A Computational Framework and SIMD Algorithms For Low-Level Support of Intermediate Level Vision Processing

M. Herbordt
C. Weems
M. Scudder

COINS TR91-26

March 1991

A Computational Framework and SIMD Algorithms for Low-Level Support of Intermediate Level Vision Processing[1]

(Preliminary Version)

Martin C. Herbordt Charles C. Weems Michael J. Scudder

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

Abstract

Computation on and among data sets mapped to irregular, non-uniform, aggregates of processing elements (PEs) is a very important, but largely ignored, problem in parallel vision processing. Associative processing [11] is an effective means of applying parallel processing to these computations [33], but is often restricted to operating on one data set at a time. What we propose is an additional level of parallelism we call *multi-associativity* as a framework for performing associative computation on these data sets simultaneously. In this paper we introduce algorithms developed for the Content Addressable Array Parallel Processor (CAAPP) [35] to simulate efficiently *within aggregates of PEs simultaneously* the associative algorithms typically supported in hardware at the array level. Some of the results are: the efficient application of existing associative algorithms (e.g. [10, 11]) to arbitrary aggregates of PEs in parallel, and the development of new multi-associative algorithms, among them parallel prefix and convex hull. The multi-associative framework also *extends* the associative paradigm by allowing operation on and among aggregates themselves, operations not defined when the entity in question is always an entire array. Two consequences are: support of divide-and-conquer algorithms within aggregates, and communication among aggregates. The rest of the paper describes a mapping of multi-associativity onto the CAAPP, and numerous multi-associative algorithms.

[1]This work was supported in part by the Defense Advanced Research Projects Agency under contracts DACA76-86-C-0015, and DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Laboratory; by the Air Force Office of Scientific Research, under contract F49620-86-C-0041; and by a Coordinated Experimental Research grant from the National Science Foundation (DCR 8500332)

1 Introduction

According to one popular methodology [22, 13], 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 or orientation. When abstracted, these collections of pixels can be represented symbolically as edges, lines, curves, patches, blobs, regions, and their combinations and intersections [6]. In the case where a group of pixels is determined to be a line, for example, it could be represented by two endpoints. Certain algorithmic approaches to low-level vision are well established, although their massive computational requirements make them impractically slow. One consequence has been the design of massively parallel processors for low-level vision applications. Despite progress in this area, research into algorithms is required to make effective use of these machines. In this paper we present a new machine independent computational model, numerous vision algorithms using that model, and a mapping of the model onto the Content Addressable Array Parallel Processor (CAAPP), a low-level vision processor that is part of the Image Understanding Architecture (IUA) being developed at the University of Massachusetts and Hughes Research Laboratories.

The work presented here is part of a machine vision research program to develop methodologies, determine their computational requirements, design hardware that meets those requirements, and provide algorithms for that hardware [29]. In turn, algorithmic development inspires new types of hardware support, bringing to light new computational *possibilities*, influencing the way we think about machine vision. Similar research efforts can be found in [32, 21, 20, 26, 1] and others, with more or less emphasis on either the architecture or the vision end of the research. A common thread in these studies is that low-level vision involves more than the window-based operations that dominated earlier research. What makes our work unique is that our programming model supports broadcast over arbitrarily shaped, contiguous, regions as a method for taking maximum advantage of the spatial proximity that dominates low-level vision computation, even when the communication patterns are irregular or anisotropic.

Until recently, our approach when confronted with the need to solve problems on mul-

tiple irregular aggregates of PEs has been ad hoc: we have often achieved good results [36], but have not previously used any uniform technique. Precedents do exist, however, demonstrating that careful orchestration of SIMD communication can promote complex behavior: the sorting algorithms of Thompson and Kung [31] and the connected components algorithm of Nassimi and Sahni [25] are examples, though the communication is still regular. Willebeek-LeMair and Reeves [37] have embedded binary trees in meshes to implement broadcast and reduction primitives on non-uniform, arbitrarily, shaped contiguous regions, and applied those operations with great success to parallel region segmentation.

We begin with a brief review of the vision methodology and computational requirements driving our research. Then, following a brief discussion of associative processing in general, we present a new extension we call "multi-associativity", that allows us to solve multiple problem instances simultaneously, given conditions that usually occur in low-level vision computation. The architecture of the target machine, the CAAPP (including the hardware support for low-level vision), are presented next. Next comes a sketch of the actual mapping of multi-associativity onto the CAAPP. We close with numerous examples of multi-associative vision algorithms.

2 A Machine Vision Methodology

2.1 Computational Requirements

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 pixels are mapped to individual PEs, and that 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 X, as well as other computations. An example of world knowledge processing might be constrained model matching between "stored models in the form of generalized relational structures [30]", 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 that arises 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 possibly several, 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, collinear, 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) [6], 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 also includes 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 information carrying points 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 innate 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.

2.2 Associative Processing

In the last 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 first we 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 the classical model of associative computation [11]:

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) or a Count. 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 [33, 35] 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 always possess a local compare, and an activity 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) [28]. 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.

2.3 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 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 it assumes 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 allowing the partition of the processor array into $k \leq N$ sets $S_i \subseteq \{S_1, \ldots, S_k\}$ of PEs. Next, we define a set of 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 set*** 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}.

