Non-Uniform Region Processing on SIMD Arrays Using the Coterie Network

M. Herbordt, C. Weems
M. Scudder

COINS TR92-02

January 1992

Non-Uniform Region Processing on SIMD Arrays Using the Coterie Network*

Martin C. Herbordt Charles C. Weems

Michael J. Scudder

Department of Computer Science
University of Massachusetts at Amherst
Amherst, Massachusetts 01003
NetAd : herbordt@cs.umass.EDU

*This work was supported in part by the Defense Advanced Research Projects Agency under contract DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Laboratory; under contract DAAL02-91-K-0047, monitored by the U.S. Army Harry Diamond Laboratory; and by a CII grant from the National Science Foundation (CDA-8922572). M.C. Herbordt is also supported by an IBM Fellowship.

Abstract: Computation on and among data sets mapped to irregular, non-uniform, aggregates of processing elements (PEs) is an important problem in parallel vision processing, arising in segmentation and in support operations for intermediate-level grouping tasks. The difficulty is that the SIMD processors which map so effectively to pixel-based processing are restricted here in data dependent computations by their limited control mechanisms. Associative processing is an effective means of applying parallel processing to non-uniform computations (Weems 1984), but often results in operating on one data set at a time. We address this problem by introducing an additional level of parallelism we call *multi-associativity* which provides a framework for performing associative computation on these data sets simultaneously.

In this article we present algorithms developed for the coterie network (Weems et al. 1989) to simulate efficiently *within non-uniform aggregates of PEs simultaneously* the associative algorithms typically supported in hardware at the array level. One result is that algorithms requiring a number of operations proportional to the diameters of the regions in a mesh connected topology can be executed in constant or logarithmic time using the coterie network. Other results are: the efficient application of existing parindent 0.0truein associative algorithms (e.g. (Falkoff 1962; Foster 1976)) to arbitrary aggregates of PEs in parallel, and the development of efficient new multi-associative algorithms, among them parallel prefix and convex hull. The multi-associative framework also *extends* the associative paradigm by allowing operations on and among aggregates of PEs themselves, operations not defined when the entity in question is always an entire array. Two consequences are: the support of divide-and-conquer algorithms within aggregates, and communication among aggregates. Numerous multi-associative low-level vision algorithms are presented.

Key Words: low-level vision, non-uniform problems, SIMD processing, associative processing, parallel algorithms, coterie network, reconfigurable arrays.

1 Introduction

According to one popular methodology (Marr 1982; Hanson and Riseman 1987), the task of low-level machine vision is to reduce the enormous amounts of input data to a manageable size, and to present those data in a format or representation that allows their easy manipulation. This reduction process usually consists of collecting, characterizing, and labeling groups

of pixels having some property in common: for example a value in a spectral band, a texture measure, or a gradient magnitude and orientation. When abstracted, these collections of pixels can be represented symbolically as edges, lines, curves, patches, blobs, regions, and their combinations and intersections (Brolio et al. 1989). Certain algorithmic approaches to low-level vision are well established, although their massive computational requirements make them impractically slow. This problem is magnified by the requirement that low-level computation be used not just for pre-processing, but also as a part of a continuous feedback loop as interpretation hypotheses are tested and refined. One consequence has been the design and use of massively parallel processors for low-level vision applications.

SIMD arrays have been widely applied to pixel-based processing because of their large numbers of processing elements (PEs), and because many useful algorithms involve only uniform communication operations. Their success has been limited, however, in data-*dependent* computations (such as those enumerated above), which usually result in non-uniform and irregular communication. (See Figures 1-4 for an example of non-uniform region formation.) One methodology that has been applied to solving problems on multiple non-uniform data sets is associative processing (Foster 1976; Weems 1984). For small numbers of data sets this is very efficient; however, we are often restricted to operating on one data set at a time, and low-level vision processing often involves computing the attributes of many thousands of features (Brolio et al. 1989). We have also had some success with what could best be described as ad hoc methods: the solutions are often efficient (Weems et al. 1991), but are not the result of a uniform technique. Others (Willebeek-LeMair and Reeves 1990; Tilton 1988) have embedded binary trees in meshes to implement broadcast and reduction primitives on non-uniform, contiguous regions, and applied those operations with great success to parallel image segmentation. Their method, although optimal for mesh connected topologies, is still linear with respect to the dimensions of the regions.

We propose an additional level of parallelism we call *multi-associativity* as a framework for performing computation on multiple, arbitrarily shaped aggregates of PEs simultaneously, thereby taking advantage of the proximity inherent in many pixel-mapped computations. The strategy is to emulate efficiently *within aggregates of PEs simultaneously* the associative operations typically supported in hardware at the array level. These include broadcast from controller to array and feedback from array to controller (e.g. count and global-OR of tag bits). Once we have efficient implementations of these primitives (constant

Figure 1: 256×256 8 gray level input image

Figure 2: Segmentation of images often results in a very large number of regions

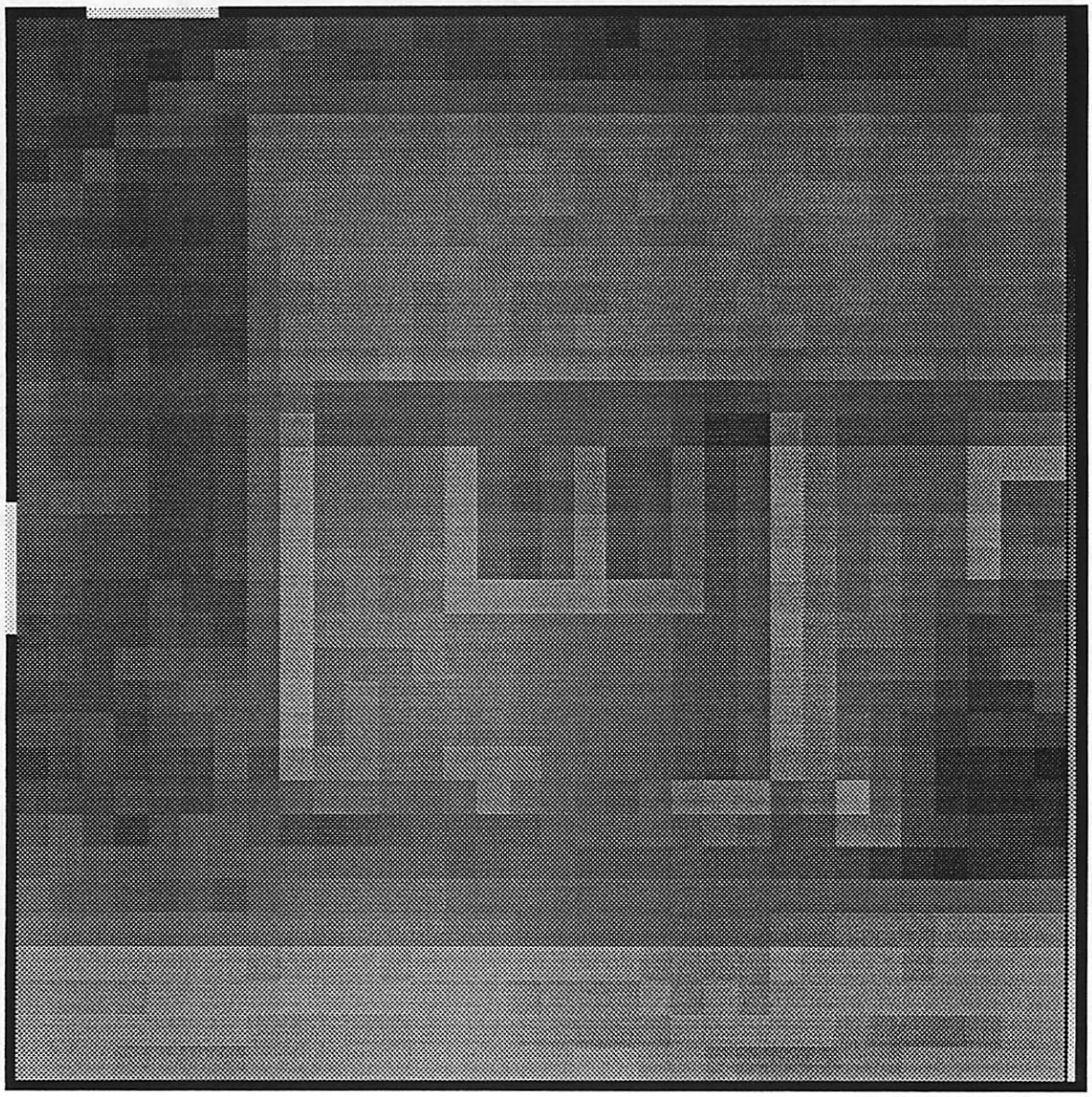

Figure 3: 32×32 8 gray level sub-image of input

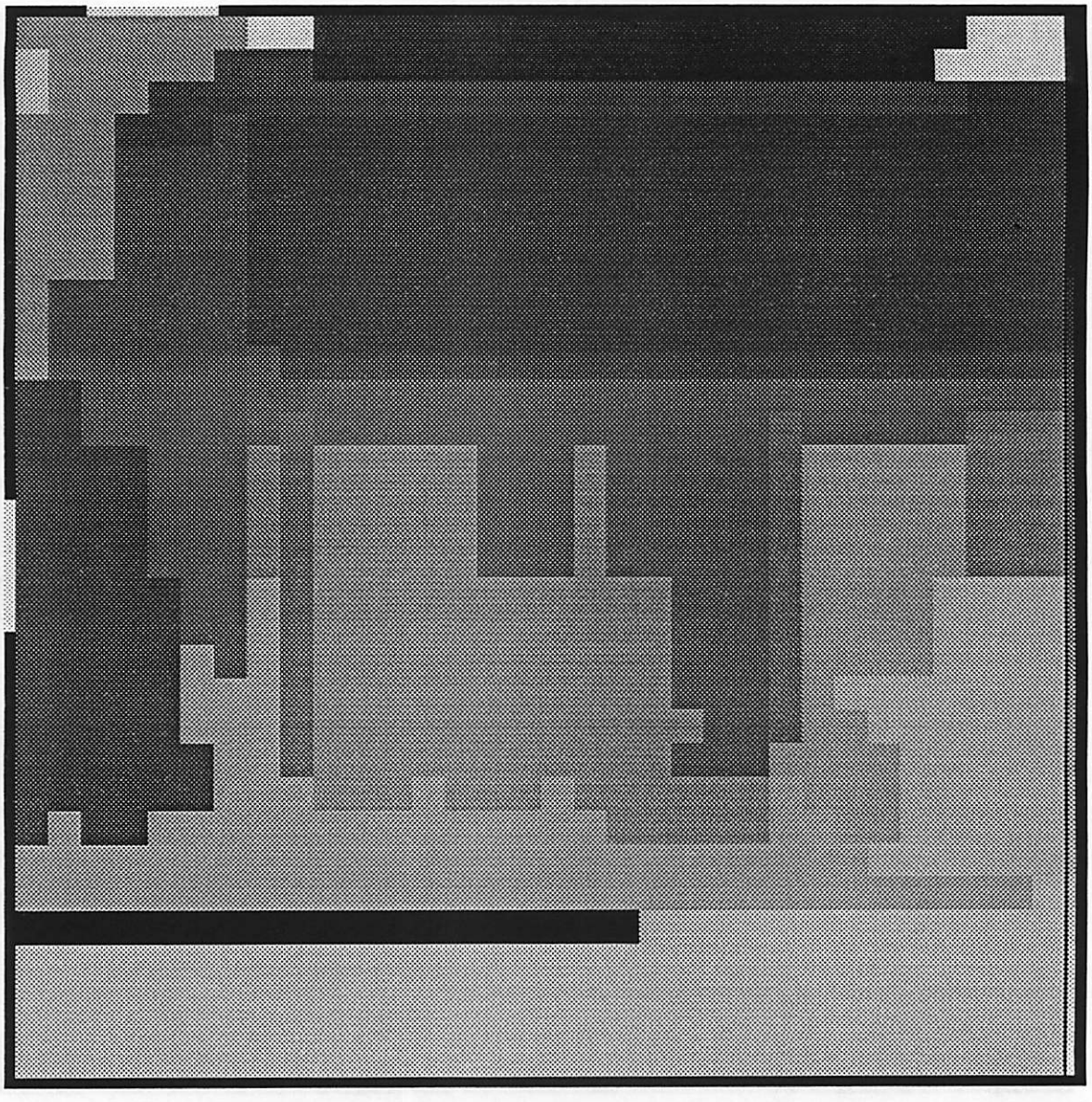

Figure 4: Corresponding 32×32 sub-image of segmentation

or logarithmic complexity), we can apply them to existing associative algorithms and use them to build other complex functions. For this strategy to succeed on a real SIMD array (with only a single thread of control), we must construct multi-associative primitives that have a low branching factor with respect to the shapes of the aggregates. In this paper we present an efficient implementation of multi-associative primitives using the coterie network, the reconfigurable broadcast array used by the Content Addressable Array Parallel Processor (CAAPP) (Weems 1989 et al.), and demonstrate the usefulness of this approach with numerous sample algorithms.

The rest of this paper is organized as follows. Section 2 presents the vision methodology and computational requirements driving our research. In Section 3 we review associative processing and present an extension to multiple aggregates. A description of the target architecture and the coterie network follows in Section 4. Section 5 provides background on the coterie network constructs; these are used in Section 6 where the basic coterie network algorithms are presented. Section 7 contains the mapping of the multi-associative primitives to the coterie network as well as other vision primitives. We conclude with sample vision algorithms in Section 8.

