hand, if device dimensions continue to decrease, the NON-VON approach to large-scale parallelism may soon have us "living in the suburbs".

2.4 The Processing Element

The NON-VON PE is much simpler, smaller and less powerful than the processing elements incorporated in previously proposed tree machines [4], [15]. In large part, this difference reflects the SIMD execution of globally broadcast instructions, which characterizes NON-VON's typical operation. By avoiding extensive reliance on MIMD (multiple instruction stream, multiple data stream) operation, NON-VON obviates the need for large local program memories and area-expensive processing and communication hardware, and amortizes the cost of most of its control logic over a large number of independent data paths.

The result is a PE that occupies a small fraction of the area required for an ordinary microcomputer, supporting a "processor density" far greater than that of most parallel machines. From an applications viewpoint, the extreme area-efficiency of the NON-VON PE makes it economically feasible to divide primary storage into roughly "record-sized" units, and to associate a separate processing element with each such unit. This aspect of the NON-VON design is central to its processing power in large-scale data processing applications, as we shall see in the remainder of this paper.

The NON-VON 1 PE comprises:

1. A 64 word X 8 bit random access memory
2. A set of eight 8-bit byte registers

3. A set of eight 1-bit flag registers
4. A byte-wide arithmetic comparison unit (ACU)
5. A bit-wide arithmetic logical unit (ALU)
6. A byte-wide I/O switch
7. A programmable logic array (PLA)

A top-level block diagram of the PE is presented in Figure 7.

The data path is organized around two data buses — one eight bits wide, the other one bit wide. The local RAM, byte-wide registers, and ACU all communicate through the eight-bit bus. One of the eight byte registers serves as a memory address register (MAR), into which addresses are latched in the course of accessing the local RAM. (Although the NON-VON 1 PE contains only 64 bytes of RAM, the architecture is capable of supporting a local memory of up to 256 bytes.)

Two of the other registers, labelled A8 and B8, are distinguished as byte accumulators, and include special hardware for performing circular shifts. In the course of such shift operations, the bits of A8 and B8 may be rotated through two distinguished flag registers, A1 and B1, which are referred to as the bit accumulators. This feature provides a bit-serial link between the byte-wide and bit-wide portions of the data path. In addition, the ACU is capable of of comparing the contents of A8 and B8 and latching the results into the bit accumulators. Specifically, A1 is set if and only if the contents of A8 and B8 are identical, while B1 is set if and only if the A8 value is greater than that of B8. Another distinguished byte register, IO8, serves a special role, discussed below, involving the latching of data to be transmitted between PE's. The remaining byte registers (labelled C8, X8, Y8 and Z8 in Figure 7) are available for general use.

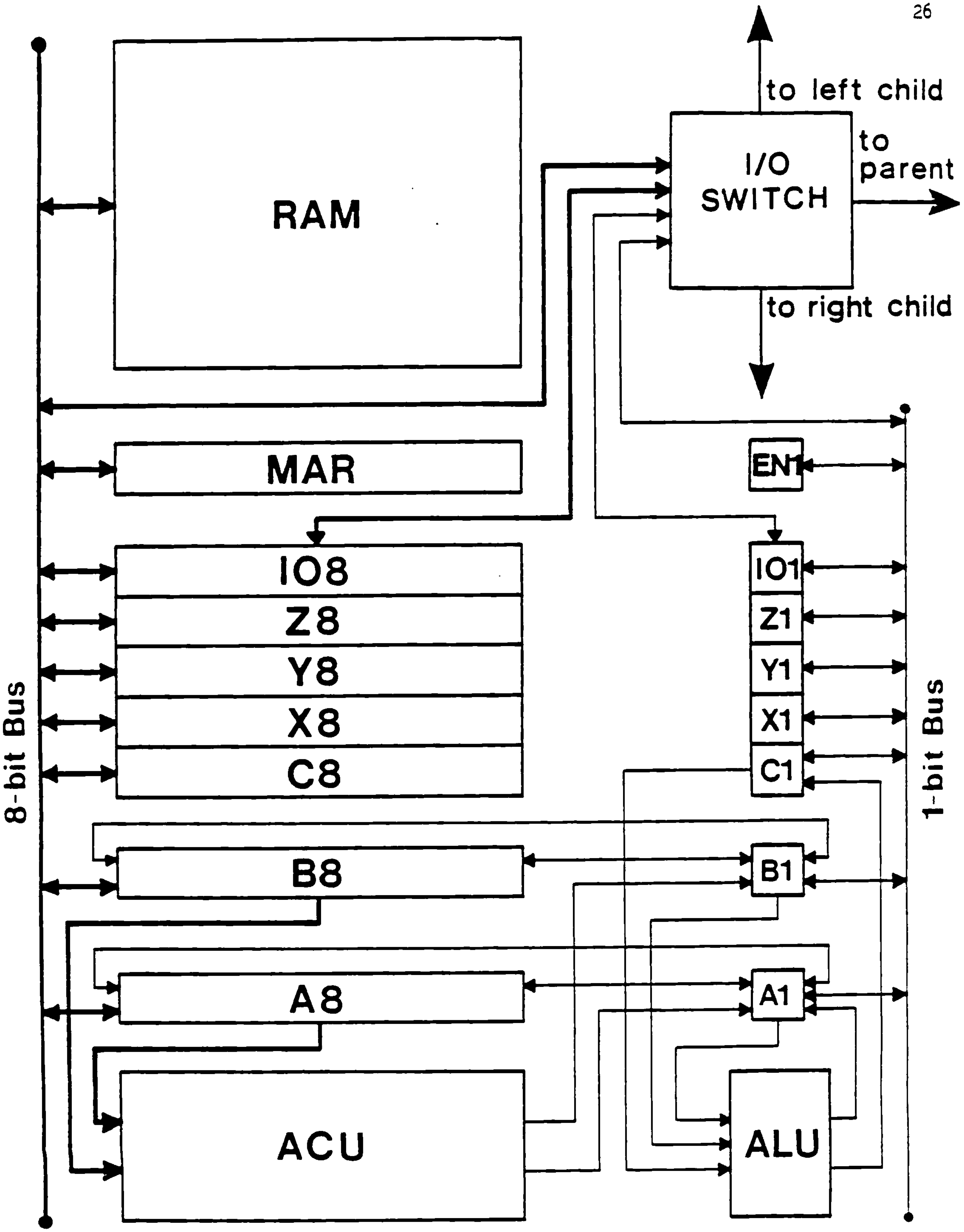

Figure 7: Block Diagram of the Processing Element

The one-bit data bus is used to transfer data among the single-bit flag registers, and to supply operands to, and obtain results from, the bit-wide ALU. As noted above, two of the flag registers, called A1 and B1, serve special roles as accumulators. In particular, the bit accumulators serve as inputs to the ALU, along with the contents of a third flag register, C1, which is used to store the carry bit in the course of bit-serial addition and subtraction. Upon execution of one of the logical function instructions (described below), the ALU is capable of computing one of the sixteen possible boolean functions of A1 and B1, and storing the result in A1. In response to an ADD1 instruction, the ALU functions as a full adder, computing sum and carry bits for the three inputs A1, B1 and C1. The sum bit is stored in A1 and the new carry bit in C1. Analogous results are produced during a SUB1 instruction.

Another flag register, EN1, is distinguished as an enable flag. This flag is used to activate and deactivate individual PE's within the PPS. In general terms, only those PE's whose enable flags are asserted will respond to instructions broadcast by the CP. If EN1 is set to 0 in a particular PE, all instructions except one (the ENABLE instruction, discussed below) will be ignored. A number of tricky issues arise in considering the behavior of enabled and disabled PE's, particularly in the case of inter-PE communication operations. These issues will be examined as part of our detailed discussion of the instruction set. Finally, another flag register, IO1, is the boolean analogue of IO8, serving as an I/O latch in the transmission of single-bit values between PE's. The other flag registers (labelled X1, Y1 and Z1 in Figure 7) may be used to store arbitrary boolean values.