In addition, we define two operations on the associative sets themselves. These are Split and Merge. Split allows, for all sets in parallel, the sets S_i to be split into any number of new sets. Merge allows any number of sets to be combined into a single set. We now have five capabilities defining multi-associative processing:

1. Multicast by a subset/Local Compare/Activity Control
2. Some/None Response to a subset
3. Count Responders to a subset
4. Select a Single Responder
5. Split/Merge sets

The first four operations map all associative algorithms defined over processor arrays to associative sets in parallel. The last operation gives the model new power; 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 restriction of the multi-associative model: there is still only one controller. Therefore, only algorithms with a branching factor $\ll$ than the number of associative sets will run efficiently. However, since most low-level vision applications lend themselves to algorithms that meet this criterion, the restriction also has a positive side: there is greater hardware efficiency because only a single controller is needed.

3 The IUA, the CAAPP, and the Coterie Network

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 [35]; 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. 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 [7], 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 [9], but the details have not yet been fully defined.

The intermediate level must provide several functions. One is assisting the CAAPP with feature extraction by providing control and dynamic structures; shared 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 shared memory.

The Content Addressable Array Parallel Processor (CAAPP) consists of 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" which is 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 [8], the MPP [3], and the DAP [17]. 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 [23] and the propagate operation in the CLIP-4 [8], and is similar to those proposed by [24, 20]. In order to distinguish broadcast by PEs from the usual broadcast by the controller, we refer to this operation as "coterie multicast".

The coterie network is one powerful addition that the CAAPP has over conventional associative processors. Each PE in the CAAPP controls a set of eight switches (see Figure 1), 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 for creation of eight-way connected regions. The isolated groups of processors (see Figure 2), 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 the same coterie.

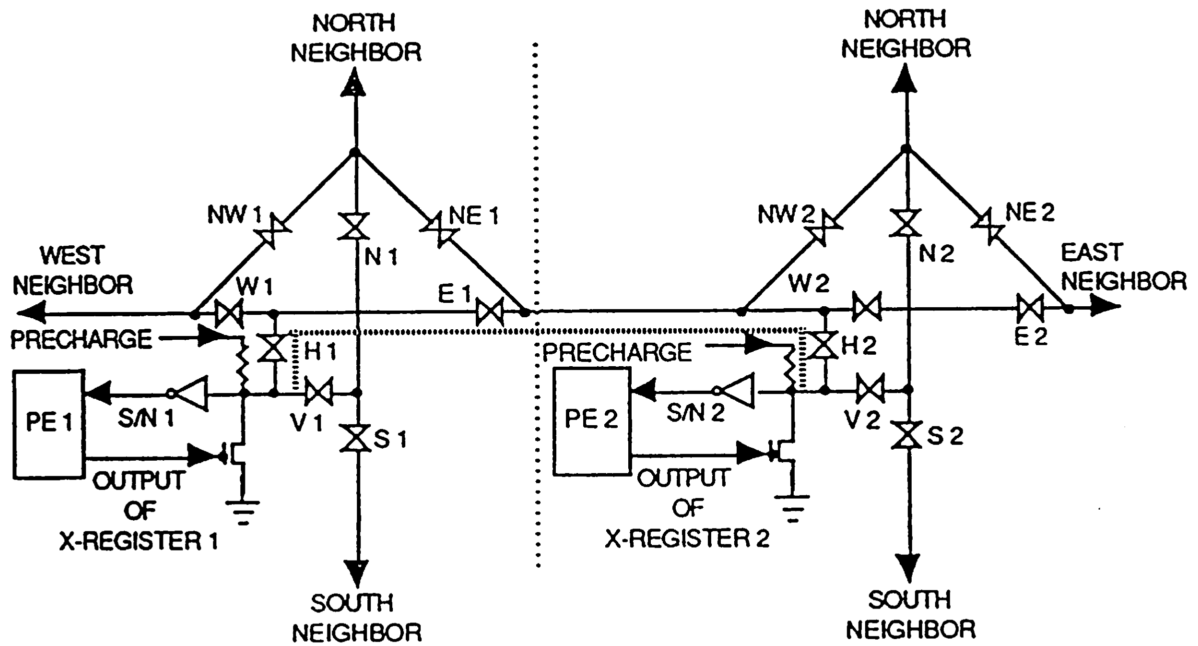

Figure 1: Two PEs and the coterie switches they control

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 [24], and the polymorphic-torus [20]. 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.

The rest of this paper is concerned with mapping multi-associativity onto the CAAPP, and examples of multi-associative algorithms.

