Associative, Multiassociative, and Hybrid Processing*

Martin C. Herbordt†

Department of Electrical and Computer Engineering
University of Houston, Houston, TX 77204-4793
EMail: herbordt@uh.edu Phone: (713) 743-4434

Charles C. Weems

Department of Computer Science
University of Massachusetts, Amherst, MA 01003
EMail: weems@cs.umass.edu

Abstract: Multiassociative processing is the simultaneous associative processing of sets of elements. In the first part of this article, we define the associative and multiassociative processing models and evaluate their performance with respect to a class of generic, spatially-mapped, functions. When compared with a conventional, realistic, parallel processing model, it is found that multiassociative processing is superior in many cases and associative processing superior in the rest. Further analysis determines that a bybrid technique which combines associative and multiassociative processing is often superior to either one alone. Methods are presented for determining optimal partitions of these computations into associative and multiassociative phases.

In the second part we describe how multiassociativity can be implemented on a reconfigurable broadcast mesh using a technique called *coterie structures*. The primary theoretical result is that techniques similar to those used in PRAM graph contraction algorithms can be used to create an optimal algorithm for implementing a critical multiassociative primitive. We also derive basic properties of coterie structures and present algorithms for basic operations such as information exchange and symmetry breaking within and among structures.

The third part describes an application of the associative models to region segmentation, an important problem in image understanding. Using these techniques, a hybrid associative algorithm is developed that reduces the number of communication instructions by more than an order of magnitude over previous algorithms.

*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 was supported in part by an IBM Fellowship.

1 Introduction

Multiassociativity is an additional level of parallelism: it is a flexible methodology for solving multiple problem instances simultaneously, and each using parallel/associative processing. In its most general form, multiassociativity resembles nested parallelism [4] to depth 2. However, it has the advantage of being based solely on primitives that can be implemented efficiently (using available hardware with some software emulation), while yet being general enough to be useful as both an algorithmic and a programming paradigm.

Multiassociative processing was introduced in [12] and has already found a place in the spatially mapped phases of machine vision computation. The performance is especially good for classes of problems that are characterized by the need for processing non-uniform, but proximate, data sets. It is also likely that multiassociativity will find more applications within machine vision, in other machine perception domains, and in other tasks that use spatially mapped data.

We begin by defining the global and multiassociative paradigms using three different models: 1) in terms of their characteristic primitive operations, 2) by the types of queries they are suited to answer, and 3) through their virtual machine models (instruction sets). We then discuss a class of functions that subsumes many tasks in spatially mapped computation and compare the performance of the associative models with each other as well as with the standard serial and parallel computational models. We find the particular function instances for which the different models have superior performance, when hybrid algorithms are of use, and how this can be determined dynamically.

In the next section, we demonstrate an efficient implementation of multiassociativity on a reconfigurable broadcast mesh [18, 28, 20, 22]. The algorithms presented here use a different approach than those normally used on that network: they are based on the concept of the *coterie structure*, first introduced in [12]. Each data set, or region, is processed using only those PEs to which the pixels of the region have been mapped. Thus, although the underlying model is a reconfigurable mesh, the algorithms operate on arbitrary connected subgraphs of the mesh. The basic method is to use knowledge that PEs can obtain about the network configuration in constant time to dynamically repartition the connected subgraphs into various coterie structures as appropriate for the algorithm. The primary result of this section is that PRAM symmetry breaking and graph contraction techniques can be applied to the coterie structures model to obtain an optimal randomized reduction algorithm (a critical multiassociative primitive) and near-optimal (within $O(\log^* N)$) deterministic parallel prefix and reduction algorithms.

In the final part we describe the application of associative processing to region segmentation, an important problem in image understanding. The critical computation is the characterization of region hypotheses. In previous work, Jenq and Sahni have published an $O(\log N)$ algorithm to compute simultaneously either the area or the perimeter of all components of an image using a reconfigurable mesh [15]. However, although their algorithm is asymptotically optimal, it requires 128 broadcast operations for each of the $\log N$ iterations. We present a hybrid associative reduction algorithm that runs in time equivalent to $7 \log n$ communication operations for a large number of segmentations and with very small variance.

2 Characterizing Associative Models

2.1 Definitions

The prototypical associative memory operation consists of the controller broadcasting a key to the cell array and then performing some action on those cells where a match occurs. The actions typically consist of reading (or writing) some value from (or into) the cell; receiving a response in the form of a SOME/NONE (global OR of the response tag bits) or a COUNT (of tag bits); or selecting a single responder for further processing. For example, the controller may execute the instruction, "How many cells have variable GREEN set to TRUE?" or "All elements with BROWN set to TRUE set SKY to FALSE." The basic associative operations are summarized below (after [9, 27]).

1. Global broadcast/local compare/activity control
2. Some/None of responders from array to controller
3. Count of responders from array to controller
4. Select a single responder

In multiassociative processing, we add the concept of a *set*, which we define as a collection of elements that share some property. Set properties are typically either an *a priori* distinguishing characteristic of the elements, say that GREEN = TRUE, or an attribute of the set itself, say, that it contains over 100 elements. The fundamental principle of multiassociativity is to replace some of the function of the controller with a subset (often a single element) of the members of each set. The basic multiassociative operations are summarized below (after [12]).

1. Within each set: broadcast by a subset/local compare/activity control
2. Within each set: Some/None of responders to a subset
3. Within each set: Count of responders to a subset
4. Within each set: select a single responder
5. Split/Merge sets
6. Transfer data between sets

The first four operations are analogous to their globally associative counterparts, but are executed simultaneously in all selected sets. Of course sets are only useful if elements can added and removed with some efficiency. This is dealt with in the last two operations which together enable dynamic creation of sets and the implementation of divide-and-conquer algorithms.

A consequence of the above definitions is that algorithms applicable to globally associative arrays are also applicable to multiassociative processing, that is, to sets in parallel. For example, the well known associative algorithm SelectSingleResponder [10, 7] can be performed simultaneously for each set using only a single global controller using the following code.

Algorithm SelectSingleResponder
{Beginning with the high-order bit, transmit ID through}
{the Response register. If any PE has a 1 in that bit,}
{turn off activity in PEs with a 0 in that bit.}

```
FOR BitNum := PEIdLength - 1 DOWN TO 0 DO
    Response := PEId[BitNum]
    IF (Response = Some)
        THEN Activity := Response
```

The following is a sample multiassociative instruction sequence:

1. Each set of elements with the same COLOR selects a
   single element to be the accumulator.
2. Each set of elements with the same COLOR send a count of the
   number of elements to their accumulator.

This sequence might be followed by a globally associative routine to obtain a histogram of the largest sets:

While Count (SetAccumulator = TRUE $\wedge$ Count > 500 $= 0$
1. SelectFirst(SetAccumulator = TRUE)
2. Read(Count)
3. Read(Color)
4. Write(FALSE $\rightarrow$ SetAccumulator)

In order to present a more formal description of global and multiassociativity, we define assembly language level machine models for associative and multiassociative memory processors. In both cases, the associative memory cells consist of some number of bits that are not distinguished in terms of function: that is, any cell location can be referenced in any cell operand.

The associative instruction consists of five fields as shown below:

1. OPCODE – Operations are: Compare Read Write Count SelectFirst
2. OPERAND 1 – Controller operand
3. OPERAND 2 – Cell operand
4. ACTIVITY – Indicates cells taking part in the computation
5. RESULT – I/O for some instructions

Field 2 is a value generated by the controller and broadcast to the array. Fields 3-5 are cell locations. As an example, the Compare instruction compares the data broadcast by the controller as indicated by OPERAND 1 with the data given by OPERAND 2 in each cell whose bit pointed to by ACTIVITY is on. If the data match, then the cell location in the RESULT field is set to TRUE.

The multiassociative instruction word consists of seven fields:

1. OPCODE – Operations are: Compare Read Write Count SelectFirst
2. PARTITION – Partitions multiassociative memory
3. OPERAND 1 – Cell operand 1
4. OPERAND 2 – Cell operand 2
5. ACTIVITY 1 – Indicates cells taking part in the computation as operand 1
6. ACTIVITY 2 – Indicates cells taking part in the computation as operand 2
7. RESULT – I/O for some instructions

In this case, all fields besides the OPCODE are cell locations. The PARTITION field divides the memory into sets: cells with identical values are members of the same set. Two activity fields are needed: one to indicate which cells are senders and one for the receivers. As an example, the Compare instruction is now applied in parallel to each set as given by the PARTITION location. Within each set, cells with ACTIVITY 1 turned on distribute the values at location OPERAND 1 to cells with ACTIVITY 2 turned on. If there are multiple writers, then the value

Model	Complexities of Basic Operators		
	Count	Update	Select
Serial	$O(N)$	$O(N)$	$O(\|\text{total number selected}\|)$
Parallel	$O(\log N)$	$O(\log^2 N)$	$O(\log^3 N)$
Associative	$O(\|S\|)$	$O(\|S\|)$	$O(\|S\| \log N)$
Multiassociative	$O(\log N)$	$O(1)$	$O(\log N)$

Table 1: Big-O complexities of basic operators on various computational models. $|S|$ denotes the number of sets into which the array has been partitioned. It is assumed that at least one set has $O(N)$ elements, which is the worst case for the parallel and multiassociative models.

becomes the bitwise logical OR of the send values. The readers compare the received value with their own value at OPERAND 2 and set RESULT according to whether there is a match.

Multiassociative *processing* extends the concept of multiassociative emory in the same way that associative processing extends associative memory. Additional computing capability is added to each cell to enable the generation of symbolic tags to constrain further processing. Rather than forcing an operand to be an existing value, it can be the result of a complex computation. A simple example of associative processing is "All elements with $G > 128$ and $R < 128$ and $B < 128$ set GREEN to TRUE." An example of a more complex multiassociative routine is "All elements with the same color that are contiguous form connected components."

2.2 A Generic Function Template

In order to discuss a real implementation of a subset of multiassociative processing, we restrict our attention to a generic function of the type that frequently arises in spatially mapped vision computation. This function will be used to compare global and multiassociative processing with each other and with the familiar serial and parallel processing paradigms.

Pixels are mapped to PEs. Those that are contiguous and have certain values in common are formed into regions. These regions are then processed using some number each of three basic operators: SelectSubset, Characterize (often a Count of PEs within a component that have a certain property), and UpdateSet. Depending on the task and the processing model, some preprocessing might be helpful to set up communication paths. Examples of tasks that use the generic function template are region-parallel versions of Count, Histogram, FindMedian, FindMean, SelectConvexHull, and many others from computational geometry.

2.3 Performance of Basic Operators

In this section we use big-O notation to make broad comparisons. For the parallel and multiassociative cases, it is assumed that there is at least one set (region) with $O(N)$ elements, a likely occurence in practice. The results are summarized in Table 1.

Serial Processing Model: In the serial model, processing time depends not on the size of the sets, but rather on the number of elements in the entire array that are involved in each computation. SelectSubset requires $O(|\text{total elements selected}|)$ operations, while the other two operators require that all elements be examined and so have $O(N)$ complexity.