The I/O switch is connected to both the eight-bit and one-bit buses, allowing the transfer of byte and flag data to the parent, left child and right child PE's (and, depending on the switch settings, to other PE's as well). Finitestate control for the I/O switch and data path are provided by a common PLA. Consideration has been given to the possibility of "factoring out" a portion of the PLA associated with each PE on a given chip into a single PLA shared by all such PE's. This approach might ultimately allow the "amortization" of part of the control logic over a large (and increasing, with reductions in device dimensions) number of PE's. While we have not employed a "PLA factorization" strategy in designing NON-VON 1, this approach is likely to be incorporated in future versions.

In order to keep the area of the PE many times smaller than that of a conventional microprocessor, many decisions have been made in which execution speed is sacrificed for silicon area. While it is difficult to rigorously defend such complex and interacting design decisions, an intuitive justification for this strategy may prove illuminating. First, it is worth mentioning that, in our experience, the savings in area made possible by such decisions in practice often vary as the square of the associated degradation in speed. While such a relationship is observed in many aspects of processor design, the routing of an ordinary n-bit data bus through a 90-degree turn provides a simple example. Note that the area required to "turn the corner" is proportional not to n, but to the square of n, as illustrated in Figure 8. More substantive examples abound.

Because the chip- and board-level layouts employed in the PPS consume area proportional to the number of PE's, the number of PE's realizable in a system containing a fixed number of chips varies inversely with the area of a single PE. In the critical sections of typical NON-VON programs, all available PE's are typically performing useful computational work in parallel. Thus, NON-VON's maximum achievable execution speed is in some sense inversely proportional to PE area. This being the case, we have found it counterproductive in many cases to achieve a given speedup at the expense of a

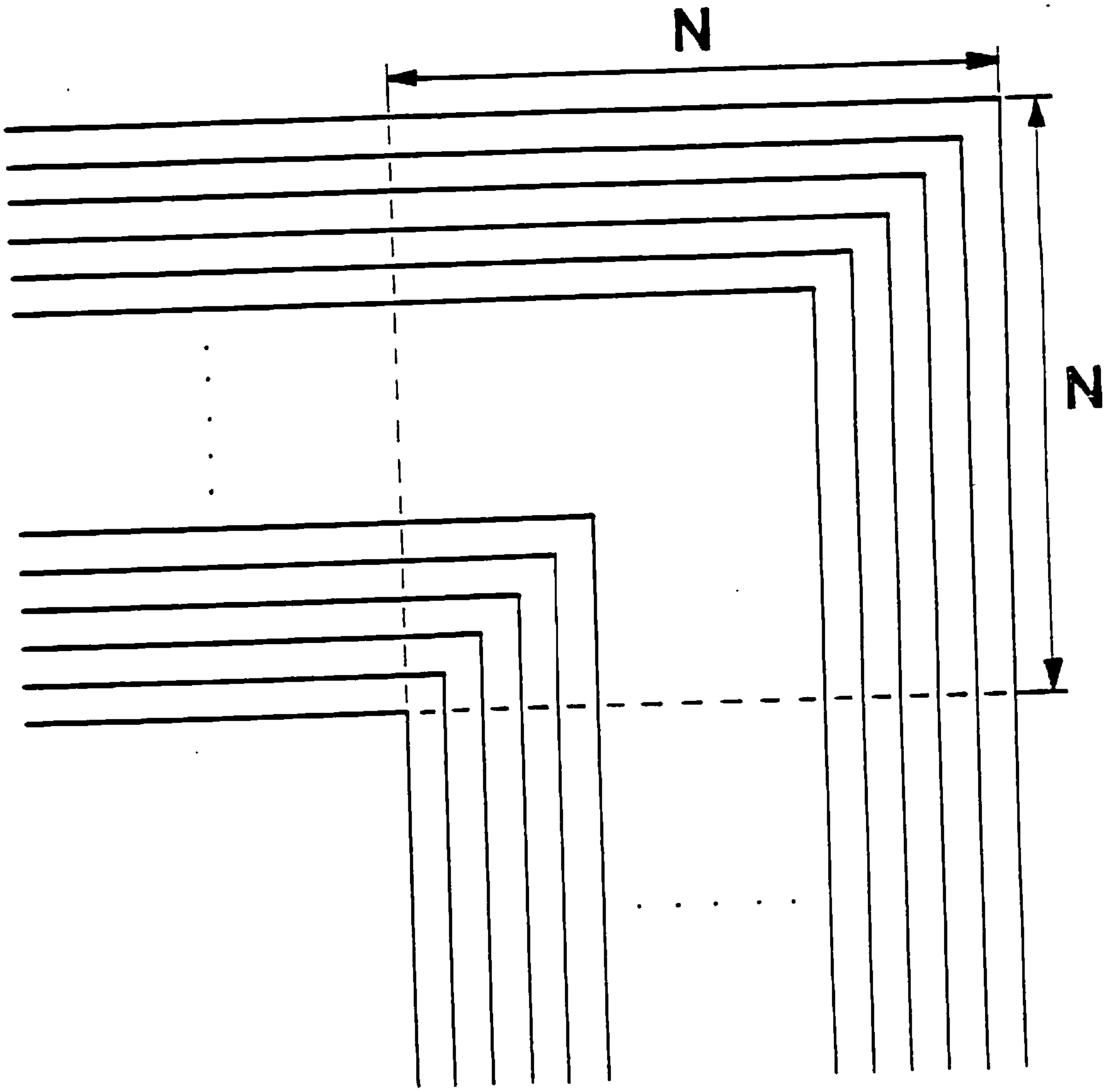

Figure 8: Routing of an N-Bit Data Bus through a 90-Degree Turn

quadratic penalty in area.

The PE instruction set provides another example of the sacrifice of execution speed within the individual PE in the interest of minimizing area, thus increasing the realizable throughput of the PPS as a whole. As we shall see shortly, the NON-VON PE executes a very small, narrow, and rather low-level set of instructions by comparison with the current generation of powerful 16- and 32-bit microprocessors. In particular, all PE instructions are eight bits long, including register operands and logical function codes. (In one case, however, the instruction is followed by a byte of data). In place of a rich set of relatively powerful instructions, we have chosen a few low-level operations having extremely simple realizations in hardware.

A single instruction typical of a contemporary 16-bit microprocessor might be implemented in NON-VON using a sequence of between one and four PE instructions. At the cost of a modest degradation in local execution speed, this strategy dramatically simplifies the complexity of (and hence, the area required for) the data path and PLA, and reduces the number of pins required to route instructions through the PPS chips.

3 Programming NON-VON

In this section, we introduce the NON-VON instruction set and describe the manner in which it is typically used in the course of programming. While a detailed discussion of each of the applications we have explored is beyond the scope of the current paper, some feeling for the kinds of techniques employed in constructing NON-VON programs is necessary to understand the basis for our architectural decisions. The remainder of this paper is thus devoted to an exposition of some of the techniques that characterize the NON-VON approach to parallel programming.

One "conceptual metaphor" we have found particularly useful in describing the principles underlying most NON-VON algorithms involves the notion of "intelligent records". This construct is explicated in the subsection immediately following our description of the instruction set. Next, we discuss the associative operations used to access intelligent records. In the fourth subsection, we describe and compare alternative techniques for the allocation and manipulation of records of various sizes (relative to the local storage capacity of a single PE). Finally, we illustrate the typical use of the techniques introduced in this section by informally describing NON-VON algorithms for a few simple symbolic and numerical applications.