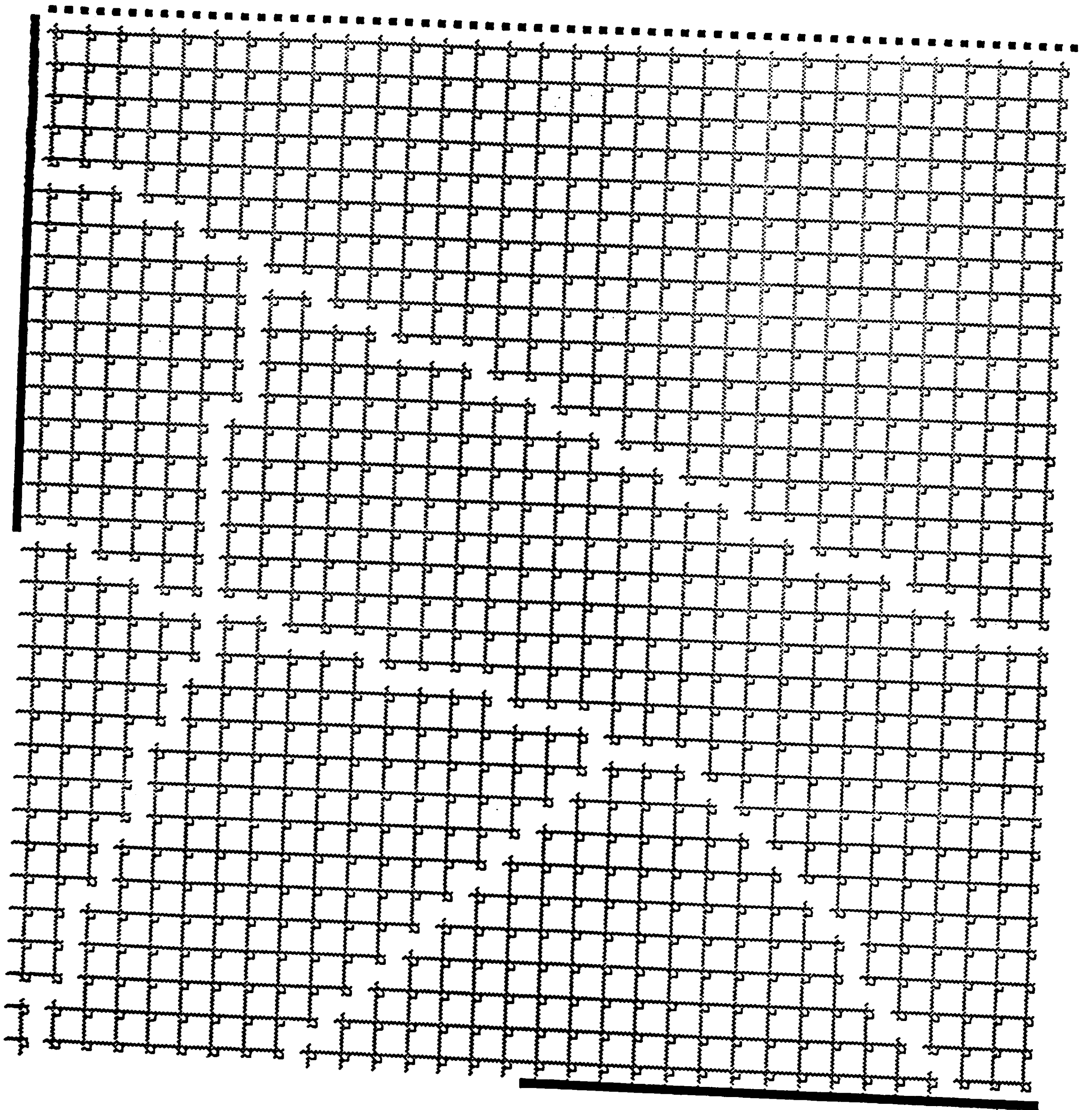

Figure 2: Coterie switch settings of part of an image after running four-way connected components

4 Mapping Multi-Associativity onto the CAAPP

At this stage of our research, it is not yet practical to map the completely general model of multi-associativity onto the CAAPP, as the correspondence between the hardware support and the multi-associative model is not precise. However, through the use of the available hardware, software emulation, and some restrictions, we have created a nearly complete implementation. These restrictions basically consist of requiring that associative sets be contiguous, not a large loss since that is a characteristic of most of the events extracted at the lowest level. We begin by describing some basic techniques used in CAAPP programming, followed by a description of how the multi-associative model is implemented.

4.1 Basic CAAPP Operations

LoadPeId. Always available to each PE is its 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.

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 machine cycles 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.

Open/Close Switches. PEs control switches that determine whether a multicast signal will pass through the section of the coterie network they control. 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.

4.2 Selected Programming Tools

We present here some of the routines that form the programming tools used to implement the basic associative operations, together with the restrictions on the sets of PEs over which they are defined.

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 [14, 15].

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.