2 A Machine Vision Methodology

Low-level vision processing is often defined in terms of capabilities: we have a massively parallel array with certain communication support, how can we use it? The answer is usually that the pixels are mapped to individual PEs, and the procedures are dominated by certain standard algorithms, such as convolution. In this section we look at low-level vision from the point of view of *requirements*; we still define low-level vision as procedures where pixels are mapped to processors, but also examine what needs arise when computation at the low level is integrated into a complete vision system with top-down as well as bottom-up processing.

Computer vision, the task of deriving descriptions of scenes from their images, is well known to require at least two types of computation: processing of sensory data, and processing of world knowledge. Sensory data processing uses the image array representation and includes line and region segmentation; stereo, motion, and texture analysis; shape from information such as shading, and many other computations. An example of world knowledge processing might be constrained model matching between "stored models in the form of

generalized relational structures (Rosenfeld 1984)," and hypotheses created from the image data. For these two types of processing to be effective, there must be also be constant, bidirectional, interaction between the two levels of representation: for example, as world knowledge is brought to bear on an image, parameters from algorithms processing sensory data must be recomputed.

A problem arising in this computational model is that the representations, pixels on the one hand and generalized relational structures on the other, are incompatible. One solution is to include one or more levels of intermediate representation through which the interaction between high and low levels takes place. These intermediate representations are characterized by a uniform symbolic representation for both high and low level vision events: a low level event, such as a contiguous, colinear, group of pixels of similar gradient orientation, has the same representation as a high level event, such as an edge predicted from a partial model match. In one such system, the Intermediate Symbolic Representation (ISR) (Brolio et al. 1989), collections of contiguous image data with similar properties are stored as named and typed symbolic entities called *tokens*.

We now refine our model of vision computation. It is the task of high level vision to hypothesize objects and collections of objects in the scene, and to search the intermediate levels for evidence of those objects. The task at the intermediate level is to provide a dynamic, reconfigurable data base. Additional data transformations also occur there, as perceptual organization algorithms split, merge, add, and/or delete intermediate-level representations. Low-level vision processing retains the standard pixel mapped functions outlined above, but must also include routines to support the dynamic recomputations demanded by the intermediate levels.

There are several requirements for low-level vision computation in the context of an overall vision system. The first two presented below are from the standard bottom-up view, the rest result from integration into a complete system.

1. Computing the attributes of individual pixels, and of small, fixed, neighborhoods of pixels. These operations include much of previous window based low-level vision work, and are characterized by regular communication patterns between nearby PEs.

2. Grouping sets of pixels that share attributes. The methods here come under the heading of segmentation. In these computations, the resulting sets are arbitrary in shape, although the points are usually contiguous.

3. Characterizing these sets; simply labeling pixels as edge/region points is not by itself sufficient. Extracting events for the intermediate levels requires collecting information about the number of points in the set, the number of points with certain characteristics, the mean and median of the pixel attributes, the spatial dimensions of the set, and many others. Another part of extracting symbolic events is gathering information about neighboring sets.

4. Collecting this information in a small number of PEs for rapid transmission to the intermediate level processor. This may be the front end, a separate parallel processor (as in the IUA), or the same processor using a different representation. In each case, however, the complexity of the interaction is reduced if the number of PEs transmitting information is small.

5. Providing support for symbol manipulation at the intermediate level. Sometimes these grouping processes can be accomplished by simple operations on the event attributes within the intermediate levels. Often, however, new symbolic events must be created at the lowest level and the attributes recomputed. This is especially true if an iterative combining procedure is used.

6. We need to be able to make all of the above computations not just once, but many times. As is well known, images of scenes are underconstrained; a consequence is that the entire image understanding process must be dynamic. Evidence in the original image can be missed in initial processing. Therefore, we must be prepared to recompute using different parameters.

What characterizes these requirements? We must support regular, local, communication, but also the extraction of data from arbitrarily shaped contiguous sets. We must be able collect this information in a small number of PEs. We must be able to operate on these sets as distinct entities, with certain inate operations, such as merge. And finally, the speed at which these characterization, collecting, and merging operations takes place must be comparable to the original computation and grouping steps. In the next section we introduce a programming methodology that supports these requirements.

3 Multi-Associative Algorithms

3.1 Introduction to Associative Processing

In the previous section we grouped computational requirements for low-level vision into six categories: (1) computing attributes of individual pixels, (2) grouping pixels into events, (3) characterizing events, (4) extracting event information, (5) support for merging events, and (6) performing 1-5 repeatedly. Here we present a general programming methodology, called multi-associative processing, that provides support for most of these groups of operations. But we first look at the categories in some more detail to characterize the computations they require.

(6) will be satisfied if (1-5) have all been implemented efficiently. (1) requires only context independent, local, communication and is supported by the CAAPP nearest-neighbor communication network. (2) will be discussed in a later paper; however, many of the algorithms presented will also be applicable to pixel grouping. (3-5) do not fit easily into the standard model of SIMD computation: they are characterized by the need for flexible communication and rapid feedback. However, it is exactly these operations that characterize associative processing (also called content addressable processing).

There are four capabilities that are key for associative computation (Foster 1976):

1. Global Broadcast/Local Compare/Activity Control
2. Some/None Response
3. Count Responders
4. Select a Single Responder

The prototypical associative operation is for the controller to broadcast a query to the array, and to receive a response either in the form of Some/None (global OR of response tag bits) or a Count (of tag bits). But associative processing, as opposed to the familiar associative memory operations, also enables the conditional generation of symbolic tags based on the values of data, and the use of those tags to constrain further processing. These capabilities are useful for low-level machine vision when the controller performs multiple logical operations on pixels, or events having multiple tags, based on their attributes and relationships to other pixels or events. Only subsets of the data are involved in any particular operation, but all

pixels and events with a given set of properties are processed in parallel. See (Weems 1984; Weems et al. 1989) for numerous examples of associative vision algorithms.

Let us examine in more detail the fundamental operations required for associative processing, and the hardware support required to perform them efficiently. Global broadcast, local compare, and activity control are all standard SIMD operations: a controller can broadcast global data as well as instructions, PEs typically possess a local comparitor, and a PE activity-control bit is the standard method to implement branching. Thus, capability (1) is available to most SIMD processors. However, the distinctive requirement of associative processing is that of rapid feedback from array to controller: if the controller must query individual PEs for responses, then all parallelism is lost. Therefore the CAAPP has been designed with specialized hardware for the rapid execution of Some/None, Response Count, and Select Single Responder (Select-First) (Rana and Weems 1990). Typical execution times of these operations are 0.1, 1.6, and 2.4 micro-seconds, respectively. For reference, the execution of a bit-serial arithmetic operation takes between 1 and 6 microseconds.

3.2 Multi-Associative Processing

Associative processing permits operations on any single selected subset of the data. But what if there are many data sets to work on simulaneously? In the model of vision presented earlier, we are computing attributes of thousands of events; a simple associative system would typically perform these computations by time-slicing between sets of pixels. We would prefer to operate on all sets of pixels simultaneously. We propose a new model we call *multi*-associative processing, in which associative operations are applied to multiple data sets simultaneously.

We now define the multi-associative model independently of specific architectures, except that we assume an SIMD array of N PEs; the implementation on the CAAPP will be discussed in the next section. To facilitate the definition, we use set notation. We begin by partitioning the array into $k \leq N$ aggregates $S_i \in \{S_1, \ldots, S_k\}$ of PEs. Next, we define associative capabilities analogous to those presented above. This time however, instead of being defined over the entire array, the operations are *executed simultaneously within each aggregate* S_i. But we still have only one controller for the entire array; how can these operations be meaningful? By replacing the role of the controller in the global broadcast, some/none, count, and select first operations with an arbitrary subset of PEs within each

S_i. For example, "global broadcast" from controller to array is replaced with: "$\forall i$, multicast by selected subset of PEs, $S_i^{MCast} \subseteq S_i$, to a selected subset of receiving PEs, $S_i^{Rec} \subseteq S_i$." In the same way, "count responders" is replaced with: "$\forall i$, send count of subset of selected PEs, $S_i^{Sel} \subseteq S_i$, to $S_i^{Rec} \subseteq S_i$." Whenever $\left|S_i^{MCast}\right| > 1$, the signal multicast to the set S_i^{Rec} is the OR of the values multicast by the individual elements of S_i^{MCast}.

We also define two operations on the aggregates themselves. These are Split/Merge and DataTransfer. Split allows, for all aggregates in parallel, the S_i to be split into any number of new aggregates, depending on some predicate. Merge allows any number of aggregates to be combined into a single aggregate. We now have *six* capabilities defining multi-associative processing:

1. Within Each Aggregate: Multicast by a Subset/Local Compare/Activity Control
2. Within Each Aggregate: Some/None Response to a Subset
3. Within Each Aggregate: Count Responders to a Subset
4. Within Each Aggregate: Select a Single Responder
5. Split/Merge Aggregates
6. Data Transfer Between Aggregates

The first four operations map all associative algorithms defined over processor arrays to associative aggregates in parallel. The Split and Merge operations give the model new power beyond the ability to carry out multiple associative operations in parallel; there are three basic advantages:

1. Split and Merge enable the use of divide-and-conquer strategies. It is now possible, for several important algorithms, to iteratively partition the S_i into subsets, solve the subproblems, and then merge. Just as in the sequential case, we can thereby reduce the computational complexity from linear to logarithmic.
2. We can save different partial results in individual PEs, as required by parallel prefix and some reductions.
3. We can take advantage of implicit ordering, for example, when we wish to extract corner points in order around a border.

We emphasize the major characteristic of the multi-associative model: there is still only one controller. Therefore, only algorithms with a branching factor much lower than the number of associative sets will run efficiently. However, since most low-level vision applications lend themselves to algorithms that meet this criterion, there is also a positive side: there is greater hardware efficiency because only a single controller is needed.

4 The IUA, the CAAPP, and the Coterie Network

4.1 Architecture

In order to build an architecture suitable for machine vision it is apparent that not only must tremendous amounts of data be processed, but that the processing must be of many types: low-level feature extraction, intermediate-level grouping, high-level model matching, and continuous communication between the levels. The philosophy behind the Image Understanding Architecture is that qualitatively different types of computation require qualitatively different types of architectures. A detailed discussion of requirements can be found elsewhere (Weems et al. 1989; Weems 1991); we summarize a few that are relevant here: the ability to process both pixel and symbol data in parallel, the ability to simultaneously maintain the representations and perform computations at the low, intermediate, and high levels, the ability to select particular subsets of data for varying types of processing; and fine-grained, high-speed communication and control among processes at each level, and between the different processing levels.

A three level architecture has been developed with each level having an appropriate architecture for the set of tasks at that level. The lowest level processes and extracts features from arrays of pixel data; these operations are usually pixel-based, requiring little communication outside the neighborhood of individual pixels. The processor at that level (the CAAPP) is a SIMD array of processing elements (PEs). The highest level must support processing by, and information exchange between, diverse course-grained processes (Draper et al. 1989), including manipulation of 2D and 3D object models, as well as the complex control strategies needed to apply those processes. The high-level processor (SPA) will therefore run a LISP-based black-board system (Erman et al. 1980), but the details are not yet fully defined.

The intermediate level must provide several functions. One is using the control and

addressing flexibility of the MIMD processors to assist the CAAPP with feature extraction; dual-ported memory is used here. Another is supporting high-level queries. And finally, the intermediate level must be a processor in its own right. In this last role, the intermediate level processor is charged with grouping and organizing symbolic representations into more complicated structures. The ICAP (as the intermediate level processor is known) is a collection of signal processing chips communicating with each other via a general routing network, and the CAAPP and SPA levels through dual-ported memory.

The Content Addressable Array Parallel Processor (CAAPP) is designed as a 512×512 associative array of one-bit processing elements (PEs). Each processing element has several general purpose registers, 320 bits of on-chip cache memory, 32K bits of main memory, and an "Activity Register" used for branching control. The Array Control Unit (ACU) broadcasts instructions, data addresses, and global data. The controller can also extract information from the array by associative polling, as hardware support is provided for Some/None and Responder-Count operations. Communication between PEs themselves can take place in two different ways: by using the nearest neighbor mesh interconnection network, and via a reconfigurable mesh called the coterie network. The first method is similar to that used by the CLIP-4 (Duff 1978), the MPP (Batcher 1980), and the DAP (Hunt 1981). In the second method, PEs transmit information by writing to a specified register connected to the coterie network. PEs then read a register which will have been set to the OR of these signals, within a local group as defined by the network configuration. This scheme is a generalization of the "flash-through" mode of the ILLIAC III (McCormick 1963), the propagate operation in the CLIP-4 (Duff 1978), and networks proposed by (Miller et al. 1988; Li and Maresca 1989). In order to distiguish broadcast by PEs from the broadcast by the controller, we refer to this operation as "coterie multicast."