Parallel Processing Model: In the parallel model we assume a controller, PE array with N processors, and a

Model	Timimg of Basic Operators		
	Count	Update	Select
Associative	$5\|S\|$ overhead, $2\|S\|$ per Count	$5\|S\|$	$2\|S\|$
Multiassociative	5000	20	20

Table 2: Estimated times in 10's of cycles for basic template function operations on a 256×256 CAAPP. $|S|$ is the number of sets in the array. Multiassociative algorithms represented are data independent.

global OR circuit. The best possible performance for combining routing networks with a non-trivial number of processors is $O(\log N)$. We assume that all sets are processed simultaneously. The best way to execute the generic function is to select a PE per set to be the accumulator. Once this has been done, Count takes one combining communication operation, or $O(\log N)$. A region-parallel Update uses a similar propagation method as a connected components labeling algorithm: it requires $O(\log N)$ communication operations for a total complexity of $O(\log^2 N)$ [19]. Select uses Update $\log N$ times (using the algorithm in [7]) and so has $(\log^3 N)$ complexity.

Global Associative Model: In this model sets are necessarily processed serially. The basic operations have the following complexity *per set*: Select is $O(\log N)$, Count and Update are $O(1)$. If $|S|$ is the number of sets, the total complexities are then $O(|S| \log N)$, $O(|S|)$, and $O(|S|)$ respectively. The complexities are derived in [27].

Multiassociative Model: Here, as in the parallel model, sets are processed simultaneously. The Count and Select operations are each $O(\log N)$ while the update operation is $O(1)$. These values are justified by results in [15, 12].

2.4 Performance of the Generic Function

If we are to assume bounded size arrays, the big-O results of the previous section cannot be taken entirely at face value. But when taken together with empirical results (see Section 4) two general conclusions can be made. The first is that multiassociative processing is faster than parallel processing as long as set characterization operations do not dominate. The second is that global associative processing is superior to both parallel and multiassociative processing when $|S|$ is small, but inferior when $|S|$ is large.

To compare global and multiassociative models in greater detail, we must determine the constants within the big-O and examine their performance as several parameters are varied. These are the proportion of Counts, Updates, and Selects in the generic function instantiation; the input image (upon which the number of sets depends); and the choice of multiassociative Count algorithm. The last choice is significant because there exist several algorithms that are not asymptotically optimal, but which are fast in practice.

The following results give a flavor for the relative performance of the models. Timings for the basic operators on a 256×256 CAAPP are given in Table 2. Graphs giving the behavior of functions with different proportions of operators are given in Figure 2. The multiassociative Count procedure used is a largely data-independent implementation of the general communication operation [12] and thus gives consistent, though suboptimal, performance.

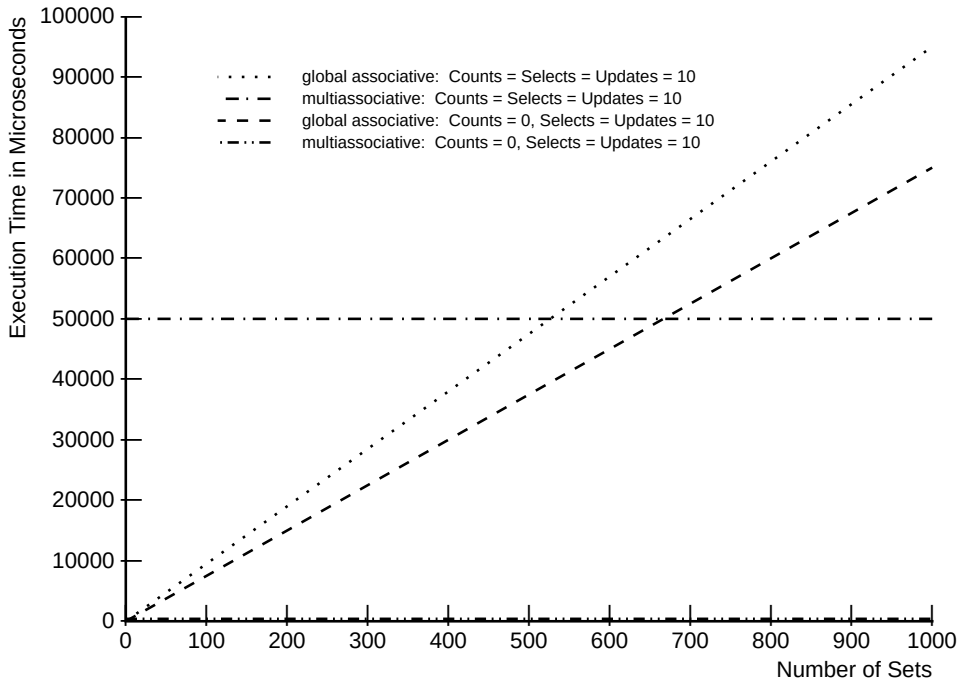

Figure 1: Plot of performance of global and multiassociative instantiations of the generic function versus the number of sets in the image. Times from Table 2 are assumed. For reference, during the early stages of region-merging segmentation algorithms, there are commonly several thousand regions (sets) to be processed.

2.5 Hybrid Associative Algorithms

The generic function we have been examining computes a function F for each of a number of sets s_i in an array. We make the following observations:

- If we apply the resources of the entire array, including those of the controller, to compute F for *any particular set* s_i of S, then a result will almost certainly be obtained more quickly than if F is computed using only those PEs to which s_i is mapped.
- In the case where F is calculated multiassociatively for all s_i of S, there is often a wide variation in elapsed time for the different s_i's. Such a distribution is shown in Figure 2.

From these observations it follows that there may be some function/partition combinations for which a hybrid of a globally associative algorithm A_G and a multiassociative algorithm A_M may be preferable to either by itself.

Algorithm Hybrid-GENERIC
DO an optimal number (O) TIMES
1. Execute an iteration of A_M.

DO UNTIL F is done for all sets
2. Use GlobalSelectSingleResponder to select a set s_i from S, the set of sets for which A_M did not run to completion.
3. Use A_G to process s_i.
4. Remove s_i from S.

We refer to the execution of the first loop as local removal and the second as global removal. To get some intuition as to what O should be and how it can be determined dynamically, we show graphically what happens to the distribution of the remaining sets during local and global removal (see Figure 3). Global removal is roughly equivalent to reducing the area under the graph by one unit per iteration; the graph of the expected new distribution

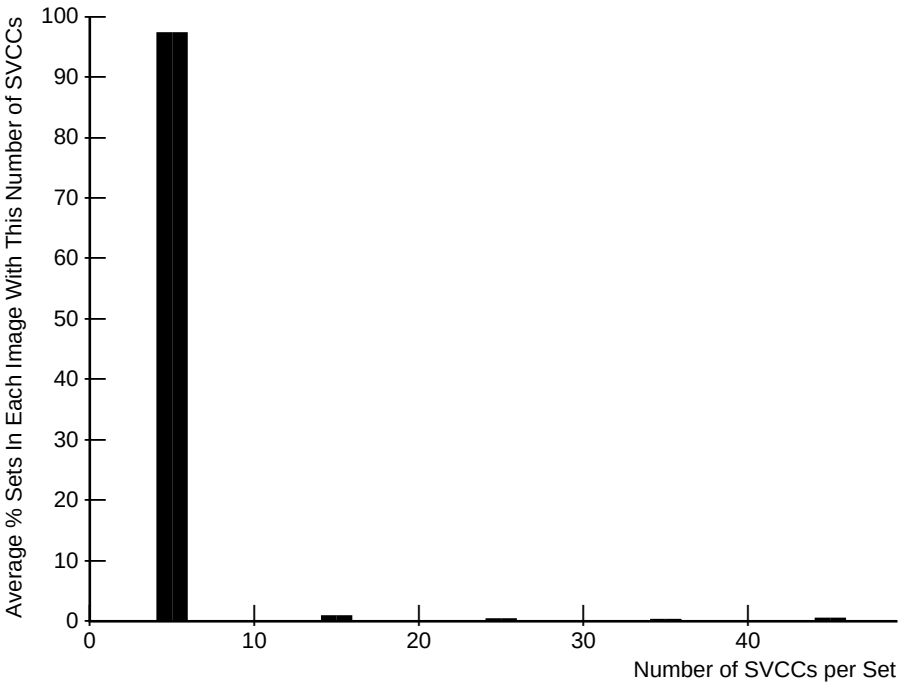

Figure 2: This histogram, taken from work on a multiassociative count algorithm, shows the fraction of sets that fall within given intervals of SVCC counts. The significance is that the performance of the algorithm in a set is proportional to the number of SVCCs in that set. The histogram shows that most sets tend to be well-behaved, but a significant fraction are not.

is generated by reducing the height of the graph at each point by a constant fraction of the height at that point. Local removal is equivalent to moving the Y-axis to the right one unit per iteration.

The optimal value for O minimizes the following expression:

$$K * O \; + \; \int_{O}^{\infty} dist(t)\, dt,$$

where K is the ratio of local to global removal execution times and $dist(t)$ is the distribution of the number of sets which require a certain number of iterations of local removal to run to completion. The integral is the number of sets remaining after the local elimination phase has been completed.

If the distribution is known *a priori*, then O can be found using the following procedure. Find the minimum point t between 0 and T (where T is the maximum non-zero value in the distribution) such that the following condition holds:

$$\forall (t')(t < t' < T) \; \left[\int_{t}^{t'} dist(t)\, dt > K(t' - t) \right].$$

The fact that $dist(t)$ is usually not known *a priori*, plus the complexity of the algorithm, make it impractical for dynamic use, however. But if the distribution decreases monotonically, as has proved to be the case in practice, then the procedure can be simplified substantially: the only t' between t and T that needs to be checked is t itself.

There are two methods for determining O in practice. O can be computed off-line for a data set known to be similar to that being processed. If the variance turns out to be small, as has often proved to be the case, then O can be fixed *a priori*, at least for that domain. Alternatively, we can use the monotonicity assumption: determining whether the local removal phase should be terminated reduces to obtaining $|S|$ after every (perhaps ith) iteration.

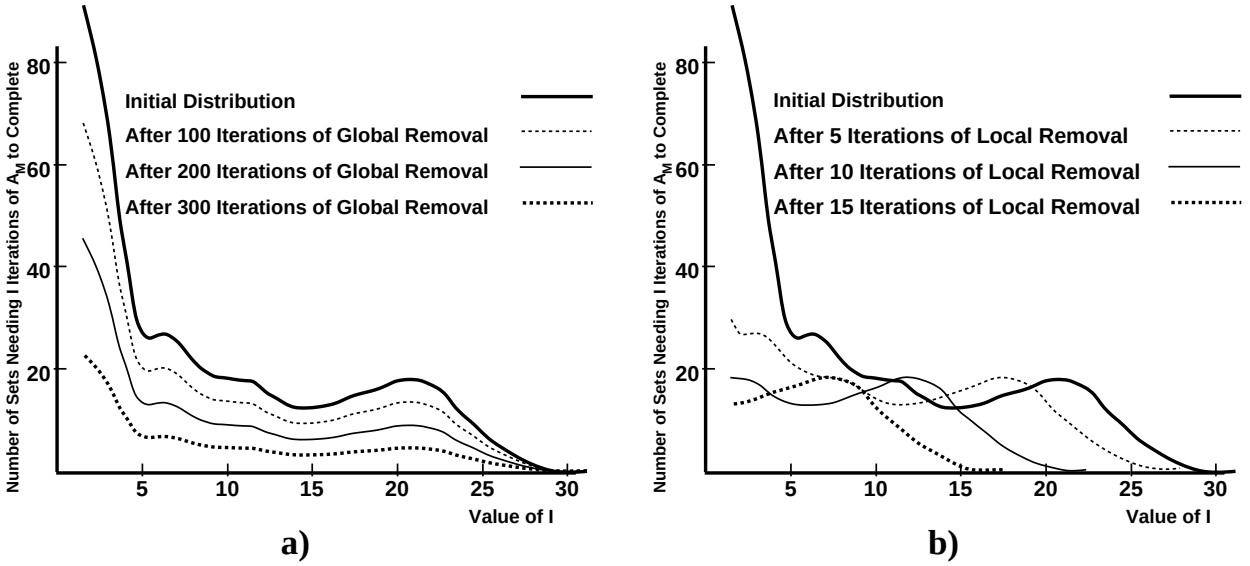

Figure 3: Local and global removal have very different effects on the distribution of the number of sets with certain values of I. I is the number of iterations of A_M left for the processing of a particular set to be completed.