Select Max/Min. The goal is to select the PEs with the greatest value within a coterie. The method is to apply a standard associative array algorithm [10] to regions. 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

```
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 3: Algorithm to find and distribute the maximum valued label in a region

starts with the high order bit $(k-1)$. PEs multicast bit $k-i$: if any are ones, 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. The restriction on these routines is that they work only for coteries, and thus only on sets of contiguous PEs. See Figure 3 for pseudo-code.

Ordered Data Collection. This technique works when the coteries are *simple curves*, defined on the CAAPP as a coterie where each member has exactly two distinct neighbors also members of the coterie, except for the end points which have only one. An end point (or max-ID point in the case of a closed curve) is selected to be the collector of the information. A rough outline of the technique, is for each PE to determine 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. This procedure requires time linear in the number of PEs holding data, and so is only efficient if that number is small.

Data Reduction Using Doubling. To use data reduction through the basic divide-and-conquer primitive of iterative doubling, the shape of the coterie must be restricted further: it must be either a horizontal or vertical line, or a *monotonic curve*. A vertical monotonic curve (the definition for the horizontal case is analogous) is defined as simple curve where each member has a row ID $\leq$ the row ID of the neighbor toward the bottom end of the curve. Further, only one PE per row (assumed to be the right-most) carries data.

The procedure begins by selecting the PE with the minimum row ID, having that PE multicast the row ID to the rest of the coterie, and having the receiving PEs subtract that index from their own row IDs. The result is an enumeration of the data carrying PEs in

each coterie. The details of the rest of this operation will be given below, but the result is a form of divide-and-conquer that results in a reduction with logarithmic complexity.

4.3 Multi-Associative Primitives

We now present the implementation of the basic multi-associative operations on the CAAPP. It should be clear from the above discussion, that to use the hardware support effectively, associative sets must be restricted to contiguous sets of PEs. Although the resulting capabilities are not as general as for the full associative model, they are sufficient for the low-level vision algorithms presented in the next section.

Create Associative Set. Associative sets are created by opening and closing the network switches in order to form coteries. These switch settings can be masks stored in memory, or may be computed locally according to any number of parameters.

Local Compare, Activity Control. These are basic SIMD operations: the first is a part of the CAAPP PE repertoire, the second is simply Select.

Multicast by a Subset, Some/None Response. These both use the multicast operation of the coterie network.

Select a Single Responder and Count Responders. These operations are somewhat different than the others because they are not supported directly, and must therefore be emulated in software. SelectSingleResponder is implemented by running SelectMax on the PE ID, and requires $3 \log N$ steps. CountResponders can be implemented in two ways: using SumCombine or using a two dimensional extension of the reduction technique outlined above. 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 distance from any PE to its leader. If d is small, however, a result that can be obtained quickly by the controller, then we use the simpler SumCombine.

Split/Merge Sets. Sets are split and merged using the same methods used to create them in the first place.

