PROJECT OVERVIEWS

The three overviews that follow are short reports of ongoing research in image understanding architecture, SIMD parallelism in computer vision, and software environments for parallel computer vision.

Image Understanding Architecture: Exploiting Potential Parallelism in Machine Vision

Charles C. Weems, Edward M. Riseman, and Allen R. Hanson
University of Massachusetts, Amhurst, MA 01003

Parallel processing is now generally accepted as necessary to support real-time image understanding applications. Much debate remains, however, about what form of parallelism to employ. Part of this debate stems from the tremendous amount and variety of potential parallelism in machine vision.

The sensory data alone is a good example: a medium-resolution image (512 × 512 pixels) consists of roughly a quarter of a million data values. In many cases, each of these values might be processed in parallel. Further, if images are obtained from a video camera, the steady stream of data lends itself to pipelined parallelism. Some data involves multiple sensors (for example, stereo or nonvisual spectral bands), thus providing yet another potential source of parallelism. Nor is it unusual to extract many different features from a given image or set of images (for example, lines, regions, texture patches, depth maps, and motion parameters), and these processes may also be carried out in parallel.

Beyond the sensory data, image understanding involves knowledge-based processing; and between these two levels of abstraction, symbolic processing has proved useful. Thus, vision researchers tend to classify algorithms and representations into three levels: low (sensory), intermediate (symbolic), and high (knowledge-based).

Of course, the existence of multiple levels of abstraction is yet another source of potential parallelism. Moreover, processing within each level presents many possibilities for exploiting parallelism. Part of the allure of developing a vision machine, from a computer architect's perspective, is this tremendous quantity, diversity, and complexity of latent parallelism. By comparison, most scientific and engineering applications have simple organizations with straightforward requirements for parallelism. (For more detailed analysis of the potential for parallelism in image understanding, see Weems.[1])

Image Understanding Architecture. Over the past five years, the University of Massachusetts and Hughes Research Laboratories have worked together to develop a hardware architecture that addresses at least part of the potential parallelism in each of the three levels of vision abstraction. A 1/64th-scale proof-of-concept prototype of this machine has been built and is shown in Figure 1. The machine, called the Image Understanding Architecture (IUA), consists of three different, tightly coupled parallel processors: the content addressable array parallel processor (CAAPP) at the low level, the intermediate communication associative processor (ICAP) at the intermediate level, and the symbolic processing array (SPA) at the high level. Figure 2 shows an overview of the architecture. The CAAPP and ICAP levels are controlled by an array control unit (ACU) that takes its directions from the SPA level.

The SPA is a multiple-instruction multiple-data (MIMD) parallel processor, while the intermediate and low levels operate in multiple modes. The CAAPP operates in single-instruction multiple-data (SIMD) associative or multiassociative mode, and the ICAP operates in single-program multiple-data (SPMD) or MIMD mode. In multiassociative mode, CAAPP cells execute the same instruction stream but in disjoint groups, with each group capable of operating on locally broadcast values and

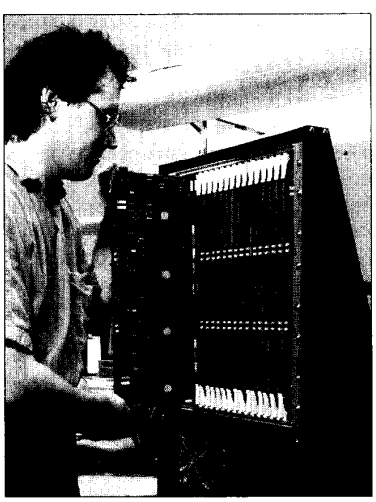

Figure 1. First-generation prototype of the Image Understanding Architecture.