3.1 The PE Instruction Set

The set of instructions executed by the NON-VON PE may be divided into six categories. The complete instruction set, grouped by category, is described below. Each instruction is followed by a brief specification of its semantics. The following symbols are employed:

<byte reg>   One of the eight 8-bit registers
                (A8, B8, C8, X8, Y8, Z8, IO8, or MAR)

<flag reg>   One of the eight 1-bit registers

	(A1, B1, C1, X1, Y1, Z1, IO1 or EN1)
<PE>	One of the physically or linearly adjacent PE's (P, LC, RC, LN, or RN)
<address>	An eight-bit address in the local RAM
<bit>	A one-bit constant
<byte>	An eight-bit constant

After the presentation of all instructions in a given group, a narrative description the typical use of each instruction is provided.

1. Register Transfer Group

OPCODE	OPERAND	SEMANTICS
LOADA8	<byte reg>	A8 <- <byte reg>
LOADB8	<byte reg>	B8 <- <byte reg>
LOADA1	<flag reg>	A1 <- <flag reg>
LOADB1	<flag reg>	B1 <- <flag reg>
STOREA8	<byte reg>	<byte reg> <- A8
STOREB8	<byte reg>	<byte reg> <- B8
STOREA1	<flag reg>	<flag reg> <- A1
STOREB1	<flag reg>	<flag reg> <- B1

The register transfer instructions are used to move data between the four accumulators (A8, B8, A1 and B1) and any of the other registers of compatible length. Note that the MAR may serve as the destination of an eight-bit STORE instruction, allowing different addresses to be stored in the MAR's of different PE's, and thus permitting simultaneous access to different locations in the local memories of different PE's, as described below. Similarly, it is worth noting that the value of EN1 may be changed from one to zero using an ordinary STORE instruction, allowing selected PE's to be disabled.

Note that transfers between arbitrary registers must be mediated by one of the accumulator registers, requiring two instructions instead of one. In the context of a massively parallel system, however, the fact that single-operand register transfer instructions are conveniently implemented in an eight-bit instruction word with very little area expended for control logic represents a significant compensating advantage.

2. Memory Access Group

READRAM	<address>	A8 <- RAM(MAR)
WRITERAM	<address>	RAM(MAR) <- A8

In order to transfer data between the local RAM and the A8 accumulator, the address of the RAM to be accessed must first be written into the eight-bit MAR register using an ordinary STORE instruction. Note that different PE's may access different RAM locations simulaneously, since the values in their respective MAR's need not be the same. This feature is essential to such applications as the parallel processing of variable-length records. The starting addresses of three variable-length fields might be stored in the first, second and third RAM locations within each PE, for example. In order to access the first byte of the second field of each record in parallel, the contents of RAM location two would be moved (by way of A8) into the MAR, and a READRAM instruction executed. Successive bytes in this field could then be accessed by performing parallel arithmetic on the address stored in the MAR.

3. Arithmetic and Shift Group

ADD1	A1 <- A1 xor B1 xor C1
	C1 <- (A1 and B1) or (A1 and C1)
	or (B1 and C1)

SUB1	A1 <- A1 xor (not B1) xor C1
	C1 <- (A1 and (not B1)) or (A1 and C1)
	or ((not B1) and C1)

ROTRA	Rotate A8 right by one bit through A1
ROTLA	Rotate A8 left by one bit through A1
ROTRB	Rotate B8 right by one bit through B1
ROTLB	Rotate B8 left by one bit through B1

While we have recently become quite interested in the implementation of parallel numerical algorithms on NON-VON-like machines, the rapid execution of purely numerical problems was not among the primary motivations for the NON-VON machine. Thus, although certain operations critical to NON-VON's typical modes of operation (data transfer and arithmetic comparison operations, for example) are performed eight bits at a time in NON-VON 1, all arithmetic operations other than comparison are performed in a bit-serial fashion.

Specifically, the ADD1 and SUB1 instruction perform one-bit addition and subtraction operations, respectively, as described earlier. Arithmetic on operands of arbitrary width are performed by repeated execution of these instructions. (Macros for eight-bit addition and subtraction, along with a number of other common sequences of PE instructions, are provided as part of the NON-VON 1 simulator.) The result is an ALU that, while fully general and extremely compact, is rather slow by comparison with conventional microprocessors in the performance of standard arithmetic operations.

In future versions of NON-VON, oriented toward the rapid execution of a wide range of numerical problems, we plan to experiment with the implementation of somewhat faster, albeit more area-expensive ALU's. It should be noted, however, that in many common data processing applications -- performing the same computation on a large number of records, for example, or computing such quantities as the mean or variance of selected fields -- the ability to perform a million or so arithmetic operations in parallel should push even NON-VON 1's effective throughput several orders of magnitude beyond those of todays fastest supercomputers.

The four rotate instructions treat the A8 and A1 registers (and similarly, the B8 and B1 registers) together as a nine-bit circular shift register. Specifically, ROTRA shifts all but the low-order bit of A8 into the next lowest bit position within A8; the low-order bit of A8 is moved into A1, and the value previously stored in A1 is moved into the high-order bit of A8. ROTLA similarly performs a left circular shift of the combined A8 and A1 registers, while ROTRB and ROTLB perform analogous shifts on the B8 and B1 registers. In combination with the one-bit logical function operations (discussed below), these instructions permit the execution of arbitrary operations involving eight-bit operands on a bit-serial basis.

4. Logical Function Group

LOGICAL <operation> A1 <- (A1 <operation> B1)
(where <operation> is a four-bit code specifying one of the sixteen possible boolean functions of two single-bit variables)

CLEAR	A1 <- 0
SET	A1 <- 1
NEGATE	A1 <- not A1
AND	A1 <- A1 and B1
OR	A1 <- A1 or B1
XOR	A1 <- (A1 and (not B1)) or ((not A1) and B1)
EQU	A1 <- (A1 and B1) or ((not A1) and (not B1))
NAND	A1 <- not (A1 and B1)

since the semantics of this operation would be undefined if both children of that parent were enabled. Thus, only LC, RC, LN and RN are legal operands for the SEND8 instruction. It should be noted, however, that the parent is capable of receiving data from it's children through the use of RECV8 LC and RECV8 RC instructions. The semantics of the SEND8 and RECV8 instructions are not immediately apparent in the case where the operand PE is currently disabled. In such cases, it is the recipient's status, and not that of the originator, which determines whether data is in fact transferred. Specifically, it is always possible to RECV data from a PE, regardless of whether it is enabled, but an attempt to SEND data to a disabled PE will not result in a transfer of data.

The SEND1 and RECV1 instructions function in precisely the same way as SEND8 and RECV8, but operate on flag operands instead of byte-wide values.

6. No Operand Group

ENABLE	EN1 <- 1 in all PE's, including those previously disabled

COMPARE	if A8 = B8 then A1 <- 1; otherwise A1 <- 0
	if A8 > B8 then B1 <- 1; otherwise B1 <- 0

RESOLVE	A1 <- 0 in all PE's except "first" PE
	where A1 = 1
	if no PE has A1 = 1,
	logical register R1 (in CP) <- 0;
	otherwise R1 <- 1

A PE may be disabled by transferring a 0 into its EN1 register using an ordinary STOREA1 EN1 (or STOREB1 EN1) instruction. In a typical application, the contents of A1 (or B1) will be set to the result of some boolean test prior to the execution of such a store instruction, resulting in the selective disabling of all PE's for which the test fails. This technique supports the "conditional" execution of a particular code sequence. Following the execution of such a sequence, an ENABLE instruction is issued to "awaken" all disabled PE's. In combination with appropriate register transfer and logical operations, this approach may be used to implement more complex conditionals, including nested "IF-THEN-ELSE" constructs.