5 Multi-Associative Vision Algorithms

We have divided this part into two sections: in the first the multi-associative primitives are built up into useful functions, in the second the vision algorithms themselves are presented. To avoid redundancy, the routines are often presented as applied to one data set; in all cases, however, they can be applied to any number of data sets simultaneously, as defined earlier.

5.1 Basic Operations

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 gets 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. 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.

Separate Border. Once connected components have been created, it is possible to form coteries made up of border PEs. Each PE sets a flag if any of its switches are open. PEs then open connections in the directions of PEs whose flags are cleared (see Figure 4).

Separate Lines. Some routines are constructed to run first in one dimension and then the other. In order to restrict coteries to horizontal or vertical lines, two methods are possible. One is to open the V or the H switches, respectively. The other is to leave the H and V switches closed, but to open the N and S, or the E and W switches.

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 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 the method used in CountPEs. Another is to

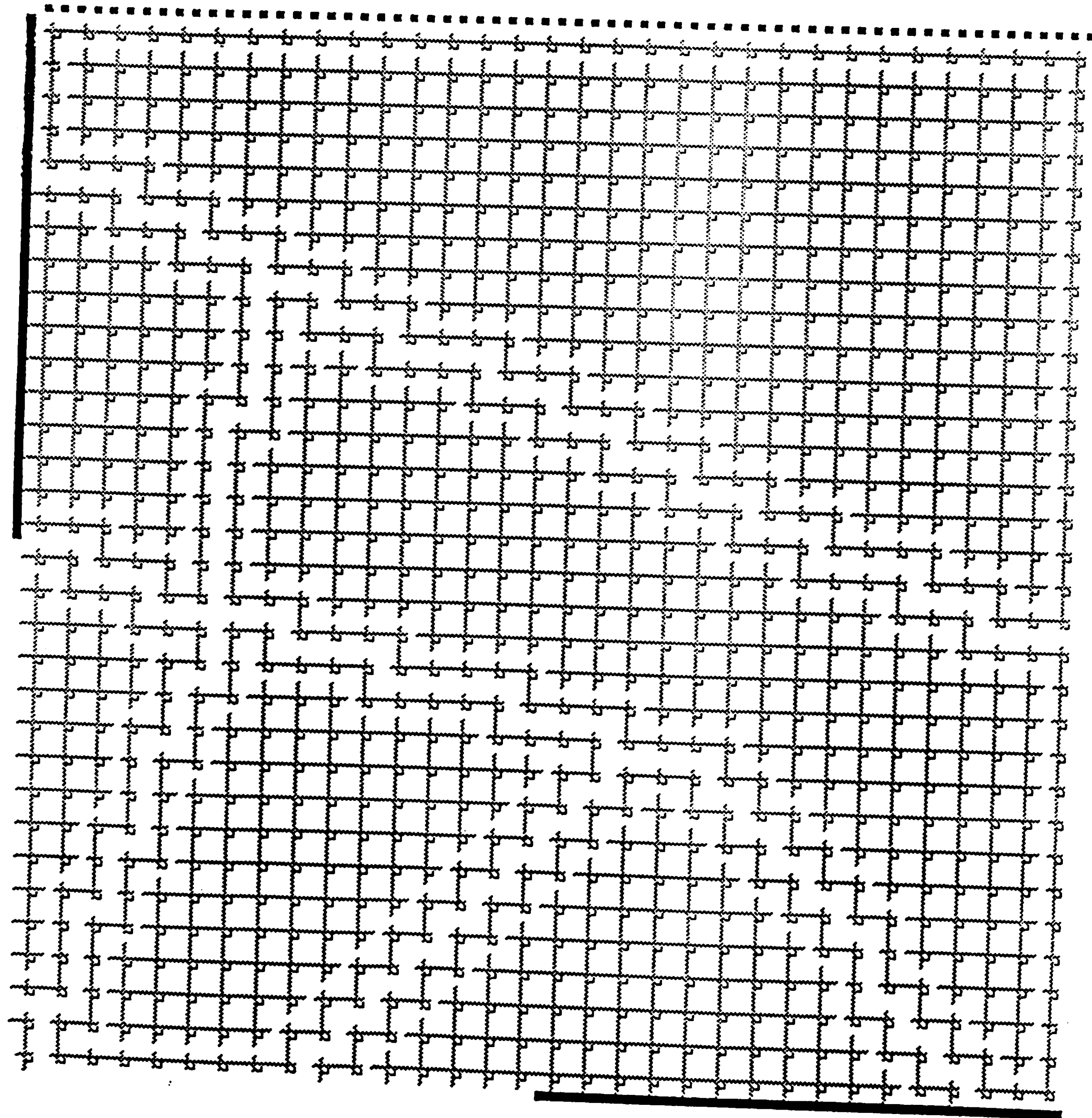

Figure 4: Coterie switch settings of same image section after running SeparateBorder

form an 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 great, then arriving data can also be removed synchronously to the ICAP. That is, the CAAPP controller waits for the information to be read through the shared memory after each iteration before continuing.

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 key value 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 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 is, for example, during the running of a convex hull.

Collect Sparse Region Info. If a small number of arbitrarily placed PEs per region contain information that is to be collected, either in a list or cumulatively, then an efficient method is the repeated use of SelectSingleResponder and Multicast. First run ElectLeader to select an accumulator. Then iterate the following for the number of points in S, the set of points containing the information to be gathered. Execute SelectSingleResponder to select the next point from S. The selected PE multicasts its information to the accumulator, and removes itself from S. The accumulator reads and processes that information.

CollectOrderedCurveInfo. This procedure takes advantage of implicit ordering determined by the position of a PE on a curve. Although the basic concept 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.

Parallel Prefix for Monotonic Curves. This procedure works for horizontal and vertical lines, as well as monotonic curves First we define parallel prefix: Given an array $[x_1, \ldots, x_n]$ of n elements, one per 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. In this section we present a multi-associative parallel prefix for horizontal lines (see Figure 5 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 and monotonic curves.

To execute parallel prefix, there must be at least an implicit ordering to the PEs, in this case it is the distance from the west end-point. Therefore, the first step of the algorithm is for PEs to calculate their offsets. Each PE checks its W coterie switch to determine whether it is on the west end of the line. If so, it multicasts its column position. All PEs on the line read the data, and subtract it from their own column position to obtain an offset.

The rest of this algorithm is illustrated in Figure 6. The binary numbers represent the 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