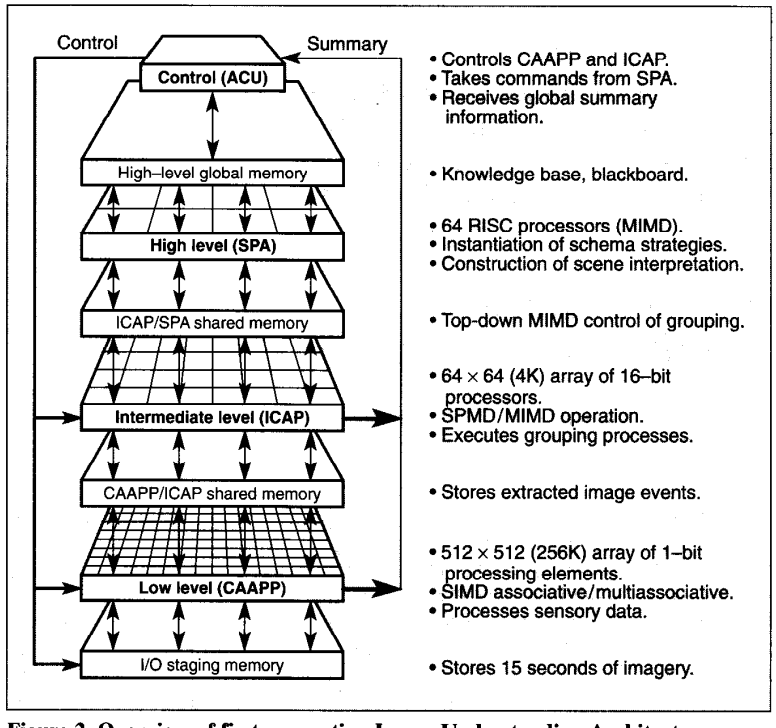

Figure 2. Overview of first-generation Image Understanding Architecture.

locally computing its own summary values in parallel with all other groups. In SPMD mode, the ICAP processors execute the same program but have their own instruction pointers so that they can branch independently.

How does the IUA address the various forms of potential parallelism described in our introduction? We will answer this question by considering the capabilities of each level in Figure 2. The I/O staging memory permits one or more sensors to input images into a buffer that can hold up to 15 seconds of imagery at 30 frames per second and a resolution equal to the size of the low-level processor array. The resolution of the images can differ from the array size, with a resulting increase or decrease in the number of frames that can be buffered.

The CAAPP consists of bit-serial processors, each with an arithmetic logic unit, registers, 320 bits of explicitly managed on-chip cache memory, and 32 Kbits of backing store (main) memory. Because it is a SIMD processor, its instructions are broadcast from the ACU. However, each processor also contains a one-bit register that controls whether it will respond to a particular instruction.

The processors are connected via a reconfigurable mesh, called the *coterie network*. Each processor controls four switches that configure the mesh connections to its four nearest neighbors (north, south, east, west) and four switches that permit signals to bypass the processor (northeast, northwest, horizontal, and vertical). When the switches are set, connected processors form a coterie. The mesh may simultaneously contain many nonoverlapping coteries.

Within a coterie, one processor may be selected to broadcast a value to the members of the coterie, or any subset of the processors may send a value bit-serially over the network. In the latter case, the processors receive the logical OR of the bits that were transmitted — that is, if some of the processors transmit a 1, then all processors receive a 1; but if none of the processors transmits a 1, then all processors receive 0. This some/none test is a valuable summary mechanism that can be used in many ways. For example, it can be used to determine the maximum of a set of values contained in a coterie.

If the array has been split into coteries corresponding to regions in an image, then we can use the maximum-value operation to label connected components. Each processor is merely given a unique value (its address) and then the maximum-value operation determines the maximum address within each coterie. The value is then broadcast to the members of the coterie as their component label. Note that all of this takes place in every coterie simultaneously, even though there is only a single instruction stream. In the CAAPP, connected-components labeling thus takes only about 50 microseconds. Many other operations on image regions and edges can be performed quickly when the network is arranged to match their shape. The ability to simultaneously perform queries and summarize results in independent groups of processors under a single instruction stream resulted in the term multiassociative for this mode of parallelism.

The main memory for the CAAPP is also directly accessible to the ICAP through a second port. Each ICAP processor has access to the 8 × 8 tile of CAAPP processors below it, providing a highly parallel data path between the two levels. Each ICAP processor is a 16-bit digital signal processor (DSP) with 128 Kbytes of program memory and 128 Kbytes of data memory. We selected a DSP because it provides a set of operations (such as single-cycle square and add) that are well suited to computations in spatial geometry. The DSP is also designed for use with a minimum amount of external logic, and it provides a set of communication channels that are used for interprocessor communication. As an example of its capabilities, the intermediate level can simultaneously match several thousand models against symbolic descriptions of events (tokens) extracted from an image by the CAAPP.