This can be done efficiently by using global count on the remaining set accumulators.

3 Implementation of Multiassociativity

In this section we describe the implementation of a multiassociative capability on the coterie network (also known as the reconfigurable broadcast mesh or RMESH).

3.1 Coterie Network

The coterie network [28] is related to other reconfigurable broadcast networks. Members of this family of networks have two characteristics: they are describable using hypergraphs [1] and they are dynamic. In particular, the *time-t* $n \times n$ *Coterie Network graph* $C_n^{(t)}$ $(t = 0, 1, \ldots)$ has node-set Z_n^2. Each node of $C_n^{(t)}$ is incident to precisely one *coterie-hyperedge*: the coterie-hyperedge incident to node (i, j) of $C_n^{(t)}$ is a subset S of Z_2^n that 1) contains node (i, j) and 2) is *connected* in the sense that the induced subgraph of the $n \times n$ mesh on the set S is a connected graph. Note that at each time t, the coterie-hyperedges partition the node-set Z_n^2 of $C_n^{(t)}$. The coterie-hyperedges of $C_n^{(t)}$ do not depend in any way on the coterie-hyperedges of $C_n^{(t-1)}$.

The coterie network processor array is built on the coterie network graph. Each PE has an input and output port connecting it to precisely one *coterie*, i.e. a (possibly irregularly shaped) bus: PEs (i, j) and (k, l) of the coterie network array are connected at time t to the same coterie (bus) just when nodes (i, j) and (k, l) of the coterie network graph $C_n^{(t)}$ are incident to the same coterie-hyperedge. A coterie is a bus in the sense that, at any time, a single incident PE can "talk" while all other incident PEs "listen." The coterie network has the additional feature that if multiple PEs "talk," the listeners receive the bitwise logical OR of those messages. For a more on hypergraphs and coteries, see [23].

In the physical implementation, each PE controls a set of four switches, N, S, E, W, enabling the creation of coteries. These switches are set by loading the corresponding bits of the mesh control register, which is viewed

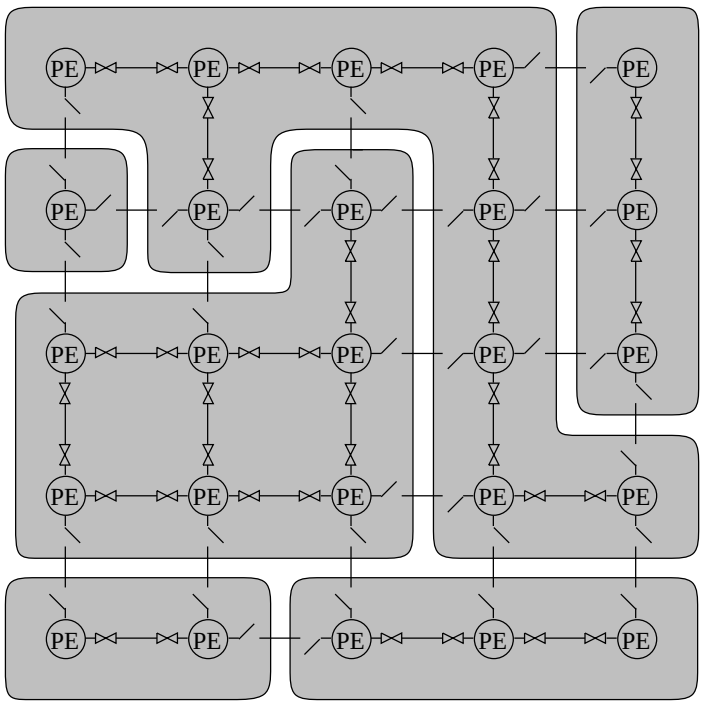

Figure 4: A 5×5 coterie network with switches shown in "arbitrary" settings. Shaded areas denote coteries (the sets of PEs sharing the same circuit).

as local storage within each PE. Thus coterie configurations can be loaded from memory or set from local data-dependent calculations. See Figure 4.

In this article we also assume that nearest neighbor mesh connections are available for local communication. These connections are not strictly necessary, either for algorithmic correctness, or for asymptotic complexity. However, they are useful conceptually and relatively inexpensive from a hardware point of view once the wires of the coterie network are in place.

3.2 Coterie Structures

Advantages of the coterie network are support of one-to-many communication within coteries, reconfigurability of the coteries, and propagation of information over long distances at electrical speeds. Disadvantages are that a circuit can carry only one datum per communication step, and that partitions of the network are constrained by the underlying geometry.

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. To do this efficiently in data dependent algorithms, each PE must 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; and 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: 1) the row and column ID, 2) the tag and the tags of the nearest neighbors, and 3) the OR of some bit of information in a given memory location of members of the coterie.

Note that there are significant differences between the coterie structures model and other models (e.g. the

PRAM and the data parallel model described by Hillis and Steele [13]). In particular, pointer jumping is not a viable algorithmic paradigm as the model does not support unit-time random access operations on non-local memory.

3.2.1 Node Classification

In coterie structure algorithms it is often convenient to classify PEs by whether a PE is an interior or a perimeter point and how the switches are set. A PE has different capabilities in controlling the flow of information through the network depending on its number of connections to its nearest neighbors. We call a PE with 0 connections *unconnected*, 1 an *endpoint*, 2 a *through-point*, 3 a *T-junction*, and 4 a *cross*.

We introduce some other terms that are used below. A closed switch that connects a PE to a coterie (implying that the opposing PE has its switch closed as well) is a *link*. A PE is a *node* if it is taking an active part in a computation, i.e. if it holds data and controls links. It is a *null node* if it takes part in a computation only for algorithmic convenience; i.e. the node itself is only carrying the identity element. And finally, it is sometimes useful for a PE to carry no data, but to control links; we refer to these as *helper nodes*.

3.2.2 Coterie Classification

It is also useful to classify entire coteries into categories (we call *coterie structures*). As is the case with data structures, certain algorithms are either available only on some coterie structures, or have different complexity depending on the structure. As we shall see later, an effective strategy is to partition a coterie that does not meet an algorithmic specification into a number of coteries that do, and then to combine the results. Before describing such algorithms, we first introduce the coterie structures that are used in the rest of this paper.

Coteries have already been defined, see Figure 5d for examples. *Horizontal (or vertical) lines* are defined to be coteries where all the North and South (or East and West) switches are open, see Figures 5e. A coterie is a *chain* if it is comprised of two endpoints joined by an arbitrary number of (0 or more) through-points. A *boundary component* is always defined with respect to a coterie: it is the subset of PEs having the property that at least one of its *eight* nearest neighbors (we include diagonals here) is not a member of the same coterie. Not all connections among those PEs are closed, however: in order for two PEs to have a link, they must be mutually adjacent to at least one PE common from outside the region. Figure 5f contains boundary components derived from the coteries in Figure 5d.

PEs are members of the same *level* of a coterie if they are in the same row (column). PEs in the same level need not be in the same line. *Singly mono-dimensionally connected components* come in two varieties, vertical and horizontal. An SVCC is a coterie where at most one PE in each level has an up-link and one PE has a down-link (the same PE may have both). See Figure 5g for examples.

A *spanning tree* of a coterie contains all PEs in the original structure, but where links have been opened to remove all cycles. See Figure 5h for examples.

A coterie is a *C-graph* if all endpoints, T-junctions, and crosses, but not necessarily through-points are nodes.

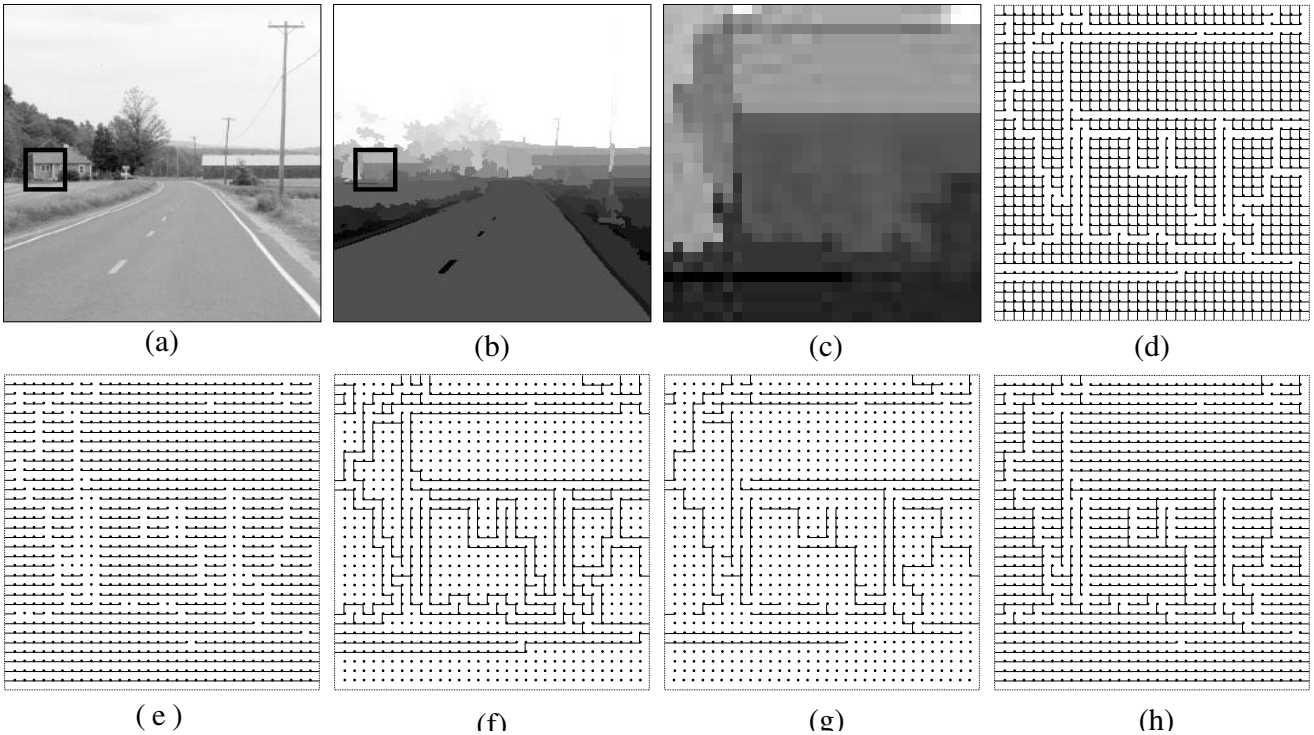

Figure 5: a) Image of a road scene b) segmentation of the image c) 32×32 sub-image of segmentation. (d-h) represent coterie structures derived from sub-image: d) coteries corresponding to regions, e) horizontal lines, f) boundary contours, g) singly vertically connected contours, h) spanning trees.