The COMPARE instruction sets the A1 flag to 1 if the contents of A8 and B8 are the same, and the B1 register to 1 if the contents of A8 exceed that of B8. By combining the two bit accumulator values using the appropriate logical instructions, it is thus possible to perform any of the six possible arithmetic relational tests ("equal to", "not equal to", "greater than", "greater than or equal to", "less than", or "less than or equal to") on the values in the byte accumulators. The result may then be used to selectively disable certain processors, allowing the use of general arithmetic tests within a conditional.

The most common use of the COMPARE instruction, however, is in the execution of content-addressable operations. As we shall see shortly, such operations are realized by broadcasting character strings or numeric values throughout the PPS, comparing them in parallel with the contents of all enabled PE's, and disabling those for which the match criteria are not satisfied. The decision to implement the COMPARE instruction using byte-wide comparator hardware was based in large part on the central role played by such content-addressable operations in most NON-VON algorithms.

The RESOLVE instruction is used in practice to disable all but a single PE, chosen arbitrarily from among a specified set of PE's. First, the A1 flag is set to one in all PE's to be included in the candidate set. The RESOLVE instruction is then executed, causing all but one of these flags to be changed to zero. (Upon executing a RESOLVE instruction, one of the inputs to the CP will become high if at least one candidate was in fact found in the tree, and low if the candidate set was found to be empty. In our simulator, this condition code is stored in the "logical register" R1, which may be thought of as existing within the CP.) By issuing a STOREA1 EN1 command, all but the single, chosen PE may be disabled, and a sequence of instructions may be executed on the chosen PE alone. In particular, data from the chosen PE may be communicated to the CP through a sequence of LOAD and REPORT commands.

If the candidate set is first saved (using another flag register in each PE), each of the candidates can be chosen in turn, subjected to individual processing, and removed from the candidate set, allowing the sequential processing of all candidates. Typically, the individual processing performed for each chosen candidate involves the broadcasting of information contained in, or derived from, that candidate to other PE's within the PPS. This paradigm for sequential enumeration is thus employed as a sort of "outer loop" in a number of highly parallel NON-VON algorithms, including the algorithm for set intersection described in Subsection 3.5.

In the NON-VON 1 prototype, the A1 flag is preserved in that PE which would be assigned the lowest number in an inorder enumeration of all nodes in the PPS tree. The use of inorder enumeration as a criterion for selecting a single PE is an artifact of the NON-VON 1 hardware design, however, and is not guaranteed by the instruction set. The RESOLVE function is implemented using special combinational hardware, embedded within the I/O switch, that propagates a series of "kill" signals in parallel from all candidate PE's to all higher-numbered PE's in the tree. As is the case for all of the global communication functions, the RESOLVE operation is very fast; hundreds of thousands of candidates might be "killed" in less than a microsecond in NON-VON 1, for example.

3.2 The "Intelligent Record" Metaphor

A large share of the data processing applications for which computers are now used involve operations on files that consist of a relatively large number of comparatively small records. In many such applications, the relevant files may greatly exceed the capacity of the primary storage device. While the design of NON-VON's SPS, and its interface to the PPS, were in fact based largely on the essential characteristics of such large-scale data processing tasks, our concern in the following discussion will be with the case in which all records are stored in the PPS. Briefly stated, the NON-VON approach to parallelizing this sort of record-processing application is based on a "nearly one-to-one" physical association of PE's and records. In such applications individual records are often, in effect, capable of manipulating their own contents in parallel. This observation suggests the notion of an "intelligent" record.

As we shall see shortly, NON-VON is designed to support the massively parallel manipulation of records that may be considerably larger or smaller than the local storage available within each PE. Furthermore, the high-level languages we are now developing for use on NON-VON permit the precise mapping between records and PE's to be made *invisible* to the user in most applications. The user-transparency of this mapping is in fact a critical aspect of NON-VON's support for the intelligent record concept, since it insulates the programmer from the details of the hardware, allowing each user-defined logical record to be treated as if it had its own private processor.

As an alternative to the intelligent record metaphor, the reader may wish to think in terms of the equivalent notion of "virtual PE's", each consisting of a single processor and an amount of local memory just sufficient to store a single record of arbitrary size.

3.3 Associative Operations on the NON-VON Machine

Before examining the manner in which NON-VON's hardware supports records of arbitrary size, let us consider the fundamental mechanisms employed in accessing and manipulating intelligent records. In contrast with a conventional coordinate-addressable computer, whose primitive instructions access its data by address, NON-VON may be considered a content-addressable machine, in which data is accessed on an associative basis. In order to illustrate the manner in which records may be accessed by content, let us consider an example in which each PE contains a single "employee record" containing fields for the name, department, years of service, and salary of the employee in question. (Some of these fields will be used in a later example.)

Suppose we wish to associatively identify the records of all employees in the sales department, and to perform some operation on all such records (either concurrently or in succession). Let us assume that the department name is stored in a five-character field beginning in the 17th location within each local RAM, and that all PE's containing an employee record are initially enabled. We now broadcast the first character in the specified department name, which is an "S", to all PE's. Each PE compares this character with the contents of its 17th RAM location, and disables itself if the two are not equal. The precise sequence of PE instructions follows:

```
BROADCAST8 "S"          ; Send the pattern character
STOREA8 B8              ;   and save it in B8
READRAM 17              ; Get the data character
COMPARE                 ; Do they match?
STOREA1 EN1             ; If not, disable this PE
```

Using a similar set of instructions, the second character is broadcast and compared with the 18th location in the local RAM of each enabled PE. After the execution of five such code sequences, only those PE's whose DEPARTMENT fields contain the string "SALES" will remain enabled. It should be noted that this process of associative marking requires time dependent only on the length of the pattern string, and independent of the number of employee records. Furthermore, the values of any combination of fields may be used as criteria for success of the associative marking operation.

In the case where different PE's are used for the storage of different types of records, operations on a given record type must be preceded by the disabling of all PE's but those containing records of that type. To facilitate this process, each record is "tagged" internally to indicate its record type. If there are only a few distinct record types, the records can be tagged by associating a different one-bit register with each record type, and setting its value to 1 in exactly those PE's containing records of the type in question. In order to enable all records of a given type, the bit contained in the appropriate flag register is simply transferred to EN1 using two register transfer instructions. For a larger number (up to 256) of record types, a distinct "tag byte" is associated with each record type, and stored in the same way as the fields of the record itself. A single BROADCAST and COMPARE sequence, followed by a STOREA1 EN1 instruction, may be used to disable all PE's except those containing records of the desired type.

Depending on the application, associative marking is typically followed by one of two operations. The first, and most common, is to perform a sequence of operations in parallel on the records contained in each of the associatively identified PE's. The second involves sending the "marked" records (or selected fields thereof) one at a time to the CP in an arbitrary sequence, using the RESOLVE and REPORT instructions. The latter operation, when applied to associatively identified records, is called associative enumeration. It should be noted that the time required for associative enumeration, while proportional to the number of "matching" records, is independent of the total number of records in the file. Both of the above applications of associative marking will be illustrated shortly in the context of particular NON-VON algorithms.

It is of course the case that either a conventional computer or a NON-VON-like machine (and indeed, any device with the power of a Turing machine) is capable of emulating the behavior of either a content- or coordinate-addressed machine. In particular, a conventional system can implement associative operations using only coordinate-addressable primitives by employing one of several well-understood partial match algorithms. Because they must provide for retrieval based on any of the 2^k possible combinations of k fields, though, such algorithms are associated with significant costs in time, space and conceptual complexity.