```
{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 := ColumNumber - Base
{Initialize switch masks. All compare operations using OpenLeftMask}
{and OpenRightMask use only the rightmost BitNum + 1 bits}
OpenLeftMask := 0
OpenRightMask := 1
FOR BitNum := 0 TO ColumnNumberLength - 1 DO
    IF (OpenEastMask = Offset) Coterie[E] := Open
    IF (OpenWestMask = Offset) Coterie[W] := Open
    Multicast(Data,DataLength,Coterie[E] = Open,Temp,Coterie[E] = Closed)
    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 5: Algorithm for parallel prefix on a horizontal line

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.

Reduce Line/Monotonic Curve. Sometimes results must be collected, but without the need to save the intermediate values. Although this procedure is simpler than parallel prefix, it still requires a logarithmic number of steps. Therefore the simplest method is to run ParallelPrefix and to ignore the intermediate values.

5.2 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,

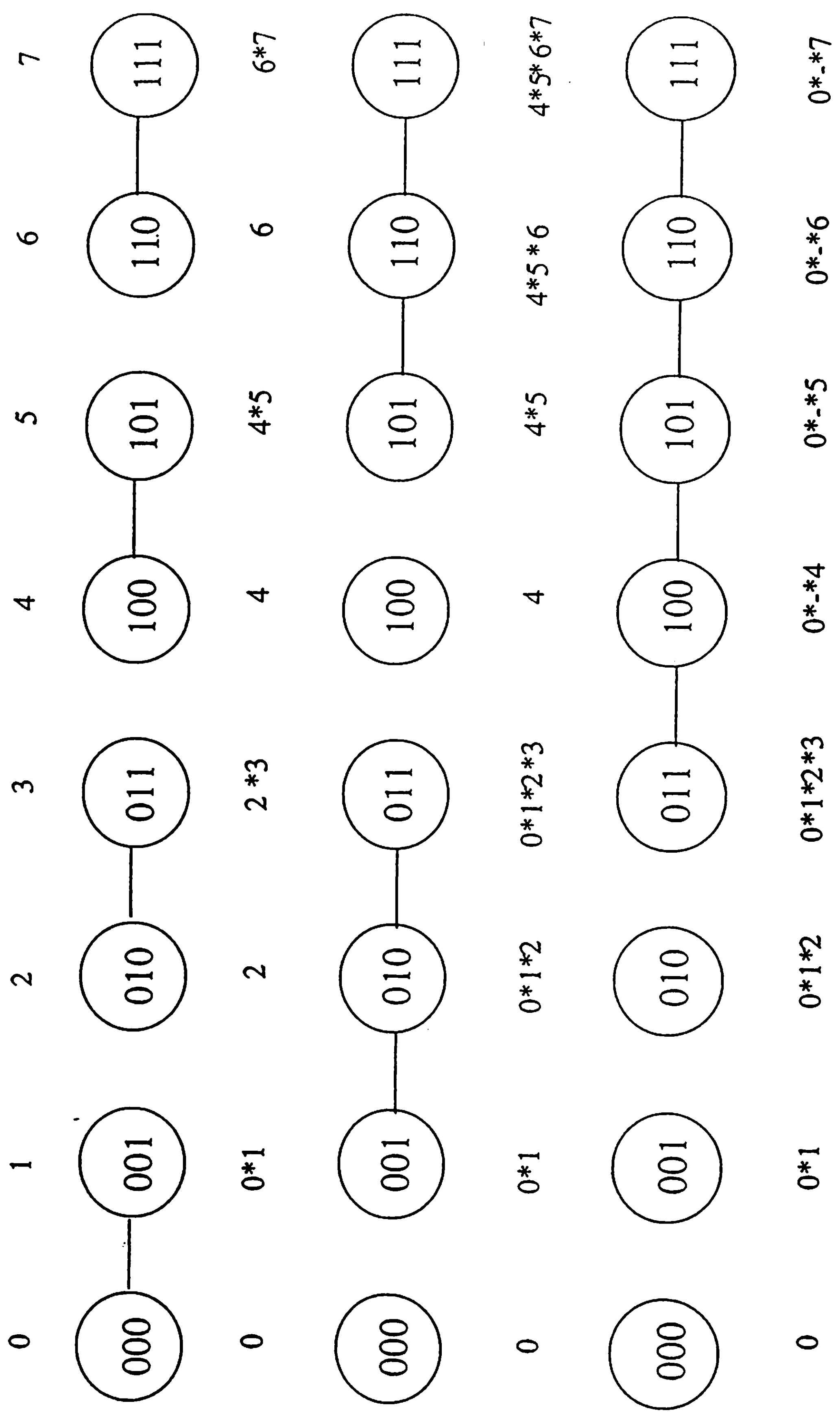

Figure 6: Illustration of parallel prefix on a line

and thus requires $12\log(n)$ steps.

LabelConnectedComponents. Perhaps the most direct use of multicast and the coterie network 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 in [34]. The ICAP loads the color into the memory of each leader. The leaders then multicast the color to their regions.

TraceBorder. One of the steps required in [34] was 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 doughnut shaped regions. One way to distinguish these borders is to first run ElectLeader to identify a point known to be on the outer perimeter, and from there transmit a message to label the border. The outer perimeter will be marked while the borders around holes will not. The traversal section of the algorithm currently uses only nearest neighbor moves and so will be described elsewhere.

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 to create a coterie from 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 on simple regions as specified above.

GetAdjacentRegionLabels. This routine is used in building a feature adjacency graph, a process essential in creating complex intermediate-level data structures, as well as to the region-merging phase of the segmentation algorithm in [4]. 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 repeated information, GetAdjacentRegionLabels will only require the number of iterations equal to the number of border regions, not the number of border PEs.

MergeRegions. During the merging phase of a feature grouping algorithm, 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. 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 that what we are really counting is the number of "selected PEs." This routine runs in three phases. The first is to divide the region into single PE width row strips, count the PEs in each strip, and leave the result at the right end. In the second phase, the right ends form a set of curves, the data of which is accumulated at an end point. Finally, the data from all of these accumulators is collected.

The first phase begins with SeparateLines to form horizontal strips. The left and right end-points of each strip are identified; the right end-points are selected to be the strip accumulators. For the next step CountPEs and CountSelectedPEs diverge. In CountPEs, each left end point multicasts its column ID, the corresponding right end point reads it and subtracts it from its own column ID. In CountSelectedPEs, the routine ReduceLine is run instead so that only the selected PEs will be counted. The strip phase of the algorithm runs

in constant time for CountPEs because only a single multicast and subtraction is needed, while CountSelectedPEs requires log time, the complexity of ReduceLine.

At the beginning of the second phase, only border PEs will have relevant information, and we could simply collect it in time proportional to the number of accumulator points. But instead we perform another log time reduction step, as follows. SeparateBorder to form a coterie consisting of the region border. Next, create monotonic vertical curves made up primarily of the strip accumulators of phase 1. This is done by finding and labeling the local minima and maxima of the border, and opening the switches so that these PEs become end-points. Next, execute ReduceMonotonicCurve so that the local minima accumulate the running count. The final phase of this algorithm consists of ElectLeader to select an overall region accumulator and CollectSparseRegionInfo to combine the subtotals.

The first two phases require at most $2 \log d$ communication steps, where d is the maximum dimension of the largest set. However, the third phase is not so easily bounded: the number of accumulators remaining 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. To deal with most of these cases, phase three is modified; the following is an example of the use of global feedback to bound an 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 a globally 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 problems remain when there are tens of regions all with tens of local minima, the likelihood of such patterns arising as the output of an image segmentation is very small.