3.3 Parallel Prefix on Simple Coterie Structures

The literature describing the properties, algorithms, and applications of parallel prefix (also known as scan) is vast; for a sample see [14, 17, 16, 3]. The definition is as follows: Given a set $[x_1, \ldots, x_i]$ of n elements with each element assigned to a different processor and a binary associative operator $*$, compute the n S_i's, where $S_i = x_1 * x_2 * \ldots x_i$, leaving the ith prefix sum in the ith processor. Besides usefulness in its own right, parallel prefix algorithms are usually also the best way of computing reductions, that is, operations identical to parallel prefix but where the partial sums need not be saved. The parallel prefix operation requires $2 \log n$ communication steps on a tree-connected parallel processor, but only $\log n$ are required for a reconfigurable bus. Refer to Figure 6 for an illustration.

The parallel prefix algorithm for an $n \times m$ rectangle is also well known and follows almost immediately from the line algorithm. It has three phases.

Algorithm Parallel Prefix Rectangle
1. Partition the rectangle into m horizontal lines. Run parallel prefix on the lines.
2. Create a vertical line from the m right endpoints. Run parallel prefix on that line.
3. The endpoints update the lines beneath them.

Parallel prefix for a rectangle requires $\log n + \log m + 1$ communication and arithmetic operations.

Somewhat surprisingly, parallel prefix on an SMCC is no more complex and only slightly more complicated than parallel prefix on a rectangle. Again, begin by partitioning the structure into lines and computing parallel prefix. The second phase is slightly more complex because the right endpoints are not neatly aligned and therefore cannot be simply merged into a vertical line. Instead, we have up to three different PEs in each row perform

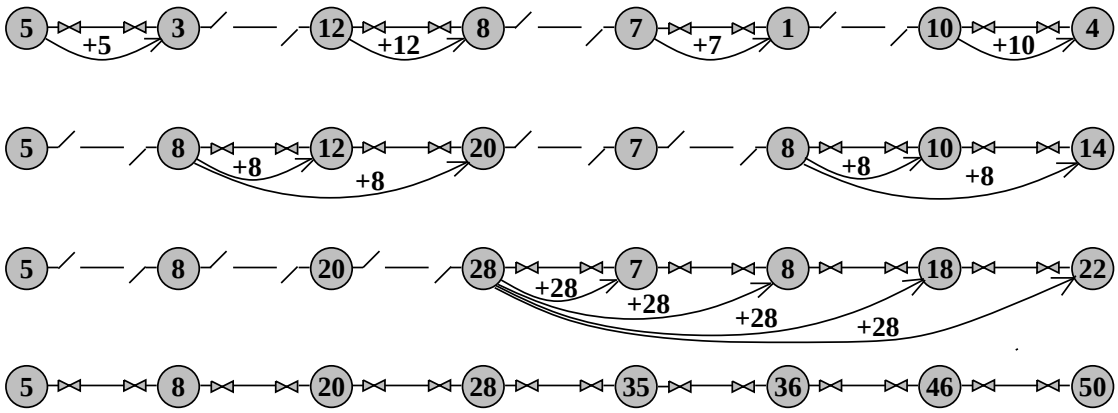

Figure 6: Three iterations and final result of parallel prefix on a line using the + operator. The arrows represent the broadcast operation to elements of the coteries formed when the switches are closed as indicated.

the functions performed by the right endpoint PE in the rectangle algorithm: the right PE endpoint holds and processes the information as before, but two possibly different PEs control the links to the rows above and below respectively (up- and down-links). Recall from the definition of SMCC that (except for the top and bottom rows), exactly one PE in each row controls the up-link and one PE controls the down-link. These PEs recognize this fact in unit time by looking at their mesh control registers. See Figure 7 for an SMCC with those PEs labelled.

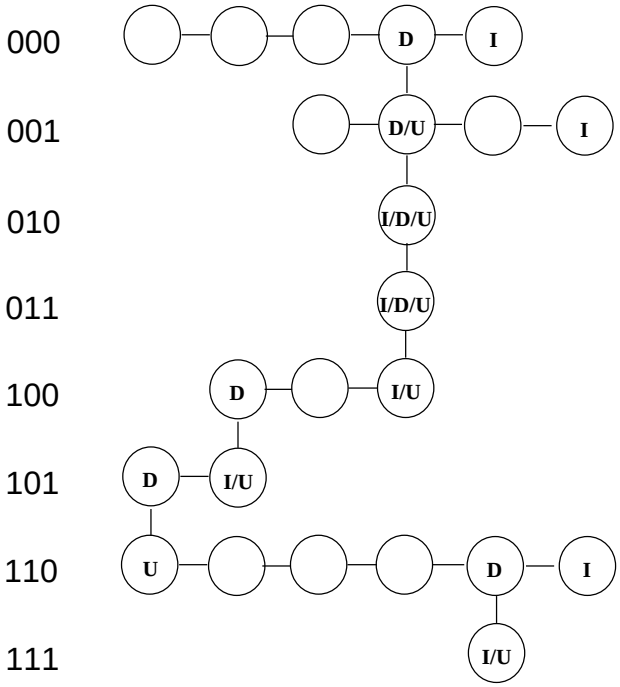

Figure 7: A singly vertically connected component (SVCC) with information-carrying, up- and down-link PEs indicated.

3.4 Basic Coterie Structure Algorithms

The algorithms presented here will be used as primitives later. The complexity is assumed to be $O(1)$ unless otherwise indicated.

3.4.1 Communication and Symmetry Breaking

Task: Transfer of data between two adjacent coteries.

Algorithm: Assume that a protocol has been decided upon between two neighbors. The PEs that are on the mutual borders of the communicating coteries close the switches toward each other. With a circuit comprising both coteries thus formed, the PEs of the sending coterie broadcast while the PEs of the receiving coterie listen. See Figure 8.

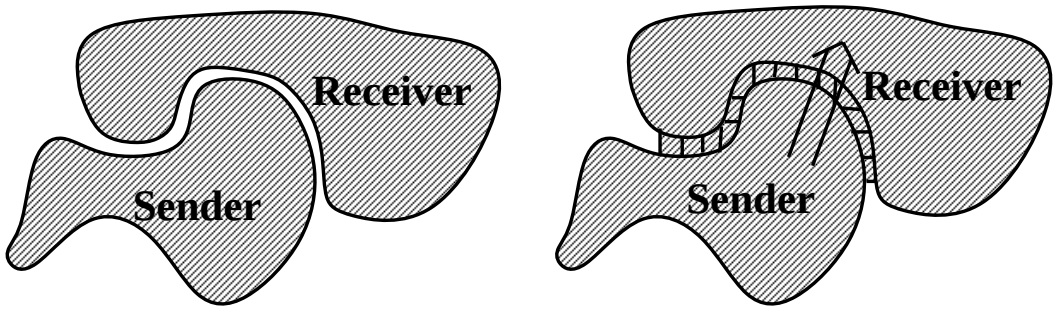

Figure 8: Coteries close mutual switches to communicate

Task: Symmetry breaking (establishing precendence) between a pair of nodes in a coterie.

Randomized Algorithm: The nodes simultaneously broadcast a sequence of independently generated random bits and their complements until the bit one node broadcasts is the complement of the bit the other node broadcast. Nodes can tell this has occurred by comparing the broadcast results (the two ORs of the two pairs of signals) with the bits they just broadcast. The node that generated the 1 is designated the sender and the other the receiver.

Deterministic Algorithm: Nodes broadcast their IDs and $\overline{ID}$s. After reading the two bitwise ORs and comparing them with the values they broadcast, each obtains its opposite's ID. The node with the higher ID becomes the sender and the other the receiver.

Task: Two nodes within a coterie exchange information.

Algorithm: Precedence is established by using one of the symmetry breaking algorithms immediately above. The task is completed with a pair of broadcasts.

Task: Each node in a C-graph exchanges information with all its neighbors.

Deterministic Algorithm: Recall that nodes are separated by chains of null nodes of arbitrary length and that therefore, all nodes have links either to other nodes, or to chains of PEs that have a node at the other end. The basic idea is to use the neighbor connections to break the symmetry; otherwise it would be impossible to tell deterministically which pair of nodes should communicate when. The cases where the length of the intermediate chain is 0, 1, and > 1 are handled separately (see Figure 9). In all three cases, the algorithm starts with the nodes using the nearest neighbor connections to transfer copies of their information to all the PEs toward which a link is established.

Case 1: Chain length $= 0$. The nearest neighbor move has already completed the transfer.

Case 2: Chain length $= 1$. The intermediate PE swaps the data from the two adjacent nodes and transfers them to

their destinations.

Case 3: Chain length > 1. The two node swap algorithm is used to transfer the data between the two endpoints of the chain. Neighbor transfers complete the exhange.

Randomized Algorithm: We use a variation of the randomized symmetry breaking procedure presented above. It is assumed that nodes have a list of links to their neighboring nodes. Nodes randomly close switches in the direction of one of the links, opening those in all the other directions. Nodes then use one of the symmetry breaking procedures to check whether the opposite node has done the same. The nodes that did not form a correspondence with a neighbor again randomly select a link for another try. This continues for a constant number of iterations during which a constant fraction of the nodes in the C-graph will have established correspondence. Corresponding nodes exchange information and temporarily remove each other from their adjacency lists. This procedure continues until all nodes have completed the information exchange with all their neighbors.

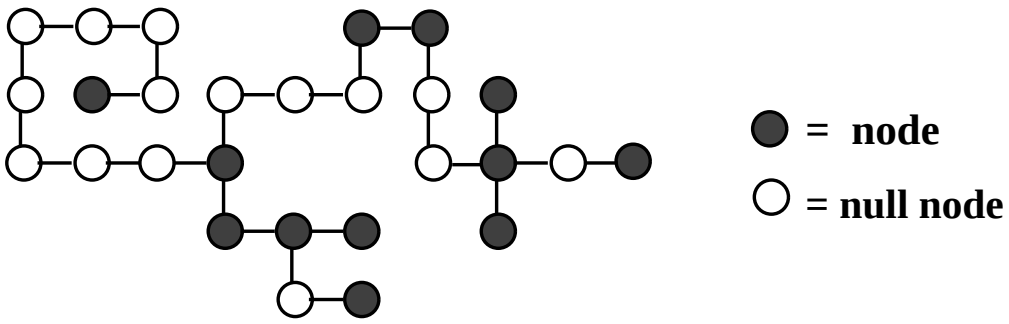

Figure 9: C-graph with nodes separated by chains of null nodes of varying length.

Task: $O(1)$ coloring a chain.