Conversely, NON-VON is capable of addressing data on a coordinate basis whenever the data under consideration is best understood in terms of an "address-like" numbering scheme. In such applications, coordinate values are explicitly stored as part of each intelligent record and associatively probed to obtain the record corresponding to a given address. This technique is employed in a number of parallel matrix algorithms, for example.

What, then, are the essential differences between NON-VON's addressing capabilities and those supported by a conventional von Neumann computer? From a software perspective, the critical point is that NON-VON uses a numerical addressing scheme only when the problem at hand is most easily described in terms of a coordinate system. In the case where records are more naturally identified by content, the programmer is relieved of the responsibility of translating his or her intentions into an artificial coordinate-based descriptive formalism.

It is our contention that the great majority of the computer applications encountered to date are most naturally described in terms of content-addressable, as oppposed to coordinate-addressable primitives. While our argument is perhaps strongest for the kinds of "business-oriented" data processing tasks that presently account for most of our society's expenditures for large-scale computing, we believe that a surprising number of "scientific" applications might also be more easily specified in content-addressable terms. By providing direct, low-level support for associative operations, NON-VON effectively shortens the path between the description and implementation of many common computational tasks, thus simplifying the task of programming.

The other essential advantage of NON-VON's hardware support for content-addressability, of course, relates to the time required for associative operations. In practice, NON-VON might provide as much as several orders of magnitude improvement over the fastest associative retrieval operations on a conventional computer system, without the need for complex, time-consuming, and area-expensive indexing or hashing operations.

3.4 Packed and Spanned Records

Up to this point, we have considered the case in which exactly one record is stored in each PE. Let us now consider the manner in which records considerably smaller or larger than the capacity of a single local RAM may be efficiently stored and manipulated within the NON-VON PPS. The former case involves the allocation of more than one record per PE, a scheme we call packed record allocation. To illustrate the manner in which small records may be packed, let us consider an application in which it is desirable to pack as many fifteen-byte records as possible into the PPS at once. (Although records of this size would be uncommon in most symbolic applications, they might well occur in, say, a sparse matrix manipulation or signal processing problem.)

Four such records might be stored in each PE, beginning in local RAM locations 1, 16, 31 and 46. We will use the term record slice to refer to a set of packed records stored in the same position within their respective PE's. (In our example, four record slices are defined.) In general terms, each operation to be performed on a packed record is carried out by issuing a separate set of PE instructions for each record slice. In order to move a single byte from the fifth to the seventh location of each of our fifteen-byte packed records, for example, we would first execute the sequence

```
READRAM 5
WRITERAM 7
```

followed by the sequence

```
READRAM 20
WRITERAM 22
```

and then by analogous sequences of instructions corresponding to the last two record slices. The high-level languages now under development for use on NON-VON are intended to relieve the programmer of the responsibility for such operations. In our Pascal-based language, for example, the user would simply declare the collection of records to be of type PACKED MULTIPLE RECORD; a subsequent assignment statement involving two fields of that record would be compiled into the four sequences of instructions discussed above.

Not all operations on packed records, though, are so simply handled. In the above example, the A8 register is used only for temporary storage of the value to be transferred, and need not be preserved after the transfer is completed for a given record slice. In general, however, the contents of certain flag and byte registers may have to be saved prior to operations on successive record slices. The question of how best to reduce the overhead involved in such "state-saving" operations is one of the more interesting considerations involved in the design of compilers for NON-VON.

While packed records may be quite useful in some applications, it should be noted that the space saved by packing is at best proportional to the increased time required to broadcast each instruction to all slices. An additional disincentive is provided by the significant compile- and execution-time overhead required for the support of operations on packed records. For these reasons, small records are packed only when this option is explicitly chosen by the programmer, based on the relative importance of time and space in the context of a given application.

In the case of records too large to fit within a single PE, each record is split among several PE's according to one of two schemes. The first, referred to as the linear allocation method, splits each record among several linearly adjacent (logical) neighbor PE's. The other, which we call bush allocation, stores each record in a distinct "tree-shaped" cluster of physically proximate PE's called a bush. In order to illustrate these schemes, let us consider an example involving records 150 bytes in length. Under either allocation scheme, each spanned record is split among three physical PE's. We will refer to the first part of each record as segment A, the second as segment B, and the third as segment C.

Using one of the "tagging" techniques introduced above, all PE's containing the A segment of a record are marked with one tag, those containing B segments with another tag, and those containing C segments with third. In algorithms requiring no parallel communication between different segments of a spanned record, the A, B, and C segments are treated as if they were distinct record types, only one of which is enabled at any given point in time. As we shall see shortly, algorithms in which activation (the state of being enabled) and data must be transferred in parallel between one segment and another within each record raise a number of more interesting issues. Parallel intersegmental transfers are handled differently (and with different average-case time complexity) in the case of linear and bush allocation. We begin with a discussion of the former technique.

In a linear allocation of our hypothetical 150-byte records, segment A might be assigned to the first PE in the linear sequence used for linearly adjacent neighbor communication (as described in Section 2). Segment B of the first record would be stored in that PE having linear number two, while segment C would be stored in the "linear three" PE. Segments A, B and C of the second record would then be assigned to the linear four, five and six PE's, respectively. The third record would be similarly split among the linear seven, eight and nine PE's, and so on. It should be recalled that two PE's that are logically adjacent in the linear sequence are not necessarily physically adjacent in the PPS tree. Thus, a single record may be split among PE's that are not physically contiguous, leading to a physical interleaving of records within the PPS. The inorder embedding employed in NON-VON 1, for example, would lead to the allocation shown in Figure 9. (The PE's are labelled with the record number and segment of the data; segment B of record 3, for example, is labelled 3B.)

To see how linearly allocated spanned records might be manipulated in the course of an actual application, let us suppose that our sample records each describe one of the employees in our earlier example. Assume also that the first two characters of the DEPARTMENT field are stored in segment A and the remainder in segment B, and that the salary field is stored entirely within segment C. Now suppose that we wish to raise the salary of all employees in the sales department by 10% in a single parallel operation. Earlier in this section, we presented an informal description of an algorithm for associatively marking each such employee record in the case of one-to-one allocation. After disabling all PE's except those containing A segments, we employ this algorithm to disable all enabled PE's except those having "SA" as