In the coterie network, each PE controls a set of eight switches (see Figure 5), enabling the creation of electrically isolated groups of PEs that share a local associative Some/None feedback circuit. Four of these switches control access in the different directions (North, South, East, West). Two switches, H and V, are used to emulate horizontal and vertical buses. The last two switches, NE and NW, are used to create eight-way connected regions. The isolated groups of processors (see Figure 6), called coteries, have access only to the multicast signals of PEs within their own coteries. For example, when a set of PEs multicasts within a coterie, the receiving PEs will read the OR of precisely those PEs multicasting within

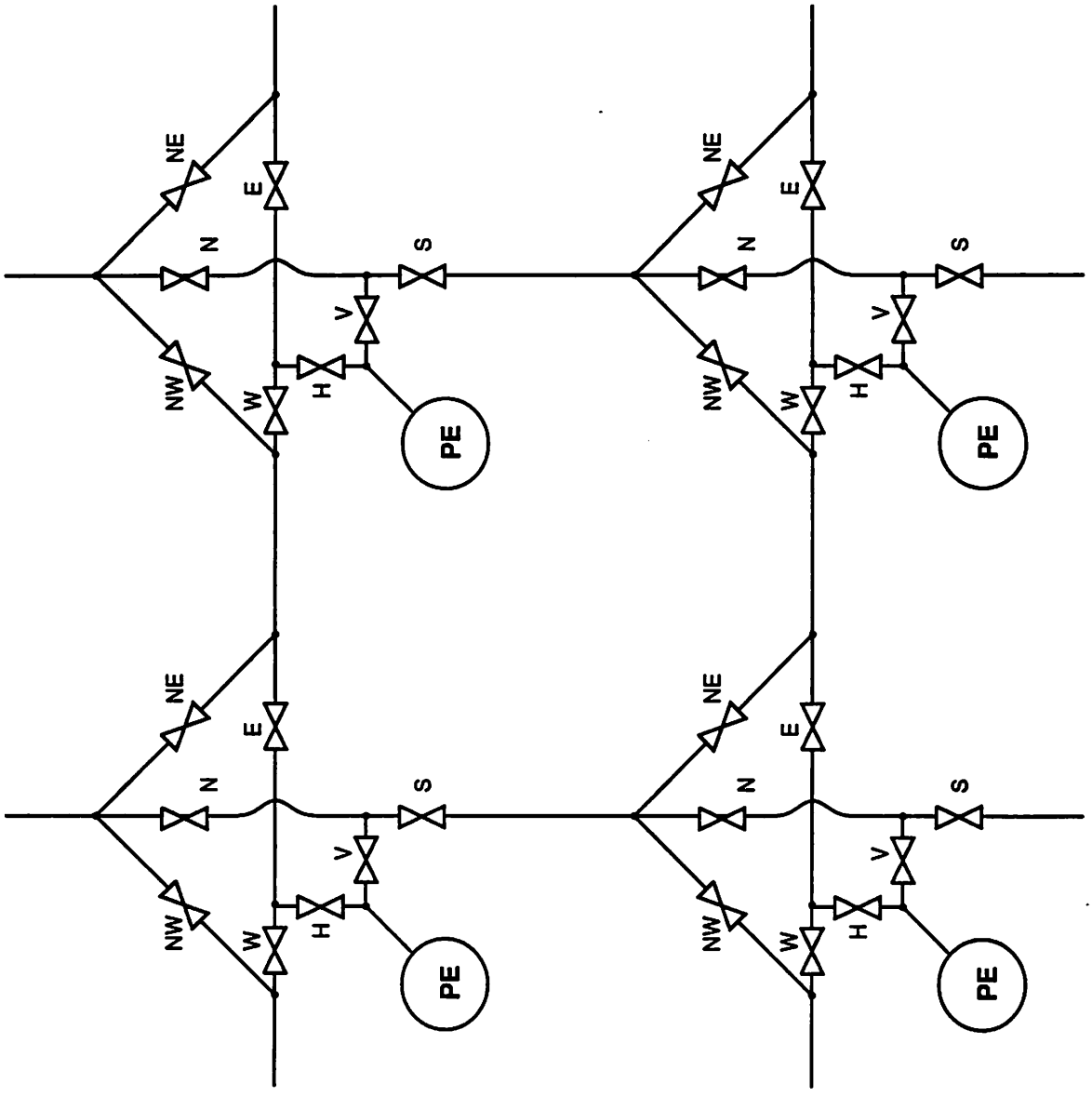

Figure 5: Four PEs and the coterie switches they control

the same coterie.

The coterie network switches are set by loading the corresponding bits of the mesh control register in each PE. Because each PE views the mesh control register as local storage, coterie configurations can be loaded from memory, or can be based on data dependent calculations. In general, the coteries can be any contiguous set of PEs, and this is the way the network is conventionally used to support grouping tasks. However, the coterie network can also be set so that columns and rows are isolated. Once this is done, the row and column "buses" can be arbitrarily segmented still further. The coterie network can thus emulate the mesh with reconfigurable buses (Miller et al. 1988), and the polymorphic-torus (Li and Maresca 1989). In this mode, the coterie network is well suited for many algorithms designed for regular topologies, such as general routing, parallel prefix, emulation of various networks, and many other useful constructs.

4.2 Basic SIMD Operations

In the next two subsections we describe some useful operations available on the CAAPP (and most SIMD processors) and on the coterie network.

LoadPeID. Each PE has an ID, defined (using standard convention) as its address in row and column coordinates. Since the ID is used often, LoadPeID is usually executed once, and the value retained in cache memory.

Select. PEs all contain an activity register, the value of which determines whether that PE will execute the instruction currently being broadcast by the controller. A PE with an activity register set to one is said to be "Selected." Often, PEs are selected according to whether an internal label matches a broadcast or multicast key.

Route and Combine. Route is defined as an operation where any source PE_i^S can send data to any destination PE_j^D. If multiple PE_i^S's send packets to the same destination PE_j^D, then those packets can be combined according to some operator. For example, in SumCombine, the result in a PE_j^D is the sum of the contents of all the packets sent there. Other combine operations include MaxCombine and combine with boolean operators such as OrCombine. On the CAAPP, route and combine are implemented as software functions; more details are given in (Herbordt et al. 1990). The fundamental result is that most communications patterns can be routed on-line—on a SIMD processor such as the CAAPP with no support for indirect addressing or queues—with a number of local communications

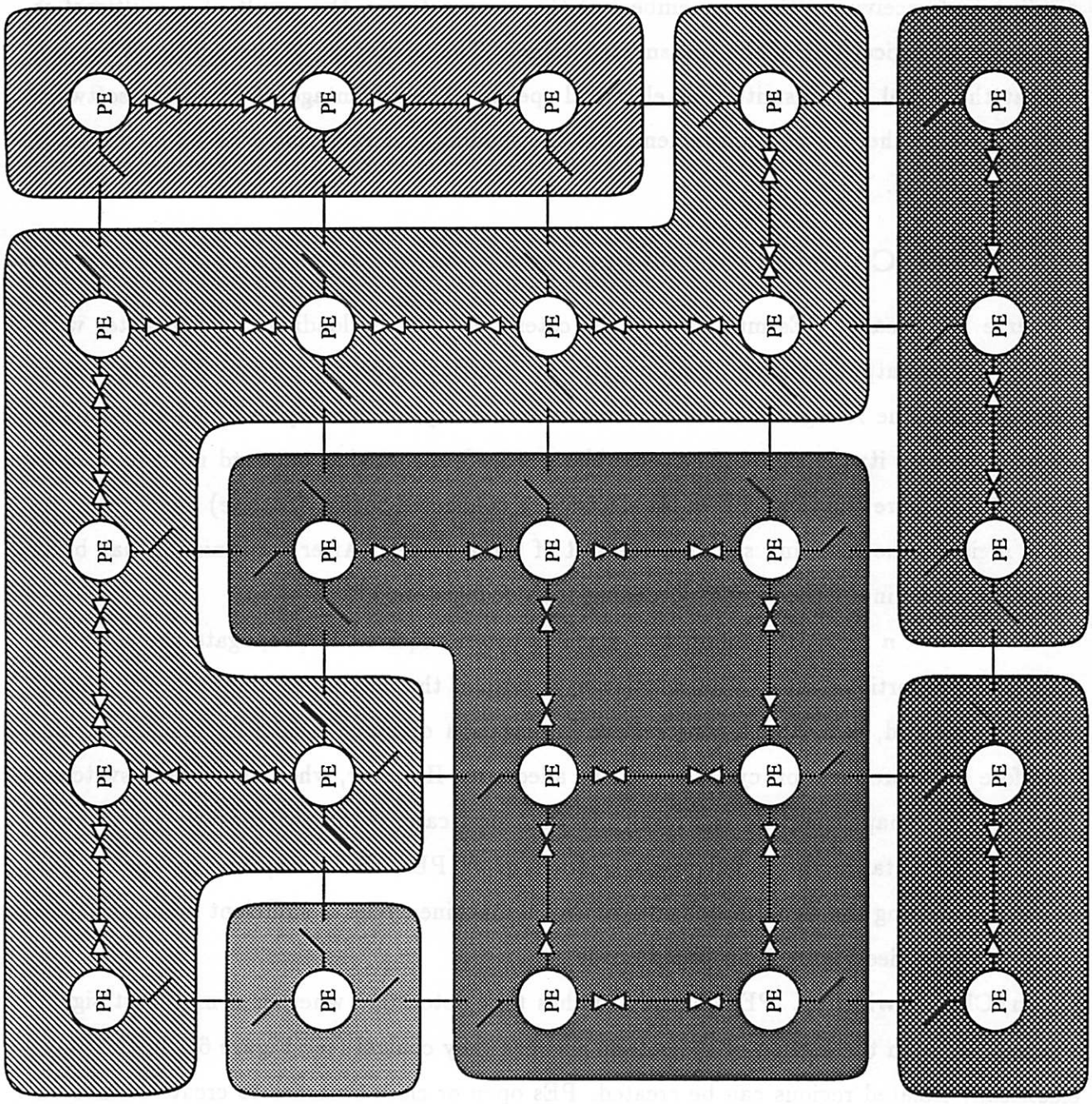

Figure 6: A small sub-array of PEs with simplified switch configuration. Coteries are shaded.

operations only slightly larger than the maximum distance any packet must travel.

OrCombine is also implemented using coterie multicast. As long as the distinct sets of sending and receiving PEs are members of the same coteries, the result of a multicast is precisely the logical OR of the transmitted values. The advantage of the multicast version is that the signal is transmitted at electrical speeds. The advantage of using the software version is that there is no requirement that the sending and receiving PEs be members of the same coterie.

4.3 Basic Coterie Operations

Coterie Multicast. PEs multicast on the coterie network by loading the X register with the bit to be output, whence it propagates at electrical speeds for some distance across the network. The X register value is retransmitted every machine cycle by all PEs having already received it; the signal thus resembles a wavefront, moving outward until it reaches every PE. If more than one PE in an electrically connected region (coterie) has written to its X register, the resulting signal is the OR of those values. After transmission has been completed, PEs input the signal by reading their X registers.

For an $n \times n$ array, the number of machine cycles required to propagate the signal to all PEs is proportional to n. The calculation is simple: the maximum Manhattan distance, with wraparound, is n, and a conservative propagation distance is 50 PEs/machine cycle; therefore $n/50$ propagation cycles should be adequate. However, when the coterie switches are set, oddly shaped regions can result. In the worst case, if the switches are set to form a spiral, the distance the signal needs to travel is N PEs. However, our experience has shown that letting the signal propagate for $2n/50$ machine cycles is sufficient in all practical circumstances. See Figure 7 for pseudo-code.

Open/Close Switches. PEs control switches that determine whether a multicast signal will pass through the section of the coterie network they control (see Figure 6). In this way, electrically isolated regions can be created. PEs open or close switches to create connected components, to isolate row or column buses, to separate region borders, or because of the parity of a bit in an address, ID, or offset as specified by some algorithm. PEs can only multicast information to other PEs that are members of the same coterie, and all coteries are made up of contiguous sets of PEs.

```
FOR BitNum := 0 TO LabelLength - 1 DO      {For all bits in the label}
    Activity := WriteCondition             {Activate PEs matching the "write" condition}
    Response := Label[BitNum]              {Transmit data}
    Propagate()                            {send data across circuit}
    Activity := ReadCondition              {Activate PEs matching the "read" condition}
    Output[BitNum] := Response             {Input transmitted data}
```

Figure 7: Procedure for coterie multicast. Parameters are Label, LabelLength, WriteCondition, Ouput, and ReadCondition.

5 Using the Coterie Network

Physically, a coterie is a bus with broadcast. More abstractly it is an arbitrarily shaped, connected subgraph of a mesh. Another view is that the coterie network is a mesh with reconfigurable buses, but without the buses being restricted to one dimension. Some of the advantages of the coterie network are support of one-to-many communication (sometimes many-to-many) within coteries, the reconfigurability of the coteries, and the propagation of information over long distances at electrical speeds. The disadvantages are that a circuit can carry only one data element per communication step, and that the creation of possible partitions for multiple communications are restricted by the underlying geometry of the network.

The fundamental strategy in using the coterie network is to orchestrate the partitioning of the array (or some previously determined subsets) into coteries such that the maximum number of PEs needing to exchange information on any algorithm-step can do so. Since we are concerned primarily with data dependent algorithms, the methodology requires each PE to determine in a small constant number of time-steps what role it is to play in the next phase or operation of the algorithm: whether it is a sender, receiver, or off; as well as how the section of the network controlled by that PE should be configured. The key is using the information available locally to each PE, for example:

- the row and column ID,
- the tag and the tags of the nearest neighbors, and
- the OR of some bit of information in a given memory location of members of the coterie.

In the next sections we will show how this information, together with iterative partitioning, can be used to construct efficient algorithms, but first present some definitions.

5.1 Local PE Coterie Definitions

In coterie algorithms it is often convenient to classify a PE with respect to its position in the coterie. The most basic information is whether the PE is an interior or perimeter point, that is, whether any of its (either four or eight) nearest neighbors has a tag different from its own. Another classification is according to how the switches (N,E,S,W for the purposes of this paper) are set. Depending on the number of connections to the coterie, the PE will have different capabilities in controlling the flow of information through the network. Since algorithms often involve multiple partitions, a PE may have different switch settings during different algorithm steps. Depending on the number of connections to its nearest neighbors, we call a PE *unconnected* (0 connections), an *endpoint* (1), a *through-point* (2), a *T-junction* (3), or a *cross* (4).

5.2 Coterie Stuctures

Certain coterie algorithms are available only on some of of the possible array partitions. Therefore, an obvious strategy is to partition a coterie that does not meet the algorithmic specification into a number of coteries that do, and then to combine the results. We next look at classes of coteries: Examples of structures described have been created from Figure 4 and can be found in Figures 8-13.