GetMean. The next two algorithms are derived from the standard associative model [11]: they can now be applied multi-associatively as all of the 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 that have the ith bit of L set, and scaling that count by 2^i. We start at the high

```
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 7: Algorithm to compute mean

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 7 for pseudo-code.

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 and count the PEs with a label greater than G and run CountSelectedPEs again. Depending on whether the count is higher of lower than $C/2$, the the new guess G is either $\frac{H+G}{2}$ or $\frac{L+G}{2}$. H or L are also updated. The algorithm continues for $\log(Max(H) - Min(L))$ iterations.

Histogram within Regions. Run ElectLeader to select an accumulator. For each bin, multicast the value 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.

ParallelPrefix. Among the applications that use parallel prefix are: the enumeration of selected PEs, Radix Sort, parsing regular languages [16], and carry propagation in multi-gauge emulation [2]. See [18, 5] for many more. ParallelPrefix can be run multi-associatively if the regions consist of rectangles; this technique is also extendible to semi-convex regions, that is, regions convex in the horizontal or vertical direction. For regions of arbitrary shape, the multi-associative implementation of parallel prefix is more complex, and the applications

fewer. One application that remains useful, however, is enumeration of PEs, an essential operation for implementing distributed buckets in a multi-associative histogram.

Start by separating the region into horizontal strips with SeparateLines. As in CountPEs, identify the left end PEs of each line and have them multicast their column IDs to the rest of the line. By subtracting this offset, PEs obtain their positions relative to the beginning of the line; we now execute ParallelPrefixLine on these strips. Next, form a coterie of the right end points of the lines with SeparateContours, forming a line or monotonic vertical curve. Execute ParallelPrefixCurve and move these partial results down one PE. Multicast the values back down the rows. Each PE reads the multicast and combines that value with its own to obtain the final result. ParallelPrefix requires at most $2\log d + 3$ arithmetic and communications operations.

Reduction. Reduce combines information from multiple PEs according to some operator, and leaves the result in a single PE. Reduce can be executed by methods analogous to the one dimensional case: run ParallelPrefix and ignore 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.

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 [27]. Two leading methods for finding the convex hull on a serial processor are the Graham Scan [12] and the Jarvis March [19].

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 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 that it makes with its predecessor and its successor. If the angle is reflex, then eliminate that point. The serial Graham Scan is of complexity $O(N \log N)$, which is 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 $\overline{ps_i}$ that makes the smallest angle with respect to the positive X-axis. This process is repeated for all h points on the hull, and therefore the Jarvis March 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 parallel versions of these algorithms are again 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 when the loop has closed. The complexity is therefore $O(h)$, the number of points on the hull, if SelectMin is counted as a unit operation.

6 Conclusions and Future Work

The two keys to the success of a computational framework are for the user to be able to simply and efficiently:

1. create the necessary algorithms for the chosen application, and
2. map the framework onto an existing architecture.

With respect to algorithms, it is never possible to know in advance all of the uses to which a computational framework will be put, but in the present context we have addressed this problem both generally and specifically. Generally, we have broken down the application into classes of problems, whence we derived a set of computational requirements. Those satisfied by multi-associativity include rapid feedback of information from multiple, irregular, data sets. Specifically, we have created numerous multi-associative algorithms that enable the extraction of data over multiple features simultaneously, often through the use of a divide-and-conquer strategy.

With respect to mapping onto an existing architecture, we have presented a direct implementation of multi-associativity on the CAAPP, with the restriction that PE aggregates be contiguous, and that SelectFirst and CountResponders be emulated in software. But most of the features of interest at this level are either contiguous, or can be derived simply through other means (e.g. a Hough transform), while the emulations have only logarithmic complexity, a small price when the number of features is typically in the thousands.

Our work on low-level support for higher level processing continues; a current project includes a parallel implementation of the ISR at the ICAP layer, for which it is likely that new multi-associative algorithms will be needed. We will also be using multi-associativity in other domains: the application to segmentation is also currently under way. Also under consideration are more problems from computational geometry. Another long term project is the mapping of multi-associativity to other SIMD architectures and the evaluation of features with respect to this model.