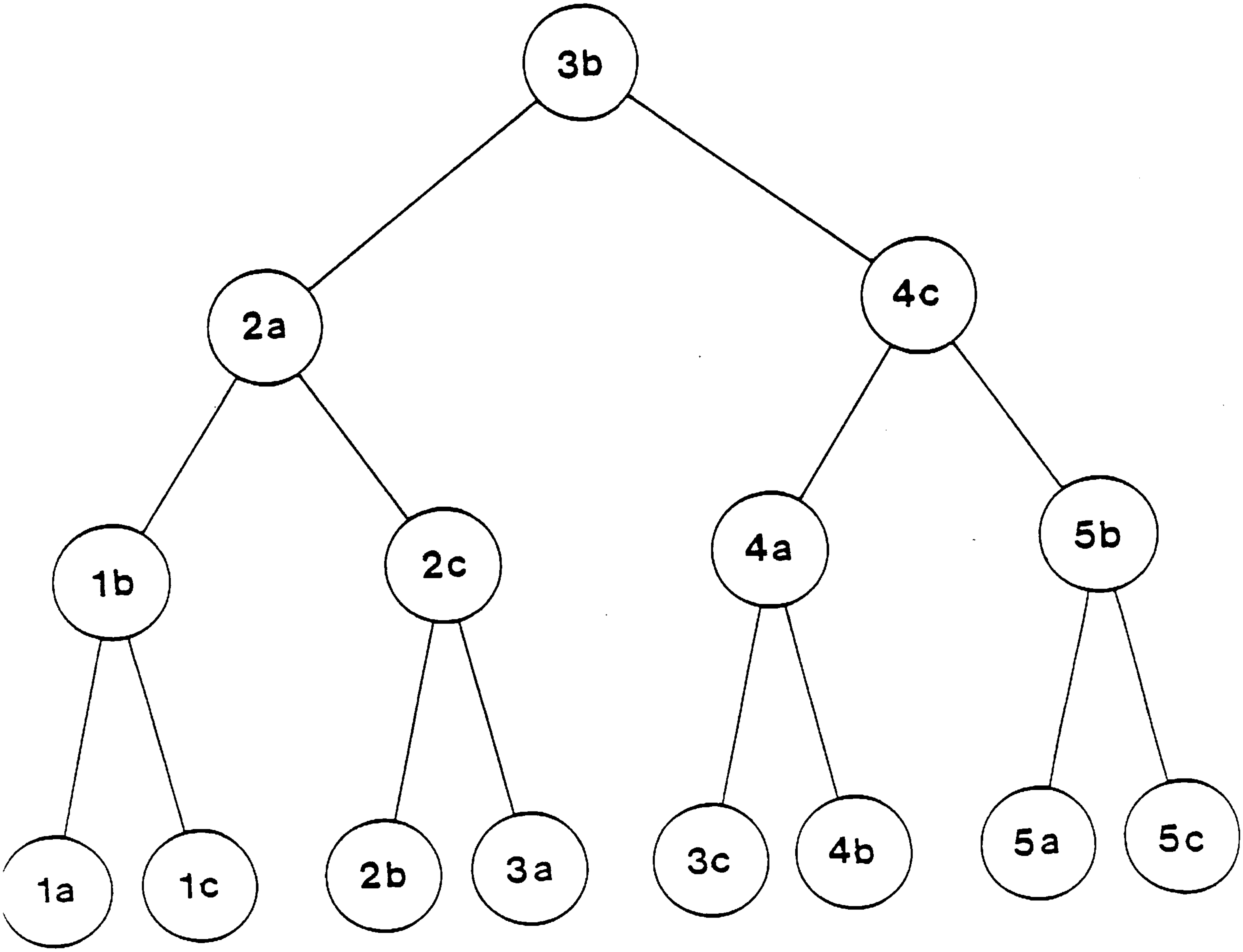

Figure 9: Linear Allocation of Spanned Records

the first two characters of their DEPARTMENT field.

At this point, each PE that remains enabled transfers activation to its right linear neighbor. This step is realized through the use of a code sequence that includes a SEND1 RN instruction, which concurrently communicates a boolean value from each PE to its linear neighbor. At the end of this sequence, which will not be detailed here, the B segments of all records whose DEPARTMENT fields begin with "SA" are enabled, and all A (and C) segments are disabled. The characters "LES" are now matched against the corresponding characters in all enabled records, leaving enabled only the B segments of all records corresponding to employees in the sales department. Activation is now propagated to the C segments of all such PE's, and a sequence of instructions issued to increase the salary fields of all such records by 10%.

In contrast with the linear allocation scheme, the technique of bush allocation groups all segments of a given record together physically within the PPS, as shown in Figure 10. Each of the "tree-shaped" clusters of PE's enclosed within a rectangle in Figure 10 is called a bush. Within a given bush, successive record segments are assigned to PE's according to the bounded-neighborhood mapping introduced in Section 2.3. The precise manner in which record segments are allocated within a bush, and bushes within the PPS tree, is presented elsewhere [18].

Bush-allocated spanned records are manipulated in much the same way as their linearly-allocated counterparts, but using the direct physical tree connections in place of the indirect linear pathways for the parallel propagation of data and activation. In our example application, the first two characters of the string "SALES" are matched concurrently in all of the "A" PE's shown in Figure 10. Each matching PE then enables its parent (a "B" PE) using a RECV1 LC instruction. Upon completion of this matching operation, each PE still enabled executes a code sequence including a SEND1 RC

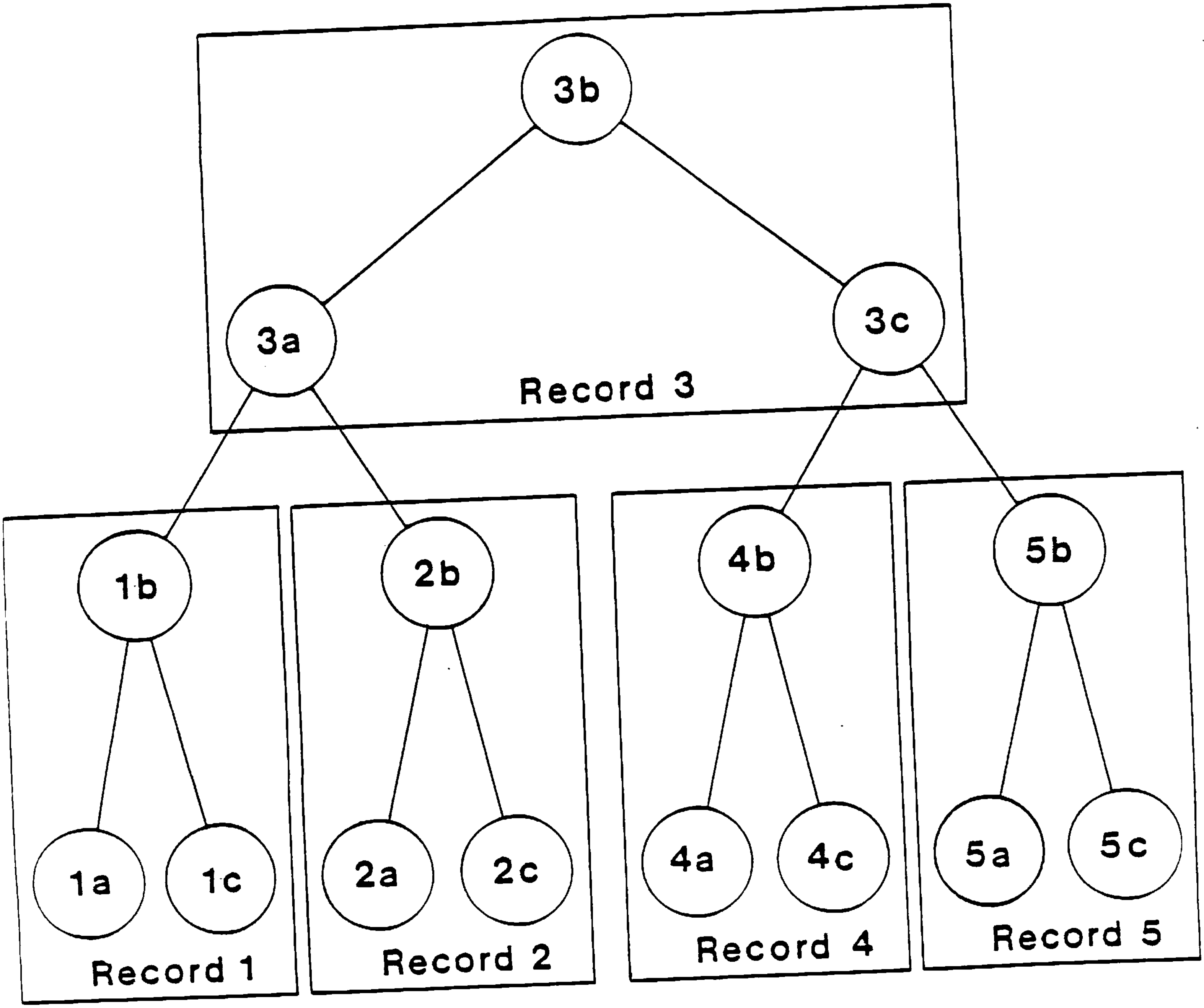

Figure 10: Bush Allocation of Spanned Records

instruction to enable its right child (a "C" PE), which then increases its salary field by 10%. As in the case of linear allocation, the transfer of data and activation between segments is fully parallel.