5.2.1 Coterie

Definition

This is the general structure: A coterie is any aggregate of processing elements sharing the same circuit. Figure 8 contains coteries derived from the segmented image fragment of Figure 4.

Method of Construction

Coteries can be created by loading masks, or by having PEs close switches in the directions of neighboring PEs with whom they share an attribute.

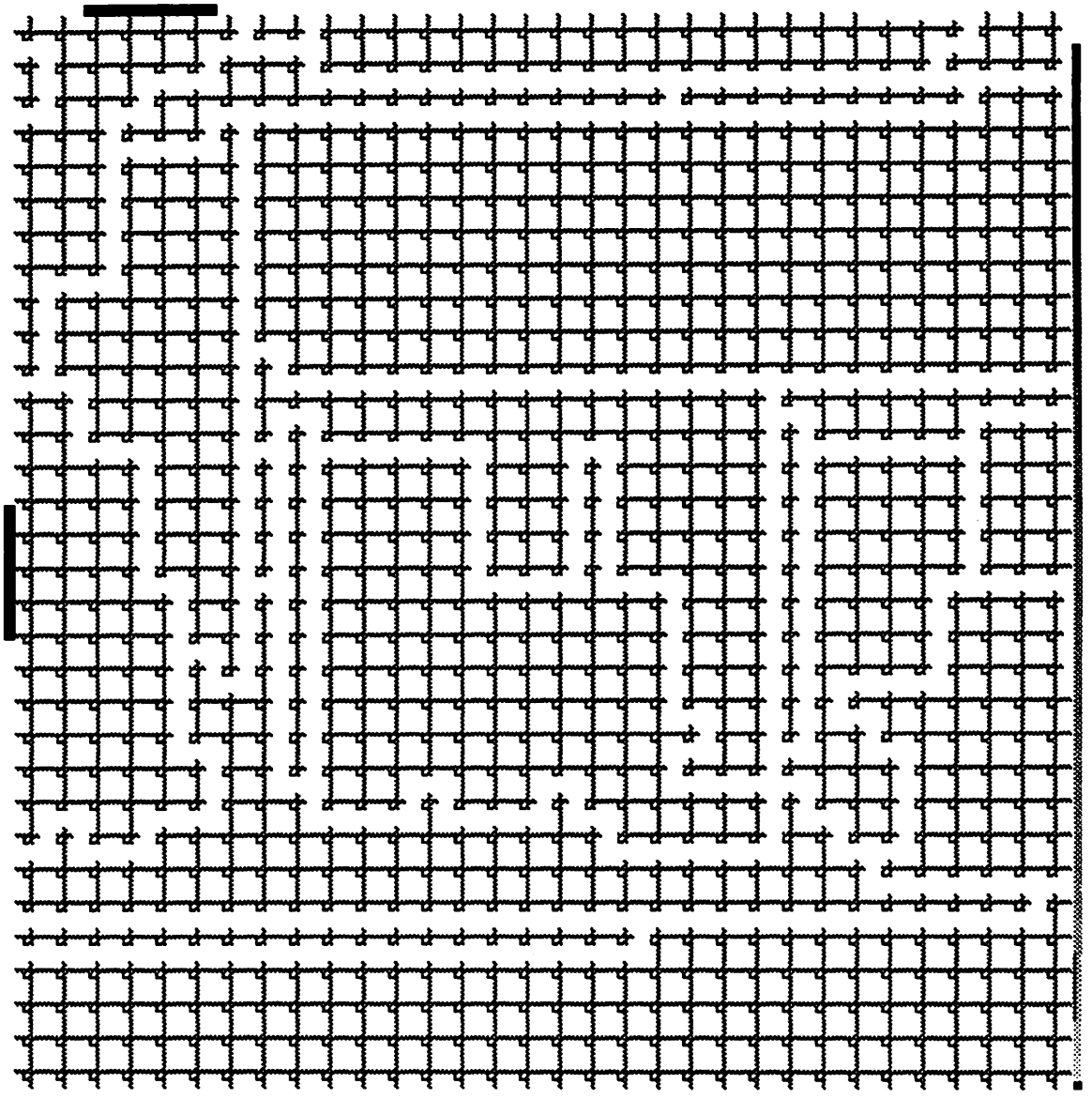

Figure 8: Coterie structures derived from Figure 4—Coteries corresponding to regions

5.2.2 Vertical (Horizontal) Line

Definition

Lines are coteries where all member PEs have the same row or column ID. Figure 9 contains horizontal lines derived from the coteries in Figure 8.

Method of Construction

In order to partition a coterie into vertical (horizontal) lines, open the E and W (N and S) switches.

5.2.3 Boundary Contour

Definition

The obvious definition of a boundary contour of a coterie is the set of exterior points of that coterie together with all their mutual connections. However, this definition can result in boundaries that are not connected or—in the case of two-pixel wide regions—redundantly connected. As it is often useful to traverse the boundary, we modify the definition slightly. The boundary contour of a coterie has as members the set of *eight*-connected exterior PEs in that coterie. Not all connections among those PEs are closed, however: In order for two PEs to have mutually closed switches, they must be mutually adjacent to at least one PE from outside the region. Figure 10 contains boundary contours derived from the coteries in Figure 8.

Method of Construction

Each PE creates bit vectors whose eight values are determined by whether the eight-connected neighbor in a given direction is an interior, exterior, or external PE. The switches are then set according to the above definition.

5.2.4 Singly Vertically (Horizontally) Connected Contour

Definition

SVCCs are defined as coteries composed of horizontal lines with exactly one vertical connec-

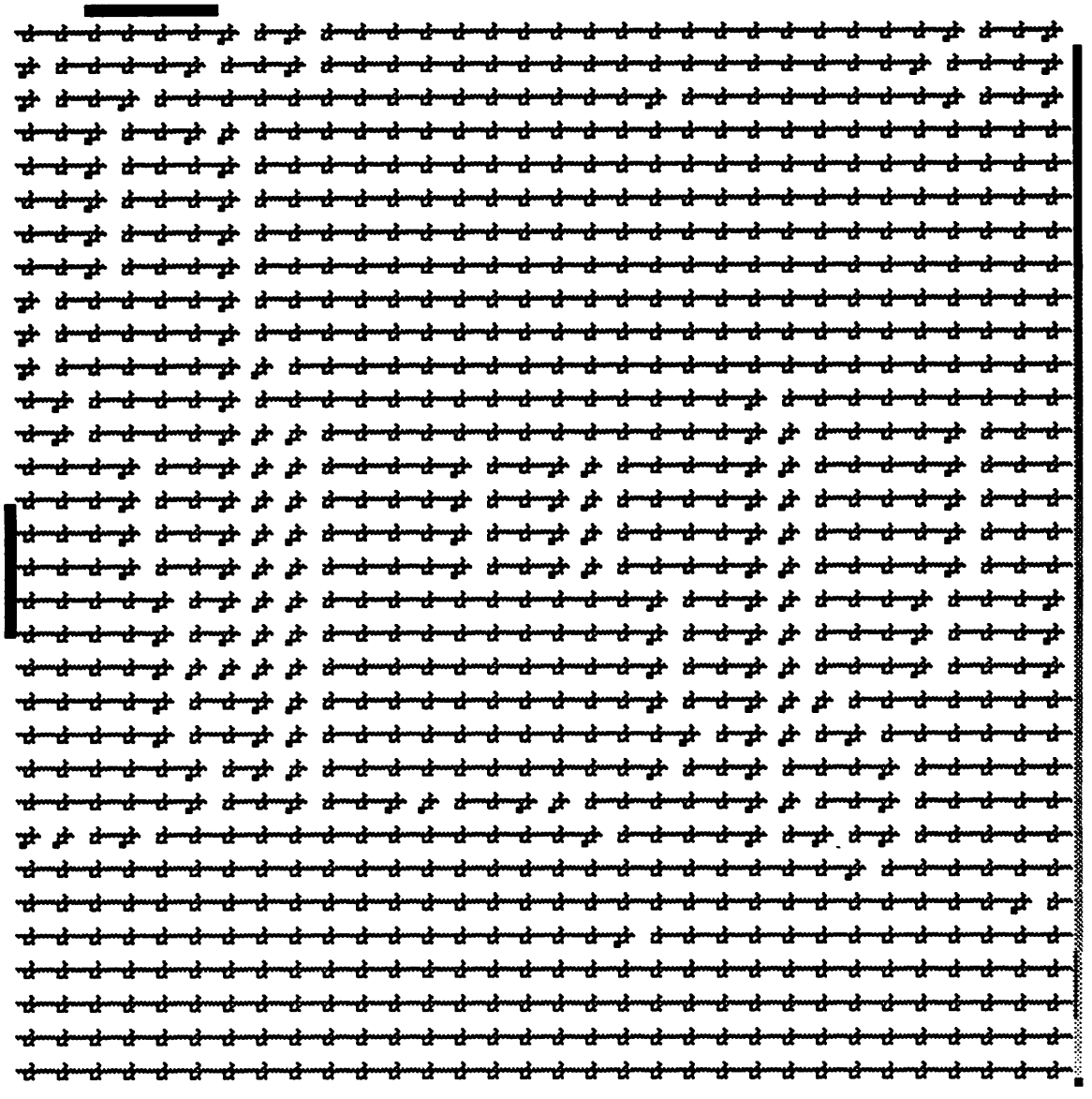

Figure 9: Coterie structures derived from Figure 4—Lines created when North and South switches are opened

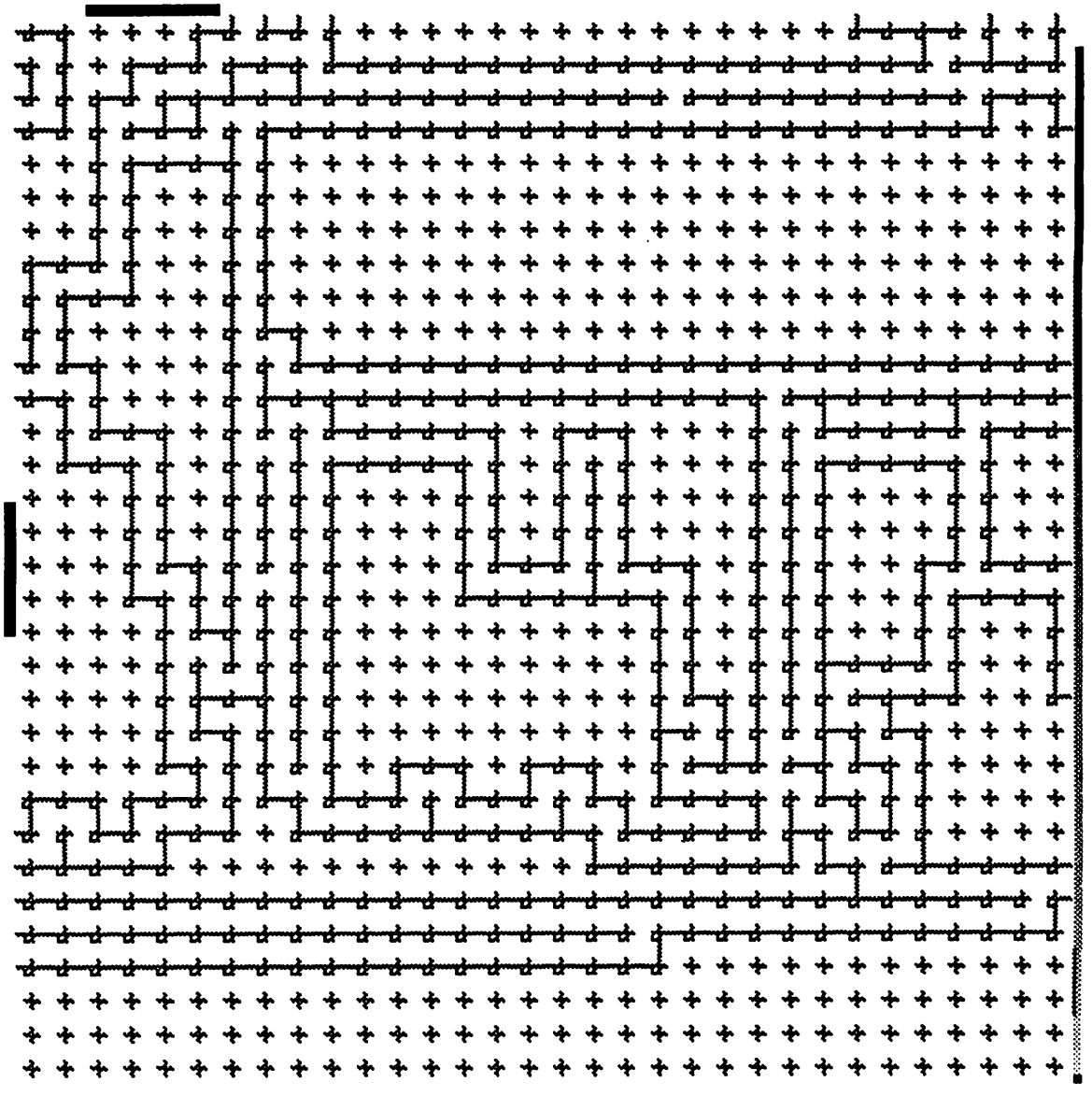

Figure 10: Coterie structures derived from Figure 4—Boundary Contours

tion between each pair. A PE in a given row whose N (S) switch is closed is called an up-link (down-link). Figure 12 contains SVCCs derived from Figure 8; Figure 10 and Figure 11 are intermediate structures in the construction. SHCCs are analogous.