Algorithm: When using the standard PRAM $O(1)$ coloring algorithm [11] for constant degree graphs, the number of colors is small when operating on an ordered structure, say a list or rooted tree (see e.g. [6]), but exponential in the degree of the graph when not. The basic idea then, is to use information available locally to nodes on the chain to create such an ordering. Nodes obtain their neighbors' IDs which are compared with their own. Nodes label themselves $\nwarrow\!\!\nearrow$, $\nearrow\!\!\nwarrow$, $\nwarrow$, or $\nearrow$, depending on whether the ID "slope" is up or down in each direction. Nodes exchange slope labels with their neighbors. The neighbor on each link can have one of two possible labels: either the slope continues in the current direction (e.g. $\nearrow$ goes to $\nearrow$), or it can go to a local maximum or minimum (e.g. $\nearrow$ goes to $\nearrow\!\!\nwarrow$). Nodes that are $\nwarrow\!\!\nearrow$ and $\nearrow\!\!\nwarrow$ cannot be adjacent to other nodes that have that same slope label and so simply color themselves $\nwarrow\!\!\nearrow$ and $\nearrow\!\!\nwarrow$. The remaining nodes form monotonic ID sequences. These sequences, which now *have* direction information, can be colored using the previously mentioned 6-color algorithm and then spliced together with the $\nwarrow\!\!\nearrow$ and $\nearrow\!\!\nwarrow$ nodes to form an 8-colored chain. To reduce the number of colors to 6, the $\nwarrow\!\!\nearrow$ and $\nearrow\!\!\nwarrow$ nodes examine their neighbors' colors and choose any of those remaining to be their own. The 6-color PRAM list algorithm has $O(\log^* N)$ complexity, as does the 6-color coterie chain algorithm.

Task: Create Maximal Independent Set of nodes in a C-graph.

Deterministic Algorithm: Use the information exchange algorithm above to implement the $O(\log^* N)$ complexity MIS algorithm found in [11].

3.4.2 Minimum Partition of Coteries Into SVCCs

In this section we prove the somewhat surprising result that a minimum partition of a coterie into SVCCs can be constructed in constant time.[1] Besides the inherent usefulness of this result (see section 5), the significance is that the algorithm performs a complex function using only information that can be determined locally by each PE in constant time.

A requirement of an O(1) partitioning algorithm is to be able to determine (as yet unspecified) properties independently of distances. We have such a tool in the F-G algorithm (see below). In particular the following question can be answered in constant time: "For any PE P with property F on a line, is there a PE Q with property G between P and either the next PE with property F (to either the left or right) or the end of the line?" We define such a PE Q to be *adjacent* to PE P. If two adjacent PEs become members of the same coterie by closing switches towards each other and opening switches away, then those PEs are said to be *merged*.

Algorithm F-G
{For all PEs with property F, find out whether there}
{is a PE with property G between it and the next PE}
{to the left with property F, or the end of the line.}
SaveCoterieSettings()
Coterie[N,S] := OPEN
If (Tag = F) Coterie[E] := Open
If (Tag = G) Broadcast(TRUE)
If (Tag = F) G-AdjacentLeft := CoterieInput
RestoreCoterieSettings()

We start by observing that a greedy algorithm produces the desired partition: starting at the top level, draw circles around each down-link. Then level by level, expand each circle to include links at the lower levels while not letting the circles intersect. See Figure 10 for an example.

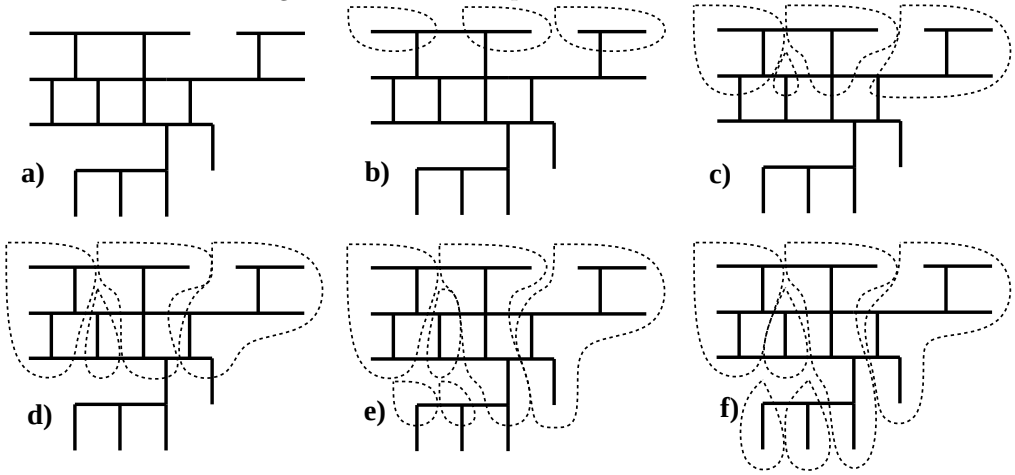

Figure 10: A coterie can be partitioned into a minimum set of SVCCs through a serial level-by-level growth algorithm.

Lemma 1 *The cardinality of the minimal set of SVCCs $|S|$ in a coterie is equal to the number of SVCCs that must be added with the addition of each new level.*

Proof: Assume a coterie partition contains fewer SVCCs than the number that need to be added at every level. Then one of the levels of the coterie would be partitioned into fewer SVCCs than by the above method. But

[1] A simple algorithm for a non-minimum, but useful, partition is presented in [12].

that would either leave vertical links unaccounted for, or an SVCC with multiple vertical links. Both of these are contradictions.

Lemma 2 *To create a minimal set of SVCCs within a coterie, it suffices to create a maximal matching between adjacent up- and down-links and merge them together.*

Proof: The maximal matching leaves the fewest possible leftover vertical links, requiring that the fewest possible new SVCCs need to be created at that level.

We now present the SVCC construction algorithm. First label the PEs with up- and down-links as U and D. All other PEs are ignored. Next, partition the coterie into lines. Within each line, create a maximal matching between adjacent U and D PEs, a sufficient condition for creating a minimal partition of a coterie into SVCCs. This is a multi-step process:

1. Each U (D) determines whether there are adjacent D (U) PEs to the left and/or right using the F-G algorithm.

2. Each U (D) PE labels itself with a 0, 1, or 2 depending on the number of adjacent D (U) PEs. We call those numbers the *incidences* of each PE.

3. The U (D) PEs send the number of incidences to their adjacent D (U) PEs, if any.

4. Each PE is labeled with an ordered triple composed of the number of incidences of the left adjacent PE, its own incidences, and the incidences of the right adjacent PE.

5. The value of the ordered triple is sufficient to determine whether each PE should open or close its East and West switches to create the optimal partition for all but the (2,2,2) case. PEs with (-,0,-) do nothing, with (*,1,*) merge with their one adjacent PE, with (1,2,1) arbitrarily merge with either left of right adjacent PE, and with (2,2,1) or (1,2,2) merge with the adjacent PE with one incidence.

6. PEs with (2,2,2) merge with adjacent PEs according to the procedure described below.

See Figure 11 for an illustration of the labeling of a single line. Before presenting the (2,2,2) case, we show the correctness of what we have described so far.

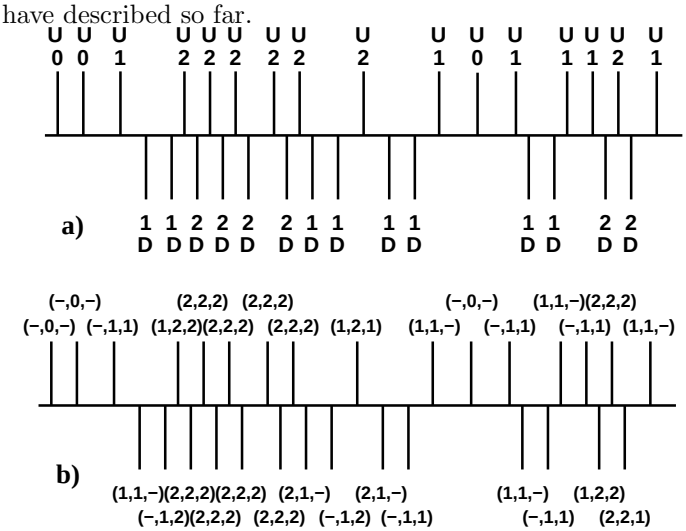

Figure 11: A line of one level of a coterie with a) up-links, down-links, and incidences labeled, and b) ordered triples of incidences of own and neighboring PEs.

Lemma 3 *A U(D) PE can only form an SVCC with a single, adjacent, D (U) PE.*

Proof: Assume not. Then either two or more D PEs must be in the SVCC at that level. If the D PE is not adjacent, the line must be open circuited. Both are contradictions.

Lemma 4 *The U-D merges effected by the SVCC construction algorithm are correct for the cases shown in step 5 (not the (2,2,2) case).*

Proof: From Lemma 4 we know we need only insure that each PE merges with the correct, adjacent PE. Cases (-,0,-), (*,1,*), and (1,2,1) are trivial: in the first nothing can be done, in the second there is no choice, and in the third the choice does not matter. In the cases (2,2,1) and (1,2,2), the PE should merge with the PE with one incidence: It is possible for the PE with two incidences to find another PE with which to merge, while the PE with one incidence has only this opportunity.

The (2,2,2) case is critical: if the merger is not done correctly, the complexity of our parallel prefix algorithm increases from $O(\log N + |S|)$ to $O(N)$. We we formalize this fact in the following lemma.

Lemma 5 *If a PE P with label (2,2,2) is not guarenteed to merge with the correct PE, then it is possible for an adversary to force the creation of an SVCC partition where $|S|$ is $O(N)$ greater than optimal.*

Proof: As previously, we are only concerned with U and D PEs. The neighborhood of the line around P necessarily has the following form: a PE with one incidence, followed by a series of PEs with two incidences, followed by a PE with one incidence (1,2,...,2,1). The series of 2 incidence PEs can have either an even or an odd number elements. If odd, and all the (2,2,2) PEs merge arbitrarily with either their right or left neighbors, then exactly one PE is left over, the best possible result. In the even case, however, such a uniform merging strategy leads to there being either zero or two left-over PEs. If no method existed to make the correct decision, it would thus be possible for an adversary to create $O(N)$ strings of constant length which after partitioning would each leave 2 unmatched links.

We now solve the (2,2,2) case. From Lemma 6 we know that we only need to worry about the case where the series of PEs with 2 incidences is of even length. There are two possibilities: the (2,2,1) PE at the right end of the $2, \ldots, 2$ series is either a U or a D; the (1,2,2) PE at the left end is always the opposite. In the first case, all (2,2,2) PEs that are U should merge left, in the second case they should merge right.