There are certain time/space tradeoffs involved in the choice of linear or bush allocation for spanned records, however. Let us first compare the space required for these two allocation methods. The linear allocation method makes progressively more efficient use of the available local RAM as the number of PE's spanned by each record increases. In particular, we would expect to waste only half the space of a single local RAM (32 bytes in NON-VON 1) per stored record in the average case. This small amount of waste is due to the requirement that the beginning of each record be aligned with the beginning of some PE's local RAM, at least in the method for parallel memory accesses we have outlined. Asymptotically (with increasing record length), the proportion of total available RAM wasted due to alignment thus approaches zero.

By way of comparison, this "waste factor" approaches 25% in the case of bush allocation. To gain an intuitive appreciation for the reason for this comparative inefficiency, consider the case of a spanned record just large enough to require 2^m PE's for storage. The smallest bush capable of storing such a record would contain $2^{m+1} - 1$ PE's, resulting in a waste of $2^m - 1$ PE's worth of RAM (in addition to an alignment penalty), or approximately half of the total available RAM, for large records. It is easily seen that the average case waste factor must fall midway between this 50% asymptotic worst case value and the best case value of no waste, which occurs for records consuming $2^m - 1$ PE's worth of RAM. Thus, linear allocation is more space-efficient than bush allocation, particularly in the case of large spanned records.

The space advantage offered by the linear allocation scheme, however, comes at the cost of an increase in the time complexity of data and activation transfers among record segments. Note that in the worst case, the data in question must be transferred from the first to the last PE in the record (with respect to the ordering imposed for purposes of linear neighbor communication). The number of instructions required for such a transfer thus varies linearly with record length in the worst (and, in fact, in the average) case. In the case of bush allocation, on the other hand, the worst case occurs when data must be passed between two leaves of a bush. On the average, such transfers require time logarithmic in the size of the record, a significant advantage in the case of large records. In the case of tranfers between _successive_ record segments, the bounded-neighborhood ordering reduces this time to a constant.

One other point is worthy of mention in connection with the choice of allocation method. First, we note that binary tree algorithms such as those described by Browning [4] can only be _directly_ implemented on NON-VON when one-to-one allocation is possible (that is, where records are no larger than the capacity of a single local RAM, and each is allocated to a different PE). Many of these algorithms, however, can be easily (and in some cases, "mechanically") adapted to apply to m-ary trees. (One important class of such algorithms will be described shortly.)

If bush allocation is chosen, such transformed algorithms can be applied to spanned records of arbitrary size, providing the bushes themselves are allocated within the PPS tree in such a way as to preserve an m-ary tree structure for purposes of inter-record communication. This requirement is satisfied by a particular kind of bush allocation called _landscaped allocation_ (discussed in [18]), in which the bushes are configured as an m-ary tree. While a thorough discussion of algorithms for landscaped bushes is beyond the scope of the present paper, the basic approach involves choosing m to be the number of leaves per bush, and treating each bush as a single node in an m-ary tree, where $m = 2^k$ for some positive integer k. (The set of bushes depicted in Figure 10 is in fact landscaped, forming a five-node quaternary tree.)

In the case of linear allocation, no such transformation is possible, since record segments are interleaved throughout the PPS. The ability to execute many parallel algorithms intrinsically tied to a tree-structured topology thus constitutes another significant advantage of bush allocation.

3.5 Examples of Symbolic and Numerical Algorithms

In order to illustrate some of the more important techniques used in the course of applications programming, we now consider a few simple NON-VON algorithms. First, we describe a highly parallel algorithm for computing the intersection of two sets. This algorithm is based on a commonly used NON-VON programming technique involving a combination of associative enumeration and parallel matching, and is closely related to the algorithms for a number of other set theoretic and relational database operations.

Next, we introduce an important technique for the massively parallel execution of algebraically associative operations. Using this technique, such quantities as the sum, maximum or mean of n numbers may be computed in O(log n) time. We then consider NON-VON's application to a rather "un-NON-VON-like" task: the simulation of large-scale physical systems. We conclude by mentioning a few other examples of symbolic and numerical applications we have considered for parallel implementation on the NON-VON machine.

In general terms, the intersection of two sets of is performed by sequentially enumerating the elements of the smaller set, and performing one associative probe for each such element to determine if it is also present in the larger set. Suppose, for example, that we wish to intersect two sets of strings, each stored in its own "virtual PE" (which may be realized using either one-to-one, packed or spanned records). As in most NON-VON algorithms, these strings may be located anywhere within the PPS, since all accesses are made on a content-addressable basis. The elements of the two sets are distinguished only by tagging, and may in fact be arbitrarily intermingled.

First, we enable all elements of the smaller set by associatively marking those having the appropriate tag. An arbitrary one of these elements is then sent to the CP using the RESOLVE and REPORT instructions, and marked so that it will not be chosen again. This value is then matched against all elements of the larger set in parallel, and a RESOLVE instruction executed to see if that string is present. If it is, the element is included in the result set. This procedure is repeated for all elements in the smaller set not already marked as having been processed. The running time of this algorithm is linear in the cardinality of the smaller set, and independent of the size of the larger one. The union or difference of two sets may be constructed in a similar manner.

It is interesting that some of the best algorithms known for set intersection on a von Neumann machine (the hashed intersection algorithms described by Trabb-Pardo [20], for example) may in fact be viewed as software emulations of the associative approach employed in our algorithm. While we have chosen set intersection to illustrate the "enumeration and probing" paradigm for pedagogical reasons, NON-VON in fact offers more significant advantages in the case of certain "more difficult" operations, whose implementation on a von Neumann machine may in practice be quite expensive. One example having particular importance in relational database management applications is the equi-join operation [5], of which set intersection may in fact be considered a degenerate case.

The tree-structured topology of the PPS is essential to many aspects of NON-

VON's operation, and thus plays an important implicit role in all of the algorithms we have discussed so far. None of these algorithms, though, have made explicit use of the tree connections. A simple example of an algorithm in which explicit physical tree communication plays an important role is the problem of adding a large number of numeric values, each stored in a distinct "logical record". We might wish, for example, to determine the total yearly payroll of our hypothetical firm by adding the salary fields of all employees.

In the interest of simplicity, let us first consider the case in which each PE in the PPS contains exactly one employee record. First, we disable all nodes except those which are the parent of some leaf node. (This is easily accomplished in constant time using an algorithm that exploits the fact that the leaves are the only nodes that can not receive a message from any descendant node.) Each of these "penultimate" nodes is then (concurrently) instructed to obtain the salary of its left child (using a sequence of RECV8 LC) instructions, and to add this value to its own salary field.

The process is repeated for all right children, at which point each penultimate node holds the sum of its own salary and those of its two children. At this point, the parents of all penultimate nodes are enabled, and all other PE's disabled; the entire procedure is then repeated. After (log n - 1) such steps, the root node will contain the sum of the salaries of all (n - 1) employees. In a full-scale NON-VON prototype containing a million PE's, we would expect the effective execution speed for such a problem to be on the order of tens of billions of arithmetic operations per second.

By substituting other algebraically associative operations in place of addition, this algorithm can be adapted to compute many other values of practical importance. The mean or maximum salary paid to any employee, for example, can be similarly computed in logarithmic time. Such operations can of course be combined with the techniques described earlier for the associative identification of records satisfying various criteria, allowing, say, the parallel computation of the average salary paid to employees in Department C who have been employed for between 3 and 5 years.