Acknowledgments

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

References

[1] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M.Lam, O. Menzilcioglu, J.A. Webb (1987): "The Warp Computer: Architecture, Implementation, and Performance," *IEEE Trans. on Computers,* C-36 (12).

[2] F. Annexstein, M. Baumslag, M.C. Herbordt, B. Obrenic, A. Rosenberg, C.C. Weems (1990): "Achieving Multigauge Behavior in Bit-Serial SIMD Architectures via Emulation," *Proc. of the 3rd Symp. on the Frontiers of Massively Parallel Computation.*

[3] K.E. Batcher (1980): "Design of the Massively Parallel Processor," *IEEE Trans. on Computers,* C-29 (9).

[4] J.R. Beveridge, J. Griffith, R.R. Kohler, A.R. Hanson, E.M. Riseman (1989): "Segmenting Images Using Localized Histograms and Region Merging," *Int. J. of Computer Vision,* 2 (3).

[5] G.E. Blelloch (1989): "Scans as Primitive Parallel Operations," *IEEE Trans. on Computers,* C-38 (11).

[6] J. Brolio, B.A. Draper, J.R. Beveridge, A.R. Hanson (1989): "ISR: A Database for Symbolic Processing of Computer Vision," *IEEE Computer,* December.

[7] B.A. Draper, R.T. Collins, J. Brolio, A.R. Hanson, E.M. Riseman (1989): "The Schema System," *Int. J. of Computer Vision,* 2 (3).

[8] M.J.B. Duff (1978): "Review of the CLIP Image Processing System," *Proc. of the National Computing Conference,* AFIPS, pp. 1055-1060.

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

[10] A.D. Falkoff (1962): "Algorithms for Parallel Search Memories," *Journal of the ACM,* 9 (4), pp. 488-511.

[11] C.C. Foster (1976): *Content Addressable Parallel Processors,* Van Nostrand Reinhold Co. New York.

[12] R.L. Graham (1972): "An Efficient Algorithm For Determining the Convex Hull of a

Planar Set," *Information Processing Letters,* 1, pp. 132-133.

[13] A.R. Hanson and E.M. Riseman (1987): "The VISIONS Image Understanding System-1986," in *Advances in Computer Vision,* C. Brown ed., Erlbaum, Hillsdale, N.J.

[14] M.C. Herbordt, C.C. Weems, D.B. Shu (1990): "Routing on the CAAPP," *Proc. of the 10th Int. Conf. on Pattern Recognition.*

[15] M.C. Herbordt, C.C. Weems, J.C. Corbett (1990): "Message Passing Algorithms for a SIMD Torus with Coteries," *Proc. of the 2nd ACM Symp. on Parallel Algorithms and Architectures.*

[16] W.D. Hillis and G.L. Steele Jr. (1986): "Data Parallel Algorithms," *Comm. of the ACM,* 29 (12).

[17] D.J. Hunt (1981): "The ICL DAP and its Application to Image Processing," in *Languages and Architectures for Image Processing,* M.J.B. Duff and S. Levialdi eds., Academic Press, London.

[18] R.M. Karp and V. Ramachandran (1988): "A Survey of Parallel Algorithms for Shared-Memory Machines," *TR 88.408,* U.C.B.

[19] R.A. Jarvis (1973): "On the Identification of the Convex Hull of a Finite Set of Points in the Plane," *Information Processing Letters,* 2, pp. 18-21.

[20] H. Li and M. Maresca (1989): "The Polymorphic-Torus Architecture for Computer Vision," *IEEE Trans. on PAMI,* PAMI-11 (3).

[21] J.J. Little, G.E. Blelloch, T.A. Cass (1989): "Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine," *IEEE Trans. on PAMI,* PAMI-11 (3).

[22] D. Marr (1982): *Vision,* W.H. Freeman, San Francisco, CA.

[23] B.T. McCormick (1963): "The Illinois Pattern Recognition Computer – ILLIAC III," *IEEE Trans. on Elect. Computers,* C-12 (12).

[24] R. Miller, V.K. Prasanna Kumar, D. Reisis, Q.F. Stout (1988): "Meshes With Reconfigurable Buses," *Proc. of the MIT Conf. on Advanced Research in VLSI.*

[25] D. Nassimi and S. Sahni (1980): "Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer," *SIAM Journal of Computing,* 9 (4).

[26] V.K. Prasanna Kumar and D. Reisis (1989): "Image Computations on Meshes with Multiple Broadcast," *IEEE Trans. on PAMI*, PAMI-11 (11).

[27] F.P. Preparata and M.I. Shamos (1985): *Computational Geometry An Introduction*, Springer-Verlag, New York.

[28] D. Rana and C.C. Weems (1990): "A Feedback Concentrator for the Image Understanding Architecture," *Proc. of the Int. Conf. on Application Specific Array Processors*, pp. 579-590.

[29] E.M. Riseman and A.R. Hanson (1989): "Computer Vision Research at the University of Massachusetts-Themes," *Int. J. of Computer Vision*, 2 (3).

[30] A. Rosenfeld (1984): "Image Analysis: Problems, Progress and Prospects," *Pattern Recognition*, 17 (1), pp. 3-12.

[31] C.D. Thompson and H.T. Kung (1977): "Sorting on a Mesh Connected Computer," *Communications of the ACM*, 20 (4).

[32] L.W. Tucker (1988): "Architecture and Applications of the Connection Machine," *IEEE Computer*, August, pp. 26-38.

[33] C.C. Weems (1984): "Image Processing on a Content Addressable Array Parallel Processor," *COINS Tech. Rpt. 84-14 and Ph.D. Dissertation*, University of Massachusetts.

[34] C.C. Weems, E.M. Riseman, A.R. Hanson, A. Rosenfeld (1988): "IU Parallel Processing Benchmark," *Proc. of the Computer Society Conference on Computer Vision and Pattern Recognition.*

[35] C.C. Weems, S.P. Levitan, A.R. Hanson, E.M. Riseman, J.G. Nash, D.B. Shu (1989): "The Image Understanding Architecture," *Int. J. of Computer Vision*, 2 (3).

[36] C.C. Weems, E.M. Riseman, A.R. Hanson, A. Rosenfeld (1991): "An Image Understanding Benchmark for Parallel Computers," *Journal of Parallel and Distributed Computing*, 11 (1).

[37] M. Willebeek-LeMair and A.P. Reeves (1990): "Solving Nonuniform Problems on SIMD Computers: Case Study on Region Growing," *Journal of Parallel and Distributed Computing*, 8.