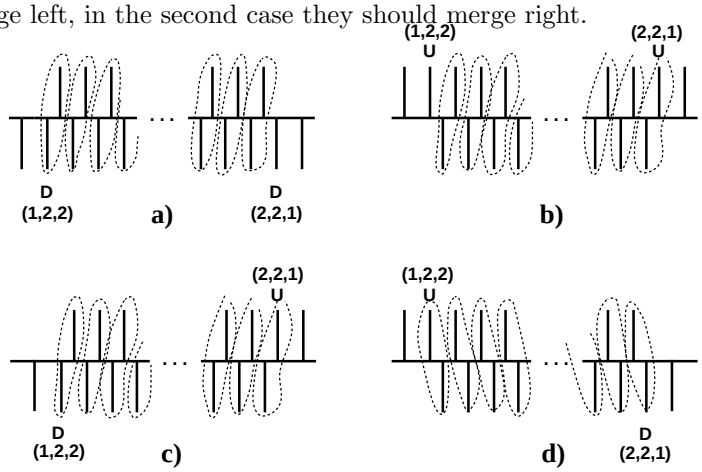

Figure 12: Four cases of merging adjacent PEs within strings (1,2,2,2,1). In a) and b), there are an odd number of (2,2,2) PEs and either the (1,2,2) or the (2,2,1) PE always remains unmerged. In c) and d), there are an even number of (2,2,2) PEs. The merges must all go in the correct direction or both (1,2,2) and (2,2,1) PEs will remain unmerged.

A procedure for handling the (2,2,2) case follows. PEs that are (2,2,1) open their East switches and PEs that

are (1,2,2) open their West switches to form lines with the pattern $(1, 2, \ldots, 2, 1)$. The PE that are (2,2,1) and U broadcast a one, those that are (2,2,1) and D broacast a zero. The procedure is repeated for the (1,2,2) PEs. The (2,2,2) PEs read these values and have four cases: (0,0), (0,1), (1,0), and (1,1). In the (0,0) and (1,1) cases, there are an odd number of U (D) PEs: these can thus merge arbitrarily (but uniformly) with their left or right (right or left) adjacent D (U) PE. In the (0,1) case the D (U) PEs merge right (left) and in the (1,0) case the D (U) PEs merge left (right). See Figure 12 for an illustration of the four cases with the merges done correctly.

Lemma 6 *The U-D merges effected by the SVCC construction algorithm are correct for the cases where all elements of the triple are 2's.*

Proof: Follows from previous two paragraphs.

Theorem 1 *The SVCC construction algorithm partitions any coterie into a minimal set of SVCCs in constant time.*

Proof: That the time is constant is immediate: there are no loops and no dependencies on the number of PEs. The correctness follows from the preceding lemmas.

3.5 Algorithms Based on Graph Contraction

In this section we again show how techniques used in other models can be applied to coterie structures. In particular, the communication and symmetry breaking algorithms of the previous section are integrated with PRAM contraction techniques (see [21, 24]) to produce an $O(\log N)$ randomized reduction algorithm and $O(\log^* N \log N)$ deterministic parallel prefix and reduction algorithms. Since the prefix algorithm follows almost immediately from the reduction algorithm, only the latter is presented.

Algorithm REDUCE
While |nodes| > 1 **do in parallel**
    Use MERGE to merge legal pairs of adjacent nodes
    such that the resulting vertices have degree $\leq d_{max}$.

Phillips shows that, for bounded degree planar graphs, there are always a constant fraction of nodes eligible for merger as long as $d_{max} \geq$ the degree of the graph [24]. That a constant fraction of nodes eligible for merger does so follows by arguments similar to those used in the previous section. The **While** loop thus executes $O(\log N)$ times.

The MERGE procedure is more complicated when applied to coterie structures than the equivalent PRAM algorithm. The reason is that in a distributed memory model, nodes cannot simply be eliminated once their data have been combined: the wires associated with the underlying PEs may still be needed to transmit information between the nodes remaining in the computation. Since these leftover PEs always through-PEs, MERGE operates on a C-graph.

The constraint that no node ever have degree > 4 creates a relatively small constant number of merge cases that can be handled separately. This number is decreased by imposing the convention that the node with smaller degree always transfer its data to the node with larger degree. We examine five cases (see Figure 13), the others are analogous.

1. An arbitrary node merges with a node with degree 1 or 2. If the lower degree node is d_1, the higher degree node eliminates the link to that node. If it is d_2, then the links remain intact, although that node is removed.
2. A d_3 node merges with another d_3 node with which it has one common neighbor. One node is selected to send data, the other to receive. The receiver retains its link to the common neighbor node while the sender removes it.
3. A d_3 node merges with another d_3 node with which it has no common neighbors. The resulting node has degree 4, but its functionality must be distributed over the two original nodes. One node holds the data and controls two links, while the other, which has now become a helper node, controls the other two links.
4. A d_4 node with a helper node merges with a d_3 node with which it has one common neighbor. The d_3 node is turned off and the link from the d_3 node to the common neighbor is eliminated.
5. Two d_4 nodes with helpers and two common neighbors merge. One of the nodes (and its helper) are turned off and its links to the common neighbors are removed.

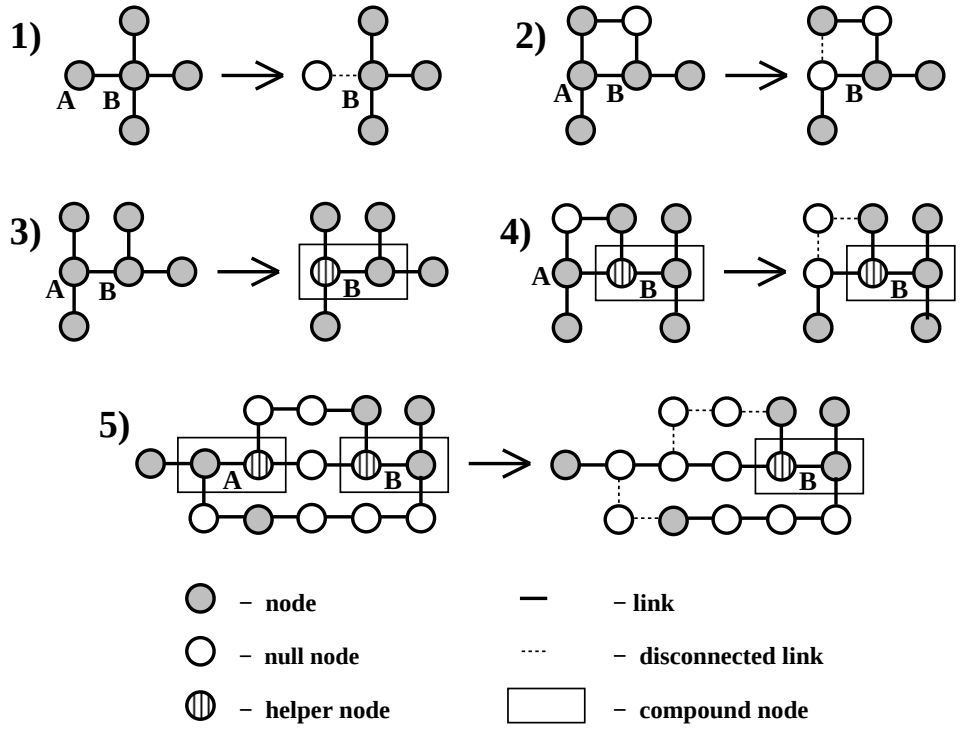

Figure 13: Five cases of nodes merging on C-graphs. In all cases, **A** merges into **B**.

For this list to be exhaustive (including analogous cases), it is necessary for the following propositions to be true:

Proposition 1: Only nodes of degree 4 can ever need a helper PE.

Proposition 2: No node ever needs more than one helper PE.

The truth of these propositions follows from the fact that for nodes that are created using the above procedures, the only possible configuration where a helper node can be introduced is in case 3. That is because compound nodes (nodes with helpers) can only be created when the node formed through merger has greater degree than either of the two constituents. We observe that after a merger where the degree does not increase—between any node and a

d_2, a d_3 with one common neighbor, or a d_4 with two common neighbors—the merged node is a through-PE with no need to control information flow.

Procedure MERGE
$\forall$ nodes **do in parallel**
1. Find adjacent nodes and create a neighbor list.
2. Exhange neighbor lists with all neighbors.
3. $\forall$ neighbor lists **do sequentially**
   Eliminate duplicates, determine neighbors with which merger is legal.
4. Execute either the random or deterministic version of SELECT-MATCH-PAIRS.
5. Combine nodes according to the cases presented above.

Since all the nodes are guaranteed to have bounded degree, steps 1, 2, 3, and 5 are $O(1)$.

Procedure DETERMINISTIC-SELECT-MATCH-PAIRS
1. Run the COTERIE-MIS algorithm from section 3.1.
2. Members of the MIS that are members of a legal merge pair choose a merge partner. The partner breaks ties if it is multiply selected.

The complexity of COTERIE-MIS is $O(\log^* N)$ which is also the complexity of the algorithm.

Procedure RANDOMIZED-SELECT-MATCH-PAIRS
While nodes in match pairs are unmatched and have an unmatched partner
   Nodes in match pairs randomly select a match partner.
   If the nodes agree, they declare themselves matched.

Since nodes have a constant probability of being matched during every iteration, the expected time of the algorithm is $O(1)$. Therefore, step 4 of MERGE is $O(1)$ if the randomized algorithm match select algorithm is used, and $O(\log^* N)$ in the deterministic case. Thus the overall REDUCE complexities are $O(\log N)$ and $O(\log^* N \log N)$ respectively.

The complexity constant improves drastically if the coterie is known to be a tree. The algorithm runs as follows:

Algorithm TREE-REDUCE
While |nodes| > 1 **do in parallel**
   Use TREE-MERGE to merge legal pairs of adjacent nodes
   where legal pairs have at least one node with degree 1 or 2.

Since trees have the property that at least half the nodes have degree 1 or 2, the **While** loop executes $O(\log N)$ times. The fraction of nodes available for merger in the general merge algorithm is likely to be substantially smaller.

Procedure TREE-MERGE
$\forall$ nodes **do in parallel**
1. Find adjacent nodes and compute degree.
2. Exhange degree information with all neighbors.
3. Execute either the random or deterministic version of TREE-SELECT-MATCH-PAIRS.
5. Combine nodes according to case 1 presented above.

The tree version of randomized match pair selection is identical to the non-tree version. In the deterministic case, however, we only need to run chain MIS, rather than the coterie version.

Procedure TREE-DETERMINISTIC-SELECT-MATCH-PAIRS
1. Nodes adjacent to d_1 nodes select one of them for merger.
2. Nodes with degree 3 or 4 adjacent to d_2 nodes choose one for merger iwth the d_2 node breaking ties.
3. Form chains of the contiguous remaining d_2 nodes. Members of the MIS choose merger partners, with chosen nodes breaking ties.

The advantages of TREE-MERGE are that no neighbor lists need to be created, exchanged, or searched for duplicates; that symmetry breaking is much easier on a chain than on a bounded degree planar graph; and that only case 1 of the actual node combining needs to be executed. However, since we are not aware of any algorithm to create a spanning tree out of arbitrary coteries that is itself not $O(\log N)$, TREE-REDUCE will likely only be useful if multiple reductions on a coterie partition are to be computed.

4 Practical Algorithms and Experimental Results

In this section we investigate the performance of reduction algorithms on tasks arising during the execution of a working image segmentation system. This system has been used extensively in image interpretation research in various image domains. We find that a hybrid global/multiassociative technique yields the best performance in practice.