Finally, it should be noted that such algorithms are easily generalized to support packed records. In our example, we would first add the salary fields of all record slices, leaving a single combined salary in each PE, at a cost proportional to the packing factor. The algorithm for one-to-one addition could then be applied without modification.

Spanned records can also be accomodated, but only when landscaped allocation is employed. In order to adapt our algorithm to the case of landscape-allocated spanned records, we treat each k-level bush as a node in 2^k-ary tree. The descandants of such a node are precisely the children of all leaves of the bush in question. In Figure 10, for example, the bush containing the root node is considered to be the root of a two-level tree with a fan-out of four. Each of the four other bushes in the tree are treated as leaves of this quaternary tree. In the modified algorithm, each bush adds the salaries of <u>each</u> of its "descendants" into a running sum; after approximately $\log_k n$ such steps, the bush containing the root node contains the sum of all salaries.

In order to convey some feeling for the diversity of applications for which NON-VON may provide substantial performance improvements, we now consider a problem which might first appear to be poorly suited for execution on a tree-structured machine. This application, while of only modest economic importance by comparison with conventional business data processing tasks, has for some time dominated the attention of most designers and users of "conventional" supercomputers. Although NON-VON was in fact designed to have its primary impact within the mainstream of business computing, we will succumb to the temptation to discuss its application to this more glamorous scientific application. The task to which we allude is the simulation of large-scale three-dimensional physical systems.

One technique employed in many such simulation problems uses a large number of records (often on the order of a million), each corresponding to a small cubical region in the space being simulated. Each record would typically contain a small number of scalar or vector variables (temperature or fluid velocity, for example) whose values are known to change over time according to certain physical laws involving largely local interactions. The behavior of the system is simulated by repeatedly applying the following two-step process:

1. The communication step. The values of certain variables at a given point are communicated to adjacent and "nearly adjacent" neighbors.
2. The computation step. A new value is computed for each point in the system, based on the values of variables at neighboring points.

Typically, the same numerical operations are performed at all points during each computation step. The two-step cycle is generally repeated many times to simulate the evolution of a physical system over time.

Although it was certainly not designed with this sort of task in mind, the NON-VON architecture would in fact appear to offer significant asymptotic advantages over existing supercomputer designs in the solution of such problems. Not surprisingly, NON-VON permits the computational component of such problems to be solved in time independent of n, the number of points being simulated. To do so, each "cube" of the space being simulated is associated with a distinct virtual PE, and the sequence of operations is broadcast to all such cubes for concurrent execution.

More interesting is the fact that NON-VON permits an $O(n^{1/3})$ speedup in the communication component as well. The algorithm used for communication depends on the a particular scheme for allocating the primitive cubes among the leaves of the PPS tree in such a way that the nodes at progressively higher levels correspond to progressively larger cubes. While the details of this algorithm are beyond the scope of the current paper, NON-VON's asymptotic speedup is based on the fact that the amount of data passing through each internal node is proportional to the surface areas of these recursively constructed cubes, and not to their volumes. The time complexity of a single communication step is thus $O(n^{2/3})$, and not $O(n)$, as in the case of a von Neumann machine.

While scientific computing applications have not been central to our design goals, we have investigated the potential application of the NON-VON architecture to a number of numerical problems. Among the applications we have explored are a number of signal processing, matrix manipulation, graphics and image processing problems. NON-VON's content-addressable primitives permit significant absolute and asymptotic speedups in a number of array processing applications, but provide particularly natural and efficient support for problems involving the manipulation of sparse matrices.

If numerical applications were expected to constitute a large share of the workload of such a machine, the incorporation of a full eight-bit ALU within each PE would almost certainly be warranted, even at the expense of a modest decrease in processor density. Such a change would alter neither the basic NON-VON architecture nor the essential structure of the algorithms we have developed.

Space does not permit a detailed discussion of all of the applications for which we have designed algorithms (at various levels of detail) for NON-VON. It is worth mentioning, though, that the NON-VON PPS supports the execution of several linear-time sorting algorithms, and that at least one promising technique for rapidly sorting very large files is currently under investigation. Highly efficient parallel algorithms for simple transaction processing, and for a number of other operations critical to large-scale commercial data processing, have also been explored. Although we have thus far attacked only a small sampling of the problems to which "real world" computer systems are applied, it has been our experience that most such largely symbolic applications prove amenable to massive parallelization on the NON-VON machine.

References

[1] B. Arden, "Analysis of Chordal Ring Networks", in IEEE Transactions on Computers, vol. C-30, pp. 291-301, April 1981.

[2] J. Backus, "Can Programming be Liberated From the Von-Neumann Style? A Functional Style and its Algebra of Programs", in Communications of the ACM, vol. 21, no. 8, pp. 613-641, August, 1978.

[3] S. Browning, "Hierarchically Organized Machines", in C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1978.

[4] Browning, Sally, "The Tree Machine: A Highly Concurrent Computing Environment" Ph.D. Thesis and Technical Report #3760, California Institute of Technology, January, 1980.

[5] E. F. Codd, "Relational Completeness of Data Base Sublanguages", in R. Rustin (ed.), Courant Computer Science Symposium 6: Data Base Systems, Prentice-Hall, Inc., 1972.

[6] M. Flynn, "Some Computer Organizations and their Effectiveness", in IEEE Transactions on Computers, vol. C-21, pp. 948-960, September, 1972.

[7] C. Hewitt, "Design of the APIARY for Actor Systems", in Conference Record of the 1980 LISP Conference, pp. 107-118, August, 1980.

[8] D. E. Knuth, The Art of Computer Programming, vol. 1: Fundamental Algorithms, Addison-Wesley, 1969.

[9] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)", in Sparse Matrix Proc. 1978, Society for Industrial and Applied Mathematics, pp. 256-282, 1979.

[10] F. Leighton, Layouts for the Shuffle-Exchange Graph and Lower Bound Techniques for VLSI, Ph.D. Thesis, Massachussetts Institute of Technology, August, 1981.

[11] C. E. Leiserson, Area-Efficient VLSI Computation, Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, October 1981.

[12] F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A versatile Network for Parallel Computation", in Communications of the ACM, vol. 24, no. 5, May 1981, pp. 300-309.

[13] M. Sehanina, "On an ordering of the set of Vertices of a Connected Graph", in Publications of the Faculty of Science of the University of Brno,

no. 412, pp. 137-142, 1960.

[14] C. H. Sequin, A. M. Despain, and D. A. Patterson, "Communication in X-Tree, a Modular Multiprocesser System", in Proceedings of the Annual Conference of the ACM, Washington, D.C., December, 1978.

[15] D. E. Shaw, "A Hierarchical Associative Architecture for the Parallel Evaluation of Relational Algebraic Database Primitives", Stanford Computer Science Department Report STAN-CS-79-778, October, 1979.

[16] D. E. Shaw, "A Relational Database Machine Architecture", in Proceedings of the 1980 Workshop on Computer Architecture for Non-Numeric Processing, Asilomar, California, March, 1980.

[17] D. E. Shaw, Knowledge-Based Retrieval on a Relational Database Machine, Ph.D. Thesis, Department of Computer Science, Stanford University, 1980a.

[18] D. E. Shaw and B. K. Hillyer, "Allocation and Manipulation of Records in the NON-VON Supercomputer", Columbia Computer Science Department Report, August, 1982 (in preparation).

[19] C. Thompson, "A Complexity theory for VLSI", Ph.D. Thesis, Carnegie-Mellon University, August, 1980.

[20] L. Trabb-Pardo, Set Representation and Set Intersection, Ph.D. Thesis and Stanford Computer Science Department Report STAN-CS-78-681, December, 1978.