Method of Construction

Two conditions derived from how SVCCs are used constrain their construction: (1) SVCCs are subsets of boundary contours, and (2) it can be assumed that data of interest in the computation has already been reduced to one value per row, which by convention is deposited in the rightmost PE. Even with these constraints, the problem of partitioning an arbitrary boundary contour into the *minimum* possible number of SVCCs using only a small constant number of steps is surprisingly complex. Although we have developed such an algorithm, it is quite detailed and mostly of theoretical interest. We present here a procedure for creating SVCCs from boundary contours that in practice often results in the minimum number, and if there are no embedded regions, always returns a number within a constant factor. Otherwise, the excess is proportional to the number of embedded regions.

We first summarize the algorithms. Because of the possibility of embedded regions, each horizontal line in a boundary contour can have any number vertical connections (up- and down-links) to a neighboring horizontal line. The idea in the general algorithm is find the minimal matching between up- and down-links using only information that can be obtained by each PE in a constant number of steps. In the simplified algorithm, we only make sure that the right endpoint, and the rightmost up-link and down-link are in the same coterie. The other up- and down-links we disconnect.

1. PEs that are right endpoints identify themselves as such.
2. Run SeparateBoundaryContours (see Figure 10).
3. PEs that are up-links and down-links identify themselves as such.
4. Run SeparateHorizontalLines.
5. Form multiply connected vertical contours. Right endpoints multicast a "one." PEs that are not on a line with a right endpoint open their connections (see Figure 11).
6. Identify rightmost up-link. Up-links open East coterie switch, right endpoints multicast a "one." Only the rightmost up-link will receive that one.

7. Identify rightmost down-link. Procedure analogous to step 6.

8. The rightmost up- and down-links remain as they were. The rest are opened (see Figure 12).

5.2.5 Simple Contour

Definition

Simple Contours are boundary contours where each PE is a through-point. See Figure 13.

Method of Construction

The way to create a simple contour from a boundary contour is to cut the "hairs." For each direction (N,E,S,W) in succession, the T-junctions and crosses open all their switches but one. The endpoints multicast a "one". T-junctions and crosses receiving the "one" know that the link in the "closed" direction is towards a hair, and so open that switch. Some T-junctions and crosses will remain intact as the result of the coterie having holes, forming a figure eight, etc. In this case, TraceBorder (see below) must be run to determine which additional switches should be opened.

6 Coterie Algorithms

We present algorithms and specify the coterie structure for which they are applicable. Our strategy is to use information that can be obtained locally in constant time. In the previous section we showed how coteries can be partitioned using only tag information. In this section we construct algorithms by using the additional information provided by the row and column IDs and by the data multicast within coteries.

For the rest of this paper all algorithms will be assumed to be operating over any number of coteries in parallel; for simplicity, the descriptions refer to only one. The execution time is never related to the number of coteries and is either independent of their size and shape, or has a complexity equal to the execution time on the coterie taking the longest to complete. In the latter case, we use global feedback from array to controller to determine when the algorithm has completed for all coteries.

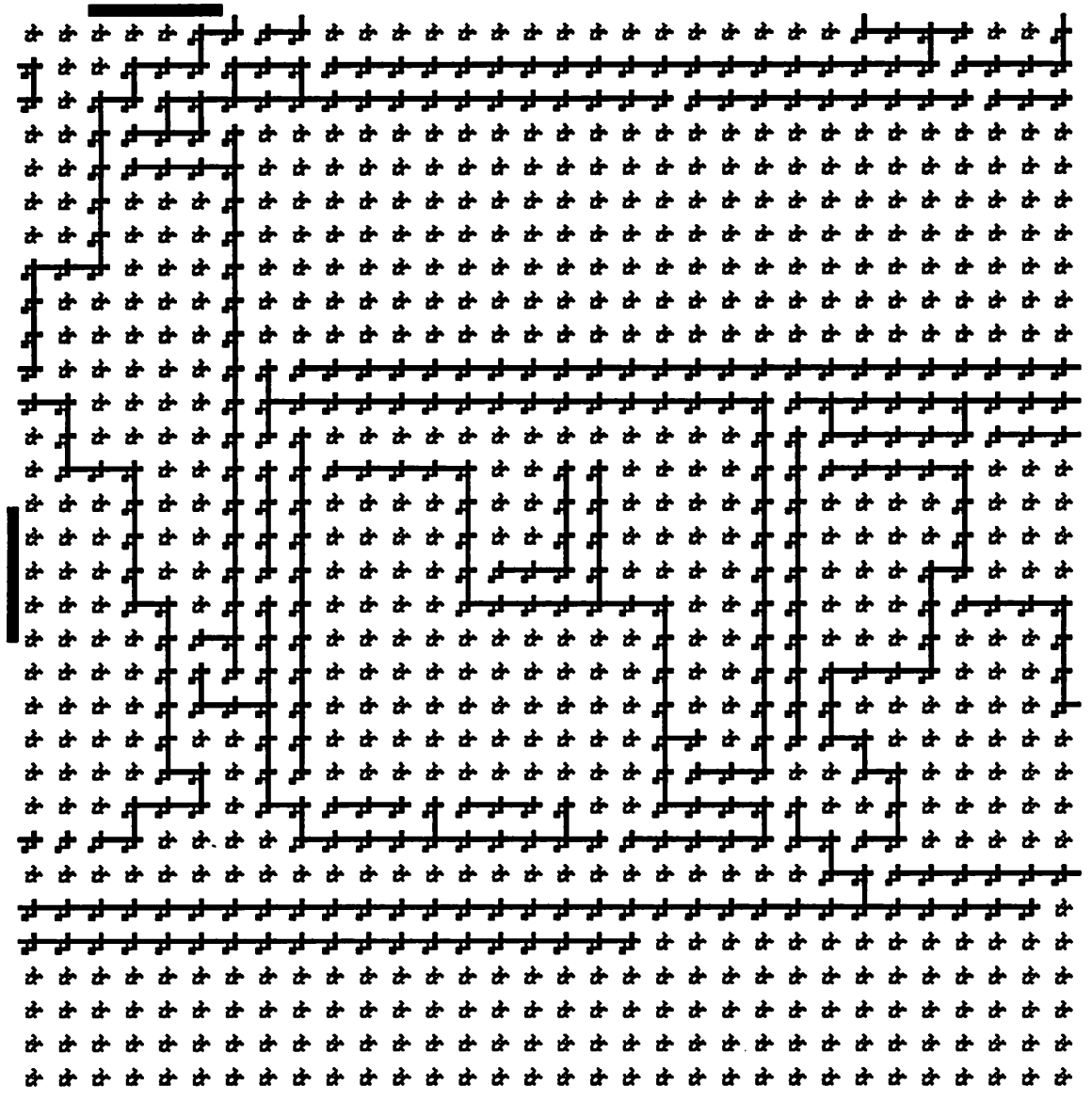

Figure 11: Coterie structures derived from Figure 4—Multiply Vertically Connected Contours

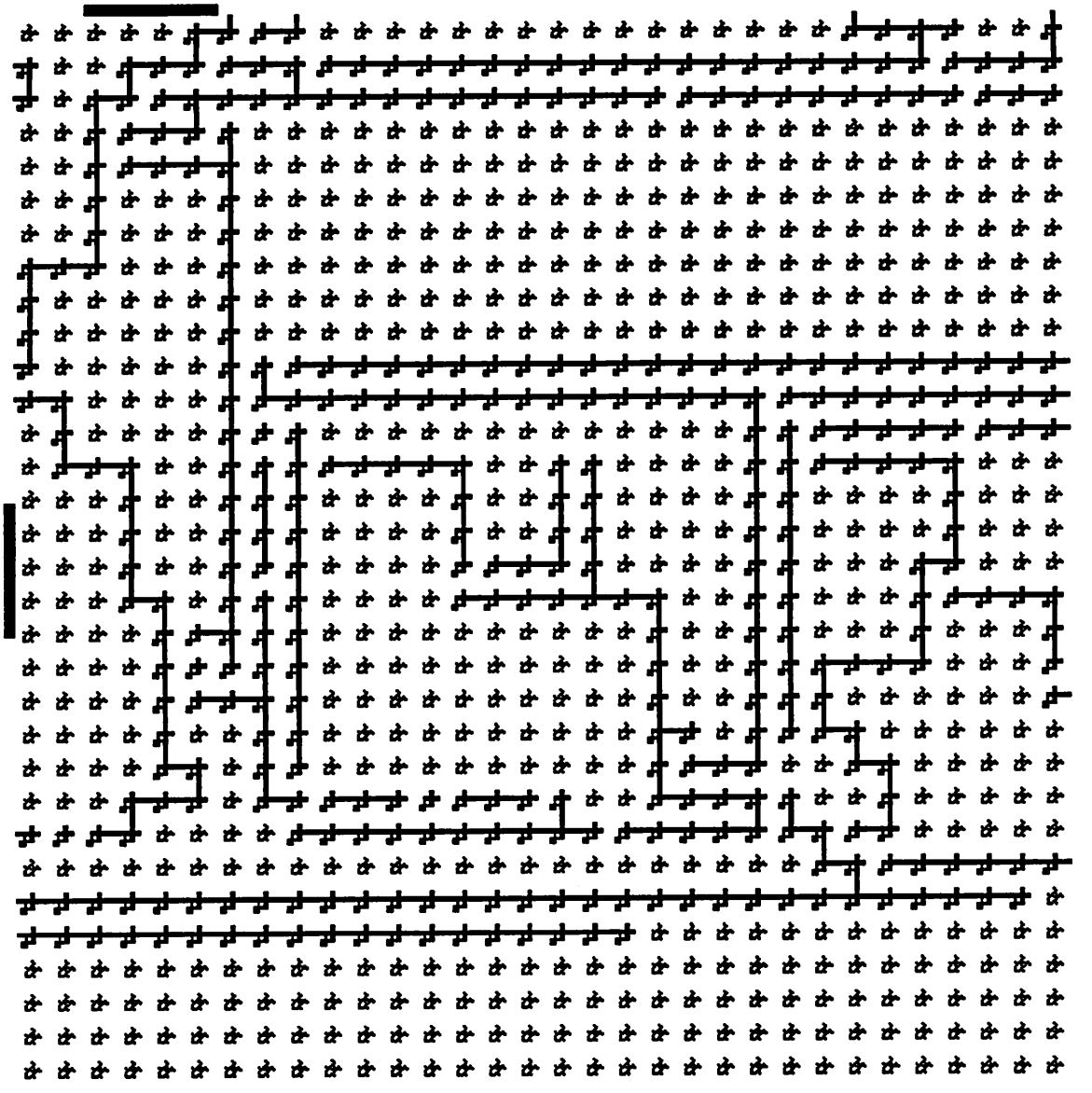

Figure 12: Coterie structures derived from Figure 4—Singly Vertically Connected Contours

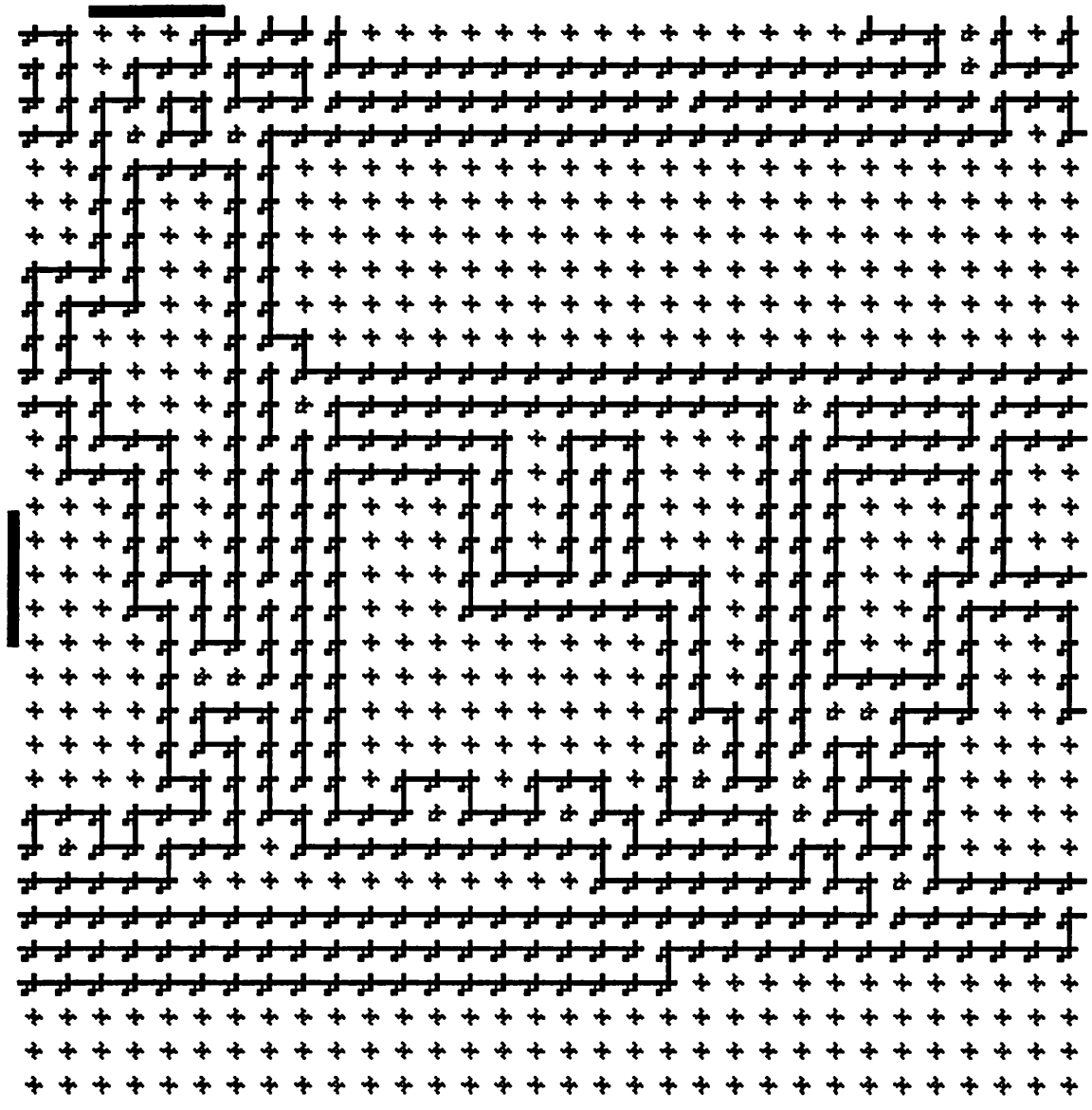

Figure 13: Coterie structures derived from Figure 4—Simple Contours

6.1 Tools

Select Max/Min. The goal is to select the PEs with the greatest value within each coterie. The method is to apply a standard associative algorithm (Falkoff 1962) to coteries. At no extra cost, all the PEs in the coterie "listen in" on the process, and so know the greatest value itself at the end of the algorithm. The algorithm is bit-serial; for k-bit integers the algorithm starts with the high order bit $(k-1)$. PEs multicast bit $k-i$: if any PEs multicast a one, then all PEs with a zero turn themselves off as they have been eliminated. If there are no ones, or if they are all ones, then no PEs are eliminated on that step. By the time the low order bit is reached, only the PEs with the maximum value remain. An analogous algorithm selects the PEs with the minimum value. SelectMax and SelectMin run in $3k$ steps, where k is the number of bits in the word. See Figure 14 for pseudo-code.

```
FOR BitNum := AddressLength - 1 DOWN TO 0 DO   {Beginning with the high-order bit}
    Response := Address[BitNum]                {transmit bit through Response register}
    IF (Response = Some)                       {If any PE has a 1 in this bit,}
        THEN Activity := Response              { turn off activity in PEs with a 0 in this bit}
    ComponentLabel[BitNum] := Response         {Save bit values for component label}
```