The ICAP connects to another dual-ported memory, which it shares with the

SPA. Each SPA processor can access data stored in this memory by any ICAP processor. Our current plans are to use a commercially available multiprocessor at this level to provide general-purpose computational capabilities for high-level processing. The SPA also has its own shared memory. The ACU, which manages the CAAPP and ICAP, is connected to that memory and communicates with the SPA processors as if it were just another processor of the same type. The full-scale IUA can thus process in parallel all pixels of a single 512 × 512 image, several thousand tokens, and up to 64 high-level processes. Simulations of the full-scale IUA have shown that it can support model-based recognition tasks at or near frame rate, which is considerably closer to real-time image understanding than previous systems. Nonetheless, even greater parallelism will be required to achieve true machine perception. (For more information on the first-generation IUA, see Weems et al.[2])

```
// Created by James H. Burrill, University of Massachusetts

# include "stream.h"
# include "IuaClassLib.h"

// Segment 'intensity_image' into regions by comparing the values of
// neighboring pixels. Return the pattern for the virtual Coterie switches.

CharPlane run_conn_comp(CharPlane &intensity_image)
{Everywhere active;        // Ensure that every pixel participates
  BitPlane temp;
  CharPlane save_connections;

  temp = (intensity_image == intensity_image.West()) &
         ~temp.WestEdge_p();
  save_connections.InsertBits(temp, WL);

  temp = (intensity_image == intensity_image.North()) &
         ~temp.NorthEdge_p();
  save_connections.InsertBits(temp, NL);

  temp = (intensity_image == intensity_image.East()) &
         ~temp.EastEdge_p();
  save_connections.InsertBits(temp, EL);

  temp = (intensity_image == intensity_image.South()) &
         ~temp.SouthEdge_p();
  save_connections.InsertBits(temp, SL);

  return save_connections;
}
```

Figure 3. Example C++ program using the image-plane class library.

Second-generation IUA. A second generation of the IUA, currently under development, reflects experience from the prototype construction, advances in machine vision research, and newer hardware technology. It retains the overall three-level structure of the first generation, but the CAAPP and ICAP levels have been significantly enhanced. The new hardware implementation will encompass 1/16th (16,384 CAAPP, 64 ICAP, and 4 SPA processors) of a full-scale second-generation system. The second-generation hardware will be half the physical size of the prototype IUA, yet will provide roughly 10 times the processing power of that system.

In the CAAPP, 256 processors now reside in a single chip, and each of these 16 × 16 processor arrays is associated with an ICAP processor. Rather than treat the I/O staging memory as an I/O device, the new CAAPP treats it as merely another bank of main memory. Greater flexibility has been added to the interface with the ICAP as well.

The ICAP processors now consist of 32-bit floating-point DSP chips, each of which is capable of 50 Mflops. In addition to the main memory of the CAAPP, each ICAP processor will have access to 1 Mbyte of local memory and 4 Mbytes of shared memory within a local cluster of four processors. Whereas the first-generation prototype connected the ICAP processors via a centrally controlled bit-serial crossbar, the 64 processors in the second generation will be fully connected by high-speed direct-memory-access channels. The array will also support a global shared memory, composed of all the local shared memories, with a hierarchical access mechanism.

Unlike the minimal ACU in the prototype system, the second generation will have a sophisticated controller, designed to support high-level languages and virtual processor arrays in the CAAPP. We have programmed the prototype CAAPP in Forth and C, using high-level syntax extensions to those languages that still require the programmer to have considerable knowledge of the machine's organization. These language extensions are really a halfway step between assembly language and high-level languages.

In contrast, the second-generation CAAPP will be programmed in standard C++, using a class library that defines image-plane data types. Programs written with the class library can be compiled and executed on any machine with a standard C++ compiler. To execute such programs on the IUA merely requires the use of a separate runtime library. Figure 3 shows a sample C++ program for the CAAPP.