4.1 A Segmentation System

Many segmentation systems use a region merging paradigm based on the extraction of local region properties and the constraint of global criteria [5, 8, 26, 2]. Region merging begins by characterizing each region by the means and standard deviations of various spectral quantities, by its size, and by the lengths of its common borders with adjacent regions. Based on these values, merge scores are calculated with respect to the adjacent regions. Region pairs are merged if their merge score is both a local maximum and surpasses a global threshold. The process is repeated until no merge scores surpass the global threshold.

The bulk of the computation occurs during the original region characterization; thereafter, characterizations are computed from the attributes of the constituent regions. The critical problem is therefore the efficient reduction of the regions of the oversegmented image.

The data set consists of 28 256×256 intensity images of which 13 are road scenes and 15 house scenes. Figure 14 shows a sample image from the road scene domain, the phase 1 output (oversegmented image), and the final segmentation. See [2] for more examples. The oversegmented images have an average of 1900 regions with a standard deviation of 788. See Figure 15 for details.

4.2 Multiassociative Reduction Algorithms

We now present an SVCC-based multiassociative reduction algorithm for arbitrary coteries that is the basis for our practical algorithm. The parallel-prefix algorithm follows as before with little added complexity.

Algorithm Local REDUCE
1. Partition the coterie into horizontal lines.
2. Reduce lines and leave the result in the right endpoint.
3. Partition the coterie into SVCCs.

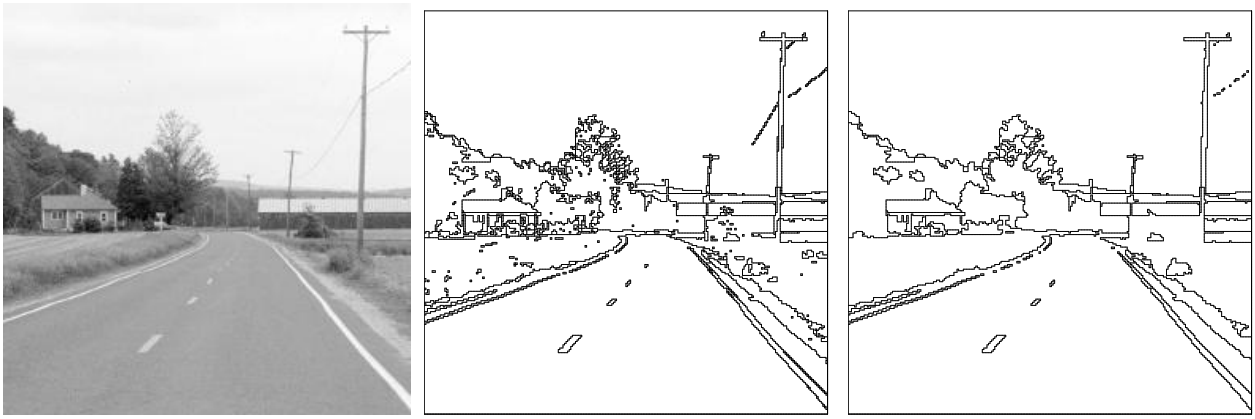

Figure 14: a) Sample image is Road Scene 16, b) over-segmentation of image, c) image after region merge.

4. Reduce SVCCs leaving the result in the bottom endpoint. The bottom endpoints of the SVCCs in each region form a set S with cardinality $|S|$.

DO UNTIL $|S| = 0$ for all coteries

6. Use SelectSingleResponder to select an element s from S.
7. s broadcasts its partial sum to the elements in S which combine that value with their own.
8. s removes itself from S.

Steps 1 and 3 are constant time operations, while 2 and 4 require $\log n$ communication and arithmetic operations.[2] In the DO loop (a phase we refer to a *local removal*), Step 6 uses an $O(\log N)$ algorithm (although with a small constant), while Steps 7 and 8 are again constant time operations. The overall complexity is therefore proportional to $\log N$ and depends on the number of SVCCs $|S|$ into which the worst behaved coterie has in the image been partitioned. Since it is relatively easy to construct a coterie where $|S| = O(N)$, we are left with an $O(N \log N)$ algorithm.

Since we are invesigating practical algorithms in this section, a quantity more important than the worst case of $|S|$ is its size during real applications. Results are presented in Figure 15. In summary, it is common for at least one region per image to be so badly behaved as to make the above algorithm impractical: the DO loop would have to be executed an average of 153 times and a value of 300 would not be unlikely. However, the number of badly behaved regions per image is small (with an average of only 13 regions per image having more than 10 SVCCs), signaling the possibility of large performance gains with a relatively small amount of additional work.

Image Type	avg number of regions	avg number of SVCCs	avg SVCCs per reg.	max SVCCs per reg.	avg count of regions w/ this # SVCCs				
					1-10	11-20	21-30	31-40	41+
road	1824.6	2871.5	1.60	216.5	1810.2	7.1	2.6	1.5	3.3
house	1965.3	2717.5	1.38	98.3	1952.8	6.7	2.5	0.9	2.4
avg all	1900.0	2789.0	1.48	153.1	1887.6	6.9	2.5	1.2	2.8

Figure 15: Image statistics for the data set after completion of the oversegmentation phase. For reference, the average number of regions per image is substantially greater than n, the average maximum number of SVCCs per region is only slightly less than n, and the average number of regions per image with more than 10 SVCCs is 13.4.

[2]For the rest of this article, the algorithm used in step 3 is assumed to be the one in [12].

One possibility is to add one or more SMCC reduction steps, i.e. to repeat steps 3 and 4 alternating between the vertical and the horizontal dimension. After all, if SVCC reduction of endpoints eliminates a large fraction of the horizontal line accumulators, it seems likely that SHCC reduction of SVCC accumulators could also be helpful.

Algorithm SMCC REDUCE
1. Partition the coterie into horizontal lines.
2. Reduce lines and leave the result in the right endpoint.
DO I TIMES
    3. Partition the coterie into SVCCs.
    4. Reduce the SVCCs leaving the result in the bottom endpoint.
    5. Partition the coterie into SHCCs.
    6. Reduce the SHCCs leaving the result in the right-most endpoint.
DO UNTIL $|S| = 0$ for all coteries
    8. Use SelectSingleResponder to select an element s from S.
    9. s broadcasts its partial sum to the elements in S which combine that value with their own.
    10. s removes itself from S.

The distribution of $|S|$ as a function of I (see Figure 16) shows that the return diminishes rapidly. What happens is that some regions have a large number of SVCC-SHCC pairs that contain each other's accumulators. No progress was ever made after I reached 20.

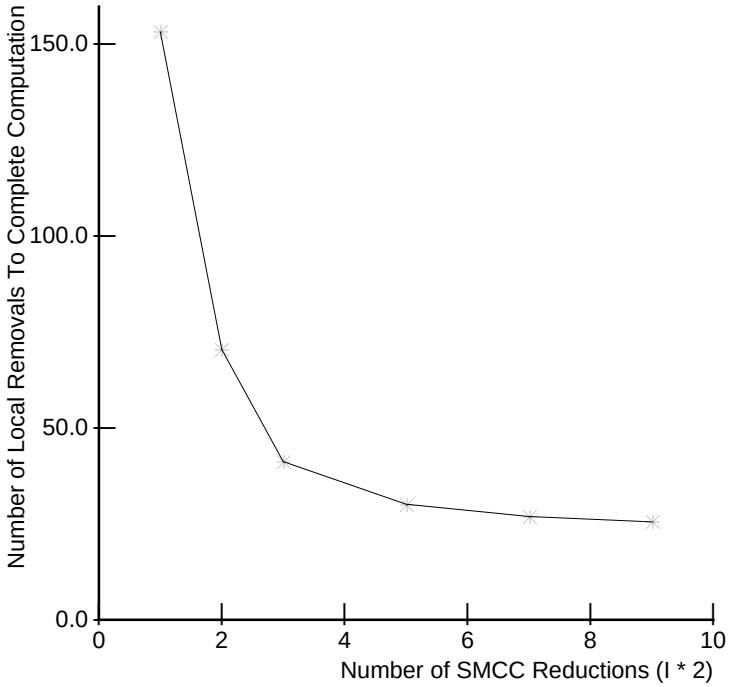

Figure 16: Graph of Max($|S|$) versus I. The reduction in the number of times local removal must be executed levels off after $I = 2$ (number of reductions = 4).

A more promising alternative is the hybrid algorithm in the following section.

4.3 A Hybrid Reduction Algorithm

For any particular region r from the set of regions R, it is usually faster to execute reduction/prefix algorithms by applying the resources of the entire array to that problem instance than it is to use only those PEs to which the

region is mapped. There follows a simple reduction algorithm based on array reduction procedures (which we call *global removal*).

Algorithm Global REDUCE
DO UNTIL $|R| = 0$
1. Use GlobalSelectSingleResponder to select an arbitrary region r from R.
2. Use the resources of the array to reduce r.
3. Remove r from R.

Depending on the operation being used during the reduction, the complexity of step 2 can range from $O(\log N)$ in the general case, down to $O(1)$ when executing the CountSelectedResponders operation with appropriate hardware support [25]. Similarly, step 1 can also have complexity from $O(1)$ using hardware described in [9] to $O(\log N)$ if only a global OR circuit is available. In any case, the algorithm is efficient if and only if $|R|$ is small. We have seen, however, that the number of regions in an oversegmentation can be huge: in the case where regions are relatively small, $|R| = O(N)$.

The idea behind hybrid reduction is to combine the global algorithm with the local removal operation used in the SMCC-based algorithm. After two reduction passes (steps 1-4), an as yet undetermined number of local removals is executed. Global removals are then used to finish the reduction.

Algorithm Hybrid-REDUCE
1. Partition the coterie into horizontal lines.
2. Reduce lines and leave the result in the right endpoint.
3. Partition the coterie into SVCCs.
4. Reduce SVCCs leaving the result in the bottom endpoint. The bottom endpoints of the SVCCs in each region form a set S with cardinality $|S|$.

DO an optimal number (O) TIMES
6. Use SelectSingleResponder to select an element e from S.
7. e broadcasts its partial sum to the elements in S which combine that value with their own.
8. e removes itself from S.

DO UNTIL $|R| = 0$
9. Use GlobalSelectSingleResponder to select a region r from R.
10. Use a global reduction algorithm to reduce r.
11. Host broadcasts result back to r.
12. r removes itself from R.

Recall from Section 2.5 and Figure 3 the rationale behind hybrid algorithms: The first loop moves the curve to the left until only the long, flat tail remains; the second loop processes the tail in a number of steps equal to its maximum height. The optimal number of iterations for the first loop (O) can be approximated using either of the two methods previously described. In particular, we see in Figure 17 that for each ratio of global to local removal time, O has a relatively small variance. This indicates that O can be fixed *a priori*, with only a small cost in compute time.

4.4 Performance and Comparison

In this subsection we examine how the Hybrid-REDUCE algorithm is likely to perform using existing technology, and how its relative performance is likely to change with technological advances. Throughout this subsection the operation is assumed to be CountSelected PEs, the data to be 32 bit integers, and the array to consist of a 256×256 grid of PEs.