Figure 14: Algorithm to find and distribute the maximum valued label in a region.

Select Single Responder. This routine selects a unique PE within each coterie by running SelectMax on the IDs of the responding PEs.

Collect Sparse Data. Assume that desired information is distributed in a subset S of PEs, and that this information is to be gathered in a distinct PE. Run SelectSingleResponder on S to activate one PE in the set. That PE multicasts its data to the accumulator, then removes itself from S. Repeat until S is empty. This procedure requires time linear in the number of PEs holding data, and so is only efficient if that number is small. For dense subsets, either Combine or Reduce (see next subsection) are used.

Collect Ordered Data. This technique works for lines and simple curves. An endpoint (or the PE with the maximum ID in the case of a closed curve) is selected to be the accumulator. A rough outline of the technique is that each PE to determines the links in the directions toward and away from the accumulator, and for the data to be transmitted in

priority according to the distance away from the accumulator. Again, this procedure is only used if the number of PEs holding data is small.

Get Offset (Establish an Ordering Within a Line or SVCC). In any coterie, but of primary importance for lines and SVCCs, a PE can easily obtain its offset from an endpoint. For example, to find the offset from the west end of the coterie, we first select a PE furthest in that direction, either by running SelectMin on the column ID, or for a horizontal line, by selecting the west endpoint. The selected PE multicasts its column ID to the rest of the PEs in the coterie, which obtain their offsets by subtracting from their own column ID.

6.2 Parallel Prefix and Reduction

Parallel Prefix and Reduction are fundamental operations on lists and arrays. Among the applications that use parallel prefix are: the enumeration of selected PEs, Radix Sort, parsing regular languages (Hillis and Steele 1986), and carry propagation in multi-gauge emulation (Annexstein et al. 1990). See (Karp and Ramachandran 1988; Blelloch 1989) for many more. First we define parallel prefix: Given a set $[x_1, \ldots, x_n]$ of n elements with each element assigned to a different processor, and a binary associative operator $*$, compute the n S_i's:

$$S_i = x_1 * x_2 * \ldots * x_i,$$

leaving the ith prefix sum in the ith processor. The operation $*$ is not necessarily commutative. The implementation of parallel prefix takes particular advantage of the coterie network: it requires only $\log n$ communication steps, rather than the $2 \log n$ required for a tree-connected parallel processor.

6.2.1 Parallel Prefix for Lines and SVCCs

We first present parallel prefix for horizontal lines (see Figure 15 for pseudo-code); no assumptions are made as to the lengths of the lines or starting points. An analogous procedure is used for vertical lines. To execute parallel prefix, there must be at least an implicit ordering to the PEs, in this case we use the distance from the west endpoint. Therefore, the first step of the algorithm is for PEs to calculate their offsets using GetOffset.

The rest of this algorithm is illustrated in Figure 16. The binary numbers represent the

```
{Assume Coterie[N,S,NE,NW] = Open, Coterie[H,V] = Closed,}
{and Coterie[E,W] set according to the region boundaries}
SaveCoterieSettings()
{All PEs get offset from east end of horizontal line}
Broadcast(ColumnNumber,ColumnNumberLength,Coterie[E] = Open,Base,All)
Offset := ColumnNumber - Base
{Initialize switch masks. All compare operations using OpenLeftMask}
{and OpenRightMask use only the rightmost BitNum + 1 bits}
OpenWestMask := 0
OpenEastMask := 1
FOR BitNum := 0 TO ColumnNumberLength - 1 DO
    IF (OpenEastMask = Offset) Coterie[E] := Open
    IF (OpenWestMask = Offset) Coterie[W] := Open
    Multicast(Data,DataLength,OpenWestMask = Offset ,Temp,ALL)
    IF (Offset > OpenEastMask AND Offset ≤ OpenWestMask)
        Combine(Temp,Data)              {combine into Data according to a specified operator}
    BitSet(BitNum+1,OpenEastMask)
    BitSet(BitNum+1,OpenWestMask)
    RestoreCoterieSettings()
```

Figure 15: Algorithm for parallel prefix on a horizontal line.

offset of the PE, the decimal numbers the data currently residing there. In the first iteration, PEs whose rightmost offset bit is a 0 open their E switches, the rest open W. The "0" PEs multicast their data, the "1" PEs receive it and "sum." In the next iteration, PEs whose rightmost *two* offset bits are $< 01_2$ do not participate. All PEs with a bit pattern ending in 01 open to the left, all PEs ending with 11 open to the right. The 01's multicast, the others receive and combine. In the next iteration (the final one of the example), PEs whose rightmost *three* offset bits are $< 11_2$ do not participate. All PEs with a bit pattern ending in 011 open their W switches, PEs with 111 open to the right. The 011 PEs multicast, the rest receive and combine.

ParallelPrefixSVCC is surprisingly similar to ParallelPrefixLine. Recall from the definition of an SVCC that all rows of PEs except the top and bottom have exactly one PE with an up-link and one PE with a down-link. Assume that only one PE per row has information to be summed. (If there were more, they could be summed first using ParallelPrefixLine.) There are therefore three PEs of interest on every level, the up-link and down-link PEs, and the information-holding PE (see Figure 17). We now execute an algorithm nearly identical to ParallelPrefixLine, the only difference being that during every iteration, the PEs that send

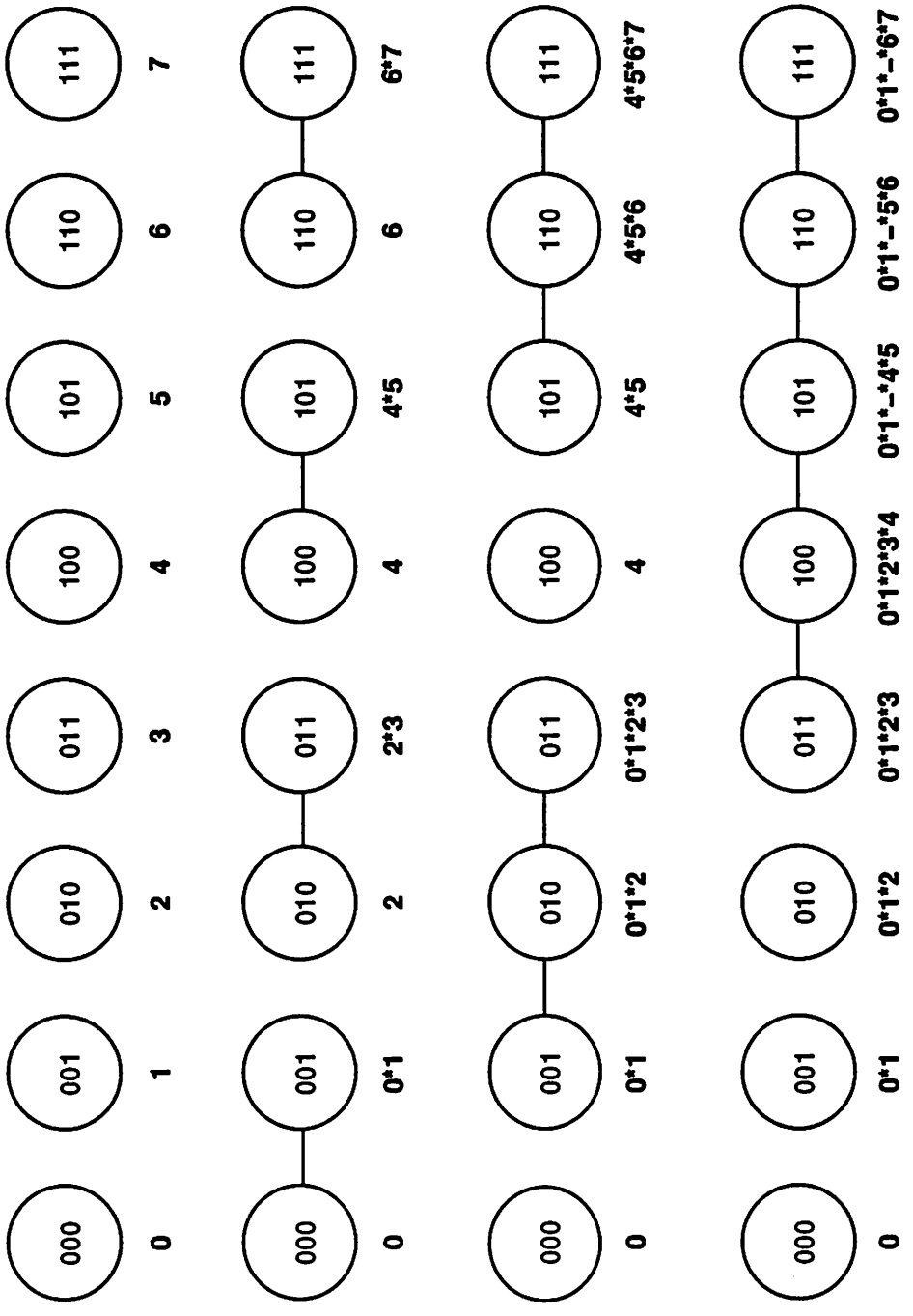

Figure 16: Illustration of parallel prefix on a line: Values and settings for three iterations.

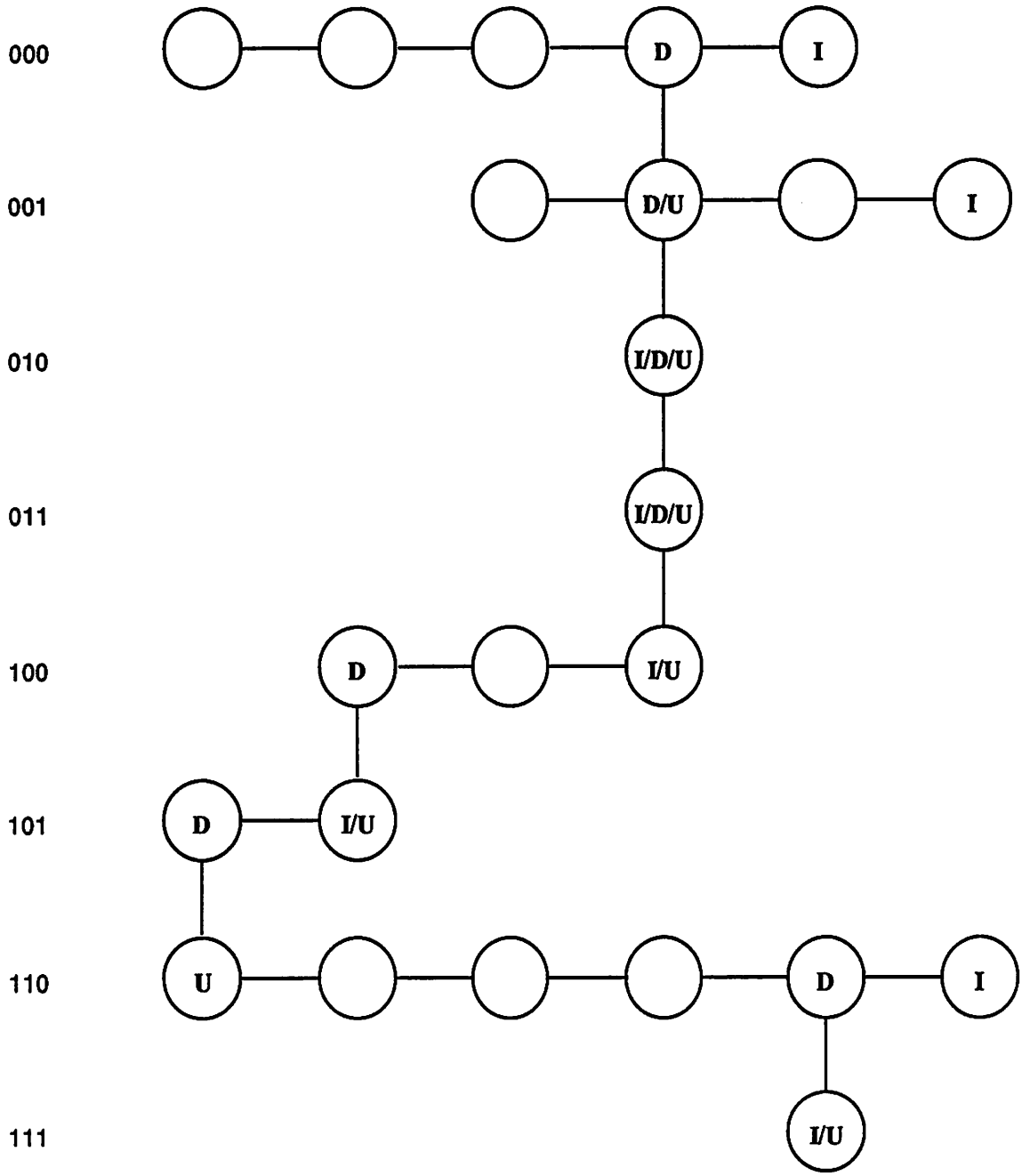

Figure 17: Illustration of an SVCC with information-carrying, up- and down-link PEs indicated