The second-generation ICAP will be programmed in C with libraries to support interprocessor communication. An Ada compiler will also be available. A symbolic database system to support processing, grouping, and matching of extracted image events and model parts is currently under development for the ICAP.

The SPA will be programmed in yet another dialect of C, and a parallel Common Lisp compiler will be available as well. A blackboard system will be available to support knowledge-based pro-

cessing at the high level. One of our long-term goals is to develop a single, unifying model and language for programming the IUA so that programmers will not have to distinguish explicitly among the three levels.

The future. Elements of a third-generation IUA are already under study. We expect it to be a transitional step between the current three-level organization with a single low-level array and future generations that will incorporate multiple, heterogeneous, low-level processors called *virtual sensors*. It may also be possible to split the hardware into more than three levels and thereby represent finer divisions of the abstraction space in more complex vision applications.

Knowledge-based machine vision is both complex and computationally intense. It also presents a unique set of opportunities for exploiting parallelism. The Image Understanding Architecture has been built to capitalize on several of those sources of potential parallelism. Because the capacity for complex parallelism in vision is far beyond the capabilities of current technology, parallel architectures for vision will continue to evolve at the forefront of innovation in architectural research.■

Acknowledgments

This work was funded in part by the Defense Advanced Research Projects Agency under contract number DAAL02-91-K-0047, monitored by the US Army Harry Diamond Laboratory; contract numbers DACA76-86-C-0015 and DACA76-89-C-0016, monitored by the US Army Engineer Topographic Laboratory; and contract number F49620-86-C-0041, monitored by the Air Force Office of Scientific Research. Funding was also received under a Coordinated Experimental Research grant, DCA 8500322, from the National Science Foundation. We thank David B. Shu and J. Gregory Nash at Hughes Research Laboratories for their contributions to the IUA design and development.

References

1. C.C. Weems, "The Architectural Requirements of Image Understanding with Respect to Parallel Processing," *Proc. IEEE*, Vol. 79, No. 4, Apr. 1991, pp. 537-547.

2. C.C. Weems et al., "The Image Understanding Architecture," *Int'l J. Computer Vision*, Vol. 2, No. 1. Jan. 1989, pp. 251-282.

Charles C. Weems is a research assistant professor and director of the Parallel Image Understanding Architectures research group at the University of Massachusetts at Amherst. His research interests include parallel architectures to support low-, intermediate-, and high-level of computer vision; benchmarks for vision; parallel programming languages; and parallel vision algorithms.

Edward M. Riseman is a professor in the Computer and Information Science Department and codirector of the Laboratory for Computer Vision Research at the University of Massachusetts. His research interests include computer vision, artificial intelligence, learning, and pattern recognition.

Allen R. Hanson is a professor in the Computer and Information Science Department and codirector of the Computer Vision Laboratory at the University of Massachusetts. His research interests are artificial intelligence, computer vision and image understanding, and pattern recognition.

Effective Use of SIMD Parallelism in Low- and Intermediate-Level Vision

P.J. Narayanan, Ling Tony Chen, and Larry S. Davis
Computer Vision Laboratory, Center for Automation Research
University of Maryland, College Park, MD 20742-3411

During the past three decades, powerful new methods for image analysis have emerged, along with novel architectural concepts for the design and construction of massively parallel machines. These developments are often motivated by the need to process images at high speeds. However, with some notable exceptions, research on architectures for image understanding has been driven more by classical models of image processing (essentially, image-to-image transformations and global feature extraction) than by the more powerful image representations and processing methods developed by the image understanding community.

In this article we consider two examples from image understanding — focus-of-attention vision and contour image analysis — and present new parallel-processing methods that effectively support these types of computations. Our research is a blend of theory and practice. On the one hand, we aim to develop algorithms whose properties are well understood and can be formally related to key aspects of machine models. On the other, we want algorithms that are easy to implement and practical in terms of their actual processing times on existing parallel machines. Our experimental research was conducted on a 16,384-processor Connection Machine CM2, and we present results of algorithm implementations on that machine.