The relative performance of primitive operations on existing technology was obtained by using results from the CAAPP prototype, a 64×64 array of bit-serial processing elements, and through hardware simulation of the full size array. The instruction durations in machine cycles are as follows: coterie communication instructions $t_{Cot} = 10$, GlobalCount $t_{GC} = 20$, and all other instruction, including ALU broadcast and PE arithmetic instructions, $t_{PE} = 1$. Since a single iteration of local and global removal take about 100 and 580 cycles respectively, the ratio $K = 6$. We have used average case values for the local and global loop counts from Figure 17: $O = 4$ and $R = 17$. These are justified because of the small variance of O. Bookkeeping added up to less than 100 cycles and was ignored.

$\mathsf{FirstTwoReductions} \Rightarrow 2 * \log n * (32 * t_{Cot} + 64 * t_{PE}) = 2624$ cycles

$$\mathsf{LocalRemoval} \Rightarrow O * \begin{cases} \mathsf{SelectSingleResponder} \Rightarrow \log N * (t_{Cot} + 2 * t_{PE}) \\ \mathsf{BroadcastPartialSum} \Rightarrow 32 * t_{Cot} \\ \mathsf{CombinePartialSum} \Rightarrow 64 * t_{PE} \end{cases} = 3480 \text{ cycles}$$

$$\mathsf{GlobalRemoval} \Rightarrow R * \begin{cases} \mathsf{GlobalSelectSingleResponder} \Rightarrow 3 * \log N * t_{PE} \\ \mathsf{GlobalCount} \Rightarrow t_{GC} \\ \mathsf{BroadcastResult} \Rightarrow 32 * t_{PE} \end{cases} = 2652 \text{ cycles}$$

Significant is that the local and global removal phases—the parts of the computation that are not asymptotically optimal—take a similar number of cycles as the reductions executed during the first phase. This means that executing extra reductions as in the SMCC-Reduce algorithm is not likely to improve performance.

	Ratio of Local to Global Removal Execution Times (K)									
	1	2	3	4	5	6	7	8	9	10
avg of O	13.3	9.7	8.2	6.6	6.2	5.7	5.3	5.2	5.0	4.8
Sigma of O	4.0	2.8	2.4	1.7	1.6	1.6	1.2	1.2	1.1	1.2
avg of R	9.3	13.4	16.7	21.7	23.6	25.8	28.4	29.2	30.9	32.2
Sigma of R	3.6	4.7	4.9	6.2	7.1	6.9	7.6	7.2	8.0	8.6

Figure 17: Average optimal values over the entire data set for local and global removal operations (O and R) and their standard deviations for different ratios of local to global removal execution times (K).

Two likely changes in hardware performance will alter the relative performance of the algorithms: the latency of the coterie network broadcast, and the path width of the coterie network bus. The expression

$$\mathsf{cycles} = \frac{512}{w}(t_{Cot} + 2) + O(\frac{32}{w} + 16)(t_{Cot} + 2) + R(70 + \frac{32}{w})$$

relates the performance of Hybrid-REDUCE to the two parameters, width $= w$ and coterie communication latency $= t_{Cot}$. The loop counter values O and R are again obtained empirically from Figure 17 where K is determined as follows:

$$K = \frac{(\frac{32}{w} + 16)(2 + t_{Cot})}{70 + \frac{32}{w}}.$$

The resultant changes in performance of the hybrid algorithm are illustrated in Figure18. The improvement due to increasing path-width levels off because SelectSingleResponder is inherently bit-serial.

The range of coterie latency to PE instruction ratios was selected with the following scenarios in mind. The curent ratio is about 10. An increase is likely as the VLSI process improves, while the basic packaging configuration

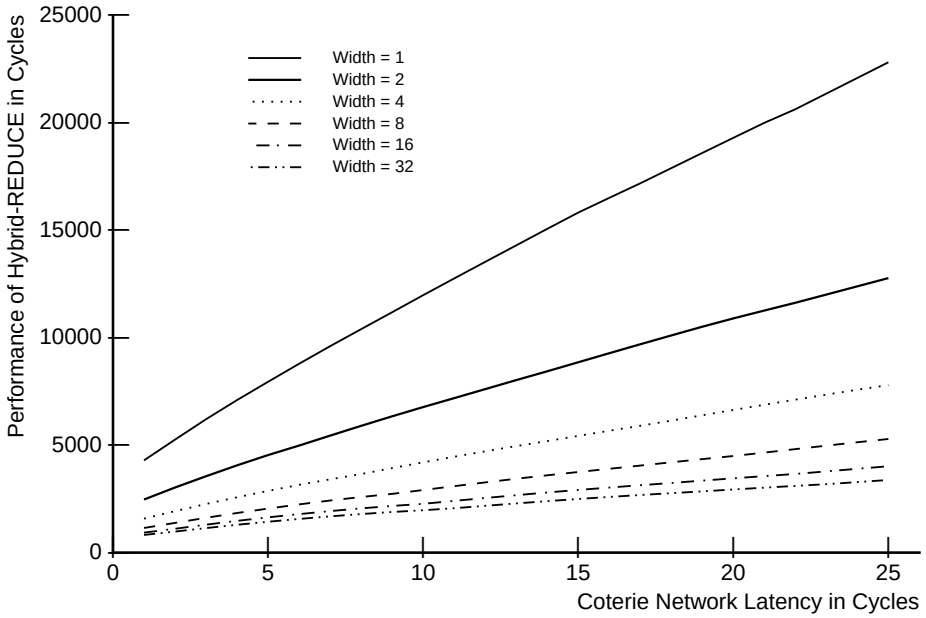

Figure 18: Expected performance of Hybrid-REDUCE given changes in network latency and path width.

remains as it is. We estimate a factor of 3 maximum relative speedup here over the short term as the design improves, but no greater since clock distribution problems for massively parallel arrays are significant. A decrease in the ratio is likely in future generation machines when the entire array will fit on a wafer (or chip). However, it is very unlikely that the coterie latency will ever be less than the global signal propagation, bounding the minimum of the ratio at 1.

5 Conclusion

We have presented new associative computational models, shown that they can be implemented efficiently on reconfigurable broadcast networks, and that they are effective in the domain of spatially mapped applications.

The method we use to implement multiassociative processing involves the use of coterie structures. This model can compensate for the absence of an efficient, general distributed memory access capability (available only in idealized models) through the orchestrated partitioning of the subgraphs induced by the data. In this way, certain PRAM results were shown to hold for arbitrary connected components of the reconfigurable mesh as well. We have also shown how to take advantage of coterie broadcast to break symmetries and to obtain information independent of distance.

The algorithms presented are significant from both a theoretical and a practical point of view. The former because they show the richness of this computational model, the latter because they are likely to be the fastest available. In fact Hybrid-REDUCE runs more than an order of magnitude faster than previous methods. Another significant result, however, is that we have shown it is possible to create efficient algorithms "without leaving the coterie"; that irregular graphs need not preclude good solutions for problems on reconfigurable broadcast networks.

We assume that the definitions, basic algorithms, and methodology will be useful in solving many other multiassociative problems. Especially promising is the use of coterie structures to solve problems in computational geometry. Of longer term interest is the problem of creating a system that efficiently responds to general multiassociative queries.

Acknowledgment

We would like to thank Seth Malitz for many useful conversations and for his encouragement in this project.

References

[1] Berge, C. *Graphs and Hypergraphs*. North-Holland, Amsterdam, 1973.

[2] Beveridge, J. R., Griffith, J., Kohler, R. R., Hanson, A. R., and Riseman, E. M. Segmenting images using localized histograms and region merging. *International Journal of Computer Vision 2*, 3 (1989), 311–347.

[3] Blelloch, G. E. Scans as primitive parallel operations. *IEEE Transactions on Computers C-38*, 11 (1989).

[4] Blelloch, G. E., and Sabot, G. W. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing 8* (1990), 119–134.

[5] Brice, C. R., and Fennema, C. L. Scene analysis using regions. *Artificial Intelligence 1* (1970), 205–226.

[6] Cormen, T. H., Leiserson, C., and Rivest, R. L. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[7] Falkoff, A. D. Algorithms for parallel search memories. *Journal of the ACM 9*, 4 (1962), 488–511.

[8] Feldman, J. A., and Yakimovsky, Y. Decision theory and artificial intelligence I: A semantics-based region analyzer. *Artificial Intelligence 5* (1974), 349–371.

[9] Foster, C. C. *Content Addressable Parallel Processors*. Van Nostrand Reinhold Co., New York, NY, 1986.

[10] Gauss, E. J. Locating the largest word in a file using a modified memory. *Journal of the ACM 8* (1961), 418–425.

[11] Goldberg, A. V., and Plotkin, S. A. Parallel $(\delta+1)$ coloring of constant degree graphs. *Information Processing Letters 25*, 4 (1987), 241–245.

[12] Herbordt, M. C., Weems, C. C., and Scudder, M. J. Non-uniform region processing on SIMD arrays using the coterie network. *Machine Vision and Applications 5*, 2 (1992), 105–125.

[13] Hillis, W. D., and Steele Jr., G. L. Data parallel algorithms. *Communications of the ACM 29*, 12 (1986), 1170–1183.

[14] Iverson, K. E. *A Programming Language*. John Wiley and Sons, New York, NY, 1962.

[15] Jenq, J. F., and Sahni, S. Reconfigurable mesh algorithms for the area and perimter of image components. In *Proceedings of the 1991 International Conference on Parallel Processing* (1991), pp. 280–281.

[16] Karp, R. M., and Ramachandran, V. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*. North Holland, Amsterdam, 1988.

[17] Ladner, R. E., and Fisher, M. J. Parallel prefix computation. *Journal of the ACM 27*, 4 (1980), 831–838.

[18] Li, H., and Maresca, M. Polymorphic Torus Network. *IEEE Transactions on Computers C-38*, 9 (1989), 1345–1351.

[19] Little, J. J., Blelloch, G. E., and Cass, T. A. Algorithmic techniques for computer vision on a fine-grained parallel machine. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11*, 3 (1989), 244–257.

[20] Maresca, M. Polymorphic processor arrays. *IEEE Transactions on Parallel and Distributed Systems PDS-4* (1993), 490–506.

[21] Miller, G. L., and Reif, J. H. Parallel tree contraction and its applications. In *Procedings of the 28th IEEE Symposium on the Foundations of Computer Science* (1985), pp. 478–489.

[22] Miller, R., Kumar, V. K. P., Reisis, D., and Stout, Q. F. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers C-42*, 6 (1993), 678–692.

[23] Obrenić, B., Herbordt, M., Rosenberg, A., Weems, C. C., Annexstein, F. S., and Baumslag, M. Using emulations to construct high-performance virtual parallel architectures. Tech. Rep. TR91-40, Department of Computer Science, University of Massachusetts, 1991.

[24] Phillips, C. A. Parallel graph contraction. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures* (1989), pp. 148–157.

[25] Rana, D., and Weems, C. C. The iua feedback concentrator. In *Proceedings of the International Conference on Pattern Recognition* (1990), pp. 540–544.

[26] Tenenbaum, J. M., and Barrow, H. G. Experiments in interpretation guided segmentation. *Artificial Intelligence 8* (1976), 241–274.

[27] Weems, C. C. *Image Processing on a Content Addressable Array Parallel Processor*. PhD thesis, University of Massachusetts, Department of Computer and Informaction Science, University of Massachusetts, Amherst, MA 01003, 1984.

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