and receive data can be different from the PEs opening and closing the links. See Figure 18 for pseudo-code.

```
{Assume coterie switches set so that coteries are singly,}
{vertically connected contours. Coterie[H,V] = closed,}
{Coterie[NW,NE] = Open}
SaveCoterieSettings()
{All PEs get offset from north end of vertical line}
Broadcast(RowNumber,RowNumberLength,Coterie[N] = Open,Base,All)
Offset := RowNumber - Base
{Identify significant PEs}
If (Coterie[N] = Closed) Uplink := True
If (Coterie[S] = Closed) Downlink := True
If (Coterie[E] = Open) InfoPE := True
{Initialize switch masks. All compare operations using OpenLeftMask}
{and OpenRightMask use only the rightmost BitNum + 1 bits}
OpenNorthMask := 0
OpenSouthMask := 1
FOR BitNum := 0 TO ColumnNumberLength - 1 DO
    IF (OpenSouthMask = Offset AND Downlink) Coterie[S] := Open
    IF (OpenNorthMask = Offset AND Uplink) Coterie[N] := Open
    Multicast(Data,DataLength,OpenNorthMask = Offset AND InfoPE, Temp,ALL)
    IF (Offset > OpenSouthMask AND Offset ≤ OpenNorthMask AND InfoPE)
        Combine(Temp,Data)              {combine into Data according to a specified operator}
    BitSet(BitNum+1,OpenSouthMask)
    BitSet(BitNum+1,OpenNorthMask)
    RestoreCoterieSettings()
```

Figure 18: Algorithm for parallel prefix on an SVCC.

6.2.2 ParallelPrefix for Coteries

To illustrate the technique for parallel prefix on arbitrary coteries, we first sketch a parallel prefix algorithm for rectangles. The algorithm runs in two phases: the first starts by separating the coterie into horizontal lines with SeparateHorizontalLines. ParallelPrefixLine is then executed on these lines. In the second phase, the right endpoints are made into a coterie with SeparateVerticalLines. ParallelPrefixLine is now run on the vertical line using the results from the first phase as input. The second phase concludes by reconfiguring the coteries as they were in phase one: Each right endpoint multicasts its result from phase two back down its row. The other PEs read the multicast data and combine it with their own to

complete the algorithm. ParallelPrefixRectangle requires at most $\log l + \log w + 3$ arithmetic and communication operations.

For arbitrary coteries, we must modify the second phase and add a third to the rectangle algorithm. We again begin by executing SeparateLines and ParallelPrefixLine to obtain the partial sums in the right endpoints. This time, however, the information carrying points do not neatly line up. Therefore, we instead run the analogous operations of SeparateSVCCs and ParallelPrefixSVCCs, completing the second phase as before with a multicast operation. This time, however, we are not yet finished: it is likely that some coteries will have multiple SVCCs. The third phase of the algorithm thus consists of running parallel prefix on the endpoints of the SVCCs and then (in a manner similar to phase two) returning the data back to the SVCC and the adjoining horizontal lines.

Here is the third phase in more detail. Select one PE in the coterie to be the accumulator. Then run a routine similar to CollectSparseData: the PEs containing the last sum in each SVCC are the $s_i \in S$. One by one, each s_i is selected, after which it multicasts its current sum to the coterie accumulator. The accumulator combines that data with the current sum, then multicasts the sum back to s_i. That SVCC accumulator then multicasts the data back up the SVCC (and its adjoining horizontal lines); those PEs read the data and combine it with their current sums to obtain their final results.

The first two phases require a logarithmic number of steps in the dimensions of the coterie. However, the third phase is not so easily bounded: the number of SVCCs from phase two is equal to the number of local minima (in terms of column ID) on the region boundary. It is possible to construct regions where this number is large, although such regions are unlikely in practice. This issue is discussed further in Section 8 .

6.2.3 Reduction

Like parallel prefix, reduce combines information from multiple PEs according to some operator; however, the partial sums are not saved. Reduce can be executed running ParallelPrefix and ignoring the intermediate results. Alternatively, reduce can be run using Combine; the choice depends on whether the operator is associative, and on the shapes of the regions.

7 Multi-Associative Vision Primitives

7.1 Multi-Associative Primitives

In order to efficiently map the multi-associative primitives to the coterie network, associative sets must be restricted to contiguous aggregates of PEs. Although the resulting capabilities are not as general as for the full multi-associative model, they are sufficient for many vision applications such as the non-uniform region processing algorithms presented in the next section.

1. Multicast, Local Compare, Activity Control. Local compare and activity control are basic PE operations, multicast is implemented through the coterie network.

2. Some/None Response. Direct implementation of coterie multicast.

3. Count Responders. CountResponders has been implemented in two ways: using SumCombine, and using a two dimensional reduction technique similar to the one presented in the last section. The latter (described in detail below) is usually the algorithm of choice, as it requires only $O(\log d)$ as opposed to $O(d)$ operations, where d is the maximum dimension of an aggregate. If the d's are all small, however—a result that can be obtained quickly by the controller, then we use the simpler SumCombine.

4. Select a Single Responder. We use the SelectSingleResponder algorithm of the previous section.

5. Create, Split/Merge Aggregates. Associative aggregates are created, split, and merged by opening and closing the coterie network switches to form electrically isolated regions (coteries).

6. Data Transfer Between Aggregates. Neighboring aggregates communicate by closing the coterie switches between them and using coterie multicast.

7.2 Vision primitives

Create Connected Component. This is the essential operation in creating multi-associative sets: once a set of coteries has been created, communication can take place within each coterie via multicast. In the four-connected version, each PE fetches the label of its four nearest neighbors via the mesh network and tests them against its own value. Switches are closed in the direction of the PEs whose labels are equal—or similar according to some measure—and opened otherwise (see Figure 19 for pseudo-code). The eight-way

connected version is slightly more complicated: to make the diagonal connection, the NW and/or NE switches of neighboring PEs must be set.

```
Coterie[N,S,E,W,NE,NW] := Open
Coterie[H,V] := Closed                                  {Initialize coterie switches }
FOR Neighbor := North TO West DO
    Equal := True                                       {Initialize flag }
    FOR BitNum := LabelLength - 1 DOWN TO 0 DO          {Compare own label with neighbor}
       Equal := Equal AND (Neighbor.Label[BitNum] = Label[BitNum])
    IF Equal THEN Coterie[Neighbor] := Closed           {If labels match, close switch to connect}
```

Figure 19: Algorithm to create four-way connected components.

Elect Leader. One of the essential parts of the information extraction process is to collect the region attributes in a small number of PEs for rapid transmission to the ICAP. An efficient method is for each ICAP element to contain a list of PEs, one per region, with which it shares memory, and to extract the region information from just those PEs. ElectLeader selects a unique PE within each set by running the multi-associative SelectSingleResponder operation.

Collect Info. There are several methods of collecting information in the leader PEs. One is to combine information as it arrives; this is a method used to count PEs. Another is to form a local array: this is necessary when collecting information that is not to be combined, such as the addresses of corner points. If the amount of information to be collected is large, then arriving data can also be transferred synchronously to the ICAP. That is, the CAAPP controller waits for the information to be read by the ICAP through the dual-ported memory before continuing with the next iteration.

Get Sorted List. Run ElectLeader to select an accumulator. Repeatedly run SelectMax on the key to get the next highest value, eliminating processors from contention after they have been selected. The PE with the maximum valued key after each iteration multicasts its data to the leader. If the keys are not unique, then SelectMax must be run, perhaps repeatedly, on the IDs after every iteration to break the ties. The complexity is roughly equivalent to the number of points in the set with the most points (N_m), times the SelectMax complexity. This routine is most useful when the number of points can be assumed to be relatively small compared to the number of points in the regions, as it usually is, for example, during the

computation of the convex hulls of the regions.

CollectOrderedCurveInfo. This procedure takes advantage of implicit ordering determined by the position of a PE on a curve. Although the basic concept is simple, in practice it is complicated by considerations of regions that are one pixel wide and regions that enclose other regions. We will only discuss the algorithm in terms of the curve being simple and closed.

Assume that the closed curve forms a coterie, that is, each PE on the curve has two closed links, one in the direction of the clockwise neighbor and one in the direction of the counterclockwise neighbor. The links are labeled by observing their relation to the local direction from inside to outside the region. Assume further that the PEs holding the information to be collected are tagged. The first step is to run ElectLeader on the tagged set to select an accumulator. The leader opens the switch connecting it to its counterclockwise neighbor in the loop. The loop is thus transformed into an open figure with the leader at one end. Every tagged PE in the chain now opens the switch connecting it to its clockwise neighbor. Next, each tagged PE multicasts its information. Because the tagged PEs have broken the coterie, only the information from one PE will reach the accumulator. The leader then multicasts a bit to the coterie, a bit that will only reach the PE whose information was just received. That PE now removes itself from the set and closes the switch to its clockwise neighbor, enabling further multicasts to pass through. The process is repeated until no tagged PEs remain.

8 Multi-Associative Vision Algorithms

FindExtremum. One of the most common uses of SelectMin/Max is to find the extrema in curves and regions. In these situations SelectMin (or SelectMax) is executed on the column or row address. In an $n \times n = N$ array, FindExtremum requires $3\log(n)$ steps.

GetSmallestCircumscribingRectangle. This algorithm follows immediately from application of FindExtremum using SelectMin and SelectMax on the row and column addresses, and thus requires $12\log(n)$ steps.

LabelConnectedComponents. Perhaps the most direct use of multicast on the CAAPP is labeling connected components. The connected components are created, as in the previous section, by closing the coterie switches in the direction of PEs with the selected value in common and opening the others, thereby forming coteries. ElectLeader is run to select a

single PE in each component. Because of the way that the switches are set, PEs in each coterie will receive the leader ID at no extra cost. This necessarily unique value is the label of the component.

PaintRegion. Although this routine consists of a single application of multicast, it shows the versatility of the IUA model to include graphics functions. Assume that a list containing the color and the address of the leader PE of each region is known in the ICAP, a scenario that occurs during the DARPA Integrated Image Understanding Benchmark (Weems et al. 1991). The ICAP loads the color into the memory of each leader. The leaders then multicast the color to their regions.

TraceOuterPerimeter. One of the steps required in the DARPA benchmark is identifying the outer perimeters of connected regions. The information locally available around a pixel is not sufficient to distinguish inner from outer borders, as occur, for example, in regions wholly enclosing multiple other regions. One way to distinguish the outer borders is use the following two-phase algorithm. In the first phase, FindExtremum is run to identify a point known to be on the outer perimeter. The second phase currently uses only nearest neighbor moves to traverse the perimeter and so is not described in this paper.

CreateBorder-CornerList. One application of CollectOrderedCurveInfo is extracting a list of corners. Although CollectSparseRegionInfo can be used here, there is a major advantage to extracting the corners in order: reconstructing the shape of the region is greatly simplified. Assume that the corner points are known. Run SeparateBorder and CreateSimpleCurve to obtain a coterie containing the border points. PEs determine the clockwise and counterclockwise links by examining neighbors to find the inside of the region. CollectOrderedCurveInfo now creates an ordered list of the corner points. This version of CreateBorder-CornerList can only be run if the resulting simple curves are closed.

GetAdjacentRegionLabels. This routine is used in building a region adjacency graph, a process essential to the region-merging phase of the segmentation algorithm in (Beveridge et al. 1989). The routine starts with the boundary PEs fetching the labels of the adjacent regions. Next, a modified version of CollectSparseRegionInfo is run on the set of boundary cells; the modification is that PEs whose data matches that just sent remove themselves from the set immediately, rather than sending the same label again. Since most of the cells will have redundant information, GetAdjacentRegionLabels will only require the number of iterations equal to the number of adjacent regions, not the number of border PEs.

MergeRegions. During the region merging phase of a segmentation, the ICAP will determine the regions to be merged; the following procedure is then executed on the CAAPP. Leader PEs in pairs of regions are sent the label of the neighboring region with which each is to merge, along with a bit telling whether it is the region with the higher or lower ID. The leaders of the lower ID regions multicast the label of the neighboring region to their coteries. The PEs on the border between the regions close the coterie switches in the direction of that other region. The leader PE with the higher ID is selected to be the new leader of the region. The "former" leader then multicasts its region characterization info (size, etc.), which is read by the current leader and combined. Alternatively, the ICAP could combine and download the new region characterization information. Also, some parameters are not easily combined and must be recomputed.

Count(Selected)PEs. An essential part of extracting low-level vision events is the characterization of sets of points in a component; some of the basic parameters needed are a count of the total points in the region and the number of points having some property. In the latter case we assume that the comparison phase has already taken place, so we are really counting "selected PEs."

CountSelectedPEs uses the coterie reduction algorithm present above. Each selected PE is given a 1, the rest 0's. The coterie is reduced using the addition operator.

CountPEs uses the same second and third phases as CountSelectedPEs, but the first phase can be simplified. After SeparateLines and GetOffset, the east endpoint already has the number of points in the Line, thereby avoiding the need for ReduceLine.

As mentioned previously, the first two phases require at most $2 \log d$ communication steps, where d is the maximum dimension of the largest set. The execution of the thirdphase, however, is proportional to the number of local minima (in terms of column ID) on the region boundary. Since this number could be large, phase three has been modified: the following is an example of the use of array associativity to bound a multi-associative algorithm.

After each iteration of phase three, the global controller (ACU) performs a CountResponders operation on the leader PEs of the regions not yet having completed. When this number falls below a threshold, say ten, then an array associative version of Count(Selected)PEs takes over; the algorithm finishes by performing CountResponders on each of the remaining regions. The hybrid version of phase three guarantees that the number of iterations required to complete the algorithm will be small in virtually all cases. Although this algorithm is

still suboptimal when there are many regions all with large numbers of local minima, the likelihood of such patterns arising out of an image segmentation is very small.

GetMean. The next two algorithms are derived from the standard associative model (Foster 1976): they can now be applied multi-associatively as all of those basic operations have been implemented. The algorithm to find the mean is similar to SelectMax in that both are bit serial over a k-bit label, and both run from high to low order ($k-1$ to 0). Assume we are trying to find the mean of a label L over the PEs in a region. We sum the L's by successively counting the PEs with the ith bit of L set, and scaling that count by 2^i. We start at the high order so that scaling can be accomplished with one shift per iteration.

Select PEs with bit $k-1$ of L set. Run CountSelectedPEs to get the count. Add the count to the accumulator (zero to start). Shift the accumulator left 1 bit. Repeat this process for bits $k-2$ to 0 of L, but without shifting after the final iteration. GetMean requires a number of iterations equal to $\log(Max(L))$; each iteration contains one add, one shift, and one CountSelectedPEs operation. See Figure 20 for pseudo-code.

```
Sum := 0                                        {Initialize Sum}
FOR BitNum := LabelLength - 1 DOWN TO 0 DO
    Activity := Label[BitNum]                   {Select PEs with bit set}
    Sum := (Sum << 1) + CountSelectedPEs()      {Count number of active PEs and scale}
Mean := Sum/CountPEs()                          {Count all PEs and divide Sum}
```

Figure 20: Algorithm to compute mean.

GetMedian. The method used is analogous to binary search: we find the range of possible values and successively halve the interval on each iteration. Start by running SelectMin and SelectMax to find the lower and upper bounds (L and H), and CountPEs to obtain C, the number of PEs in the region. Let the initial guess $G = \frac{L+H}{2}$. Select the PEs with a label greater than G; then run CountSelectedPEs again. Depending on whether the count is higher or lower than $C/2$, the new guess G is either $\frac{H+G}{2}$ or $\frac{L+G}{2}$. H or L is also updated. The algorithm converges after $\log(Max(H) - Min(L))$ iterations.

Histogram within Regions. The multi-associative version of histogram permits each region to use a different set of bins. Run ElectLeader to select an accumulator. For each bin, multicast the value (or range of values) of that bin. PEs are selected according to whether their value matches that of the bin. Run CountSelectedPEs to get the bin count. The

algorithm requires a number of iterations equal to the number of bins, and each iteration contains one CountSelectedPEs operation.

ConvexHull. The convex hull of a set of points S is defined as the smallest convex set contained in S. Intuitively, the convex hull in a plane can be found by conceptually wrapping S with a rubber band and eliminating the interior points (Preparata and Shamos 1985). Two leading methods for finding the convex hull on a serial processor are the Graham Scan (Graham 1972) and the Jarvis March (Jarvis 1973).

The Graham Scan works as follows: A point p, known to be on the hull, is chosen. Without loss of generality, let p be the point with the smallest X-coordinate, where the smallest Y-coordinate breaks any ties. For all other points s_i in S, calculate the slope of line segment $\overline{ps_i}$. Sort the s_i's by slope. Traverse the list of points in order: for each point, compute the angle it makes with its predecessor and successor. If the angle is reflex, then eliminate that point. The serial Graham Scan is of complexity $O(N \log N)$, the minimum time required for the sort. The scan phase requires only $O(N)$ steps because at most N points can be either eliminated or traversed.

The Jarvis March is analogous to wrapping the points in a package, one hull *edge* at a time. Again, the algorithm begins with the selection of a point p known to be on the hull. The slope of each segment $\overline{ps_i}$ is calculated, and the next point on the hull selected by finding the segment $\overline{ps_i}$ making the smallest angle with respect to the positive X-axis. This process is repeated for all h points on the hull; the Jarvis March thus has a complexity of $O(hN)$. In general, the Jarvis March should be used when the expected number of points on the hull h is less than $\log N$.

The multi-associative versions of these algorithms are assumed to be over sets of points in connected components. Also, we assume that the points in S are mapped to PEs according to their row and column coordinates.

The first step in the parallel Graham Scan is to select an extreme point p in S by using ElectLeader. The other s_i calculate their slopes with respect to p. The next step is to sort the PEs by slope by using a variation of GetSortedList, modified so that each s_i retains the IDs of its predecessor and successor points. Using these IDs and that of the coterie leader, the s_i determine whether they are on the hull. If a PE does not represent a point on the hull, it removes itself from S.

However, this procedure may require a few iterations to simulate the backtracking that

is sometimes necessary in the serial version. Therefore, the above procedure is repeated until no points drop out. Typically only two iterations are sufficient, although it is possible to construct cases where more are required. As we have not proven a constant bound on the number of iterations, we can only conjecture that the complexity is $O(N)$, where N is the number of points in S in the coterie that takes the longest to terminate. This assumes that SelectMin in GetSortedList is counted as a unit operation.

The Jarvis March can be parallelized somewhat more easily: A point p on the hull is found using ElectLeader, and its ID is simultaneously distributed to the rest of S. The s_i calculate the angle formed by $\overline{ps_i}$ with the column axis. Calling SelectMin locates the next point on the hull. The procedure is then repeated until the PE forming the smallest angle is p, in other words until the loop has closed. The complexity is clearly $O(h)$, the number of points on the hull, if SelectMin is counted as a unit operation.

9 Summary and Conclusion

Summary

Parallel low-level vision involves the characterization, manipulation, extraction of information from, and communication between, non-uniform aggregates of PEs. Processing of data dependent configurations of PEs has special requirements often met by associative processing. However, the result is that aggregates are operated upon one at a time (region-serial). To deal with cases where the number of aggregates is large, multi-associative processing is proposed; that is, associative primitives on aggregates simultaneously (region-parallel). The immediate consequence of using the associative primitives in parallel is the multi-associative extension of existing associative algorithms. The multi-associative split and merge operations result in further capabilities: support for divide-and-conquer, the direct use of the geometry of an aggregate to make explicit implied orderings, and the ability to retain multiple, partial results.

The viability of multi-associativity on a SIMD processor was demonstrated by mapping it onto the coterie network. Some basic results were presented, including definitions and algorithms for the construction of new coterie structures, and a parallel prefix algorithm that completes in $O(\log d)$ time-steps for most coteries. Some of the direct consequences of the impementation of multi-associativity to parallel low-level vision processing are efficient, region parallel, algorithms to: collect and extract region information (sparse, ordered bound-

ary, etc.), exchange information among regions (building region adjacency graphs), combine regions, count pixels (and pixels with specified attributes), find the circumscribing rectangle and convex hull, and label connected components. Array associative and multi-associative processing can also be used together as was shown in a hybrid Count algorithm: for most regions the multi-associative part finishes quickly, the remaining regions are then processed array associatively. The multi-associative primitives can also be used to construct region-parallel versions of existing associative algorithms such as median, mean, and histogram.

Conclusion and Future Work

The problem was to come up with SIMD algorithms that would be efficient when operating in parallel on regions of pixels with data dependent (usually non-uniform and irregular) shapes. We have constructed algorithms using the coterie network where most of the shape dependence has been removed.

Although some of our results are of theoretical interest, we are mostly interested in improving performance of the CAAPP—by itself, and as a part of the IUA system—in solving real-time vision problems. Therefore, the real measure of our success will come when we have tried our approach on a large number of test cases. A more practical problem is to find the break-even point between associative and multi-associative processing, an important problem in system tuning as the number of regions being processed can vary widely.

Acknowledgments

We would like to thank Seth Malitz, Jim Burrill, Deepak Rana, Mike Rudenko, Ross Beveridge, Bob Collins, and Bruce Draper for their useful comments.

References

Annexstein F, Baumslag M, Herbordt MC, Obrenic B, Rosenberg A, Weems CC (1990) Achieving Multigauge Behavior in Bit-Serial SIMD Architectures via Emulation. Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation, pp 186-195

Batcher KE (1980) Design of the Massively Parallel Processor. IEEE Transactions on Computers C-29(9):836-840

Beveridge JR, Griffith J, Kohler RR, Hanson AR, Riseman EM (1989) Segmenting Images Using Localized Histograms and Region Merging. International Journal of Computer Vision 2(3):311-347

Blelloch GE (1989) Scans as Primitive Parallel Operations. IEEE Transactions on Computers C-38(11):1526-1538

Brolio J, Draper BA, Beveridge JR, Hanson AR (1989) ISR: A Database for Symbolic Processing of Computer Vision. IEEE Computer 22(12):22-32

Draper BA, Collins RT, Brolio J, Hanson AR, Riseman EM (1989) The Schema System. International Journal of Computer Vision 2(3):209-250

Duff MJB (1978) Review of the CLIP Image Processing System. Proceedings of the National Computing Conference AFIPS, pp 1055-1060

Erman L, Hayes-Roth F, Lesser V, Reddy D (1980) The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. Computing Surveys 12(2)

Falkoff AD (1962) Algorithms for Parallel Search Memories. Journal of the Association for Computing Machinery 9(4):488-511

Foster CC (1976) Content Addressable Parallel Processors. Van Nostrand Reinhold Company, New York

Graham RL (1972) An Efficient Algorithm For Determining the Convex Hull of a Planar Set. Information Processing Letters 1:132-133

Hanson AR, Riseman EM (1987) The VISIONS Image Understanding System–1986. In: Brown C (ed) Advances in Computer Vision. Erlbaum, Hillsdale, New Jersey

Herbordt MC, Weems CC, Corbett JC (1990) Message Passing Algorithms for a SIMD

Torus with Coteries. Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures, pp 11-20. Also in Computer Architecture News 19(1):pp. 69-78.

Hillis WD, Steele Jr. GL (1986) Data Parallel Algorithms. Communications of the ACM 29(12):1170-1183

Hunt DJ (1981) The ICL DAP and its Application to Image Processing. In Duff MJB, Levialdi S (ed) Languages and Architectures for Image Processing, Academic Press, London

Karp RM, Ramachandran V (1988) A Survey of Parallel Algorithms for Shared-Memory Machines. Technical Report 88.408, University of California at Berkeley

Jarvis RA (1973) On the Identification of the Convex Hull of a Finite Set of Points in the Plane. Information Processing Letters 2:18-21

Li H, Maresca M (1989) The Polymorphic-Torus Architecture for Computer Vision. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11(3):233-243

Marr D (1982) Vision. W.H. Freeman, San Francisco

McCormick BT (1963) The Illinois Pattern Recognition Computer—ILLIAC III. IEEE Transactions on Electronic Computers C-12(12):791-813

Miller R, Prasanna Kumar VK, Reisis D, Stout QF (1988) Meshes With Reconfigurable Buses. Proceedings of the MIT Conference on Advanced Research in VLSI, pp 163-178

Prasanna Kumar VK, Reisis D (1989) Image Computations on Meshes with Multiple Broadcast. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11(11):1194-1202

Preparata FP, Shamos MI (1985) Computational Geometry: An Introduction. Springer-Verlag, New York

Rana D, Weems CC (1990) A Feedback Concentrator for the Image Understanding Architecture. Proceedings of the International Conference on Application Specific Array Processors, pp 579-590

Riseman EM, Hanson AR (1989) Computer Vision Research at the University of Massachusetts-Themes. International Journal of Computer Vision 2(3):199-207

Rosenfeld A (1984) Image Analysis: Problems, Progress and Prospects. Pattern Recognition 17(1):3-12

Tilton JC (1988) Image Segmentation by Iterative Parallel Region Growing With Applications to Data Compression and Image Analysis. Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation, pp 357-360. Tucker LW (1988) Architecture and Applications of the Connection Machine. IEEE Computer 21(8):26-38

Weems CC (1984) Image Processing on a Content Addressable Array Parallel Processor. Technical Report 84-14, Department of Computer and Information Science and Ph.D. Dissertation, University of Massachusetts

Weems CC, Levitan SP, Hanson AR, Riseman EM, Nash JG, Shu DB (1989) The Image Understanding Architecture. International Journal of Computer Vision 2(3):251-282

Weems CC, Riseman EM, Hanson AR, Rosenfeld A (1991) The DARPA Image Understanding Benchmark for Parallel Computers. Journal of Parallel and Distributed Computing 11:1-24

Weems CC (1991) Architectural Requirements of Image Understanding with Respect to Parallel Processing. Proceedings of the IEEE 79(4):537-547

Willebeek-LeMair M, Reeves AP (1990) Solving Nonuniform Problems on SIMD Computers: Case Study on Region Growing Journal of Parallel and Distributed Computing 8:135-149