In focus-of-attention vision, we use expectations about image structure to limit the image's processing to regions expected to contain key image features. Focus-of-attention vision is a powerful control strategy for image understanding because it lets us limit processing to relatively small subsets of an image (especially critical for sequential implementations of image understanding sys-

tems). Perhaps more importantly, it lets us use specialized feature-extraction algorithms that are highly tuned by our expectations.

How can we use massively parallel computers to build vision systems based on focus-of-attention methods? Massively parallel computers have tens of thousands of processors, and focus-of-attention vision systems analyze relatively small image windows (typically containing only thousands of pixels). The classical approach for processing images on massively parallel machines — assigning each pixel to a processor — will leave most of the machine idle. It would be preferable to use as many of the processors as possible and have the time needed to process an image window be a function of the number of pixels in that window.

We have been studying the use of data replication techniques to achieve the goal of efficient focus-of-attention vision on massively parallel machines.[1] We replicate the window to be processed many times and decompose basic image analysis operations into components that can be computed simultaneously on a SIMD machine. We briefly describe this research in the next section, "Replicated-data algorithms."

Contours (extended edges) are important image structures for both matching and recognition. Many computational stereo models and motion analysis models are based on an analysis of the geometry of image contours. Additionally, most object recognition systems operate by initially reducing the intensity image to a set of contours and then matching their geometric properties against stored models.

Contours are marked in images by processes such as edge detection and thresholding. Although it is possible to operate on the contours while they are embedded in the two-dimensional image, several reasons make it desirable, especially on richly interconnected machines like the Connection Machine, to transform the image contours from their image-plane embedding to a linear representation. The two primary reasons are

(1) The original image will ordinarily have far more pixels than the number of processors in the massively parallel computer. However, the number of pixels marked as edges by an edge detector is ordinarily only a small percentage of the pixels in the image. If we can remove the contour pixels from the image plane and store them in linear data structures, then we obviate the need to process the "empty" space in the edge image and reduce the "virtual processor ratio" (the ratio of the number of logical processors needed to perform an operation to the number of available physical processors) to 1.

(2) Richly connected machines like the hypercube-connected Connection Machine efficiently support a set of very powerful operations called *parallel prefix,* or *scan,* operators. These let us compute properties of processor subsets in time proportional to the logarithm of the subset size. Because of the arbitrary pattern of processor addresses encountered when traversing an image contour, prefix operations cannot be effectively applied to the contours while they are embedded in the image plane. Transforming the image contours to a linear representation allows us to use these prefix operations.

In summary, processing contours in their image-plane embedding makes the processing time proportional to the size of the image, while operating on them in the compact, linearized representation makes the time proportional to the logarithm of the longest contour in the image. This is a significant difference. The key is to perform the transformation from the image to the linear representation efficiently. This transformation involves, as a first important step, ranking the pixels in each contour. In a previous paper[2] we presented one simple $O(\log N)$ algorithm for ranking image contours (containing N points) and discussed its implementation on the Connection Machine. In this article we sketch the algorithm and illustrate the advantages of linearizing contours by considering the problem of piecewise linear approximation of contours.

Replicated-data algorithms. Focus-of-attention vision can be supported by an approach based on techniques of data replication (see Narayanan and Davis for more details[1]). Our approach involves replicating image windows many times on the processor array and decomposing a computation into subtasks that are solved simultaneously using the copies. The partial results from the copies are combined to generate the overall problem solution. The technique uses data parallelism within each copy of the data structure and operation parallelism across the copies. The justification for this approach is that the number of processors is becoming a less critical resource in data-parallel computing using massively parallel SIMD processor arrays, and its importance will continue to decline. To speed up processing of the relatively small data structures that arise in focus-of-attention vision on such machines, we need to devise techniques using a greater number of processors than there are data elements in the data structure, and divide the task performed on each data element among multiple processors.

Data-parallel algorithms depend on efficient embedding of the data structure onto the topology of the processor array machine's interconnection network. In replicated-data algorithms, embedding has two dimensions: (1) Embedding of the individual copies must map proximate data elements to adjacent processing elements; (2) corresponding data elements in different copies should have an efficient interconnection pattern among themselves for efficient computation across the copies. The mechanisms provided by the machine to distribute the data to the different copies and to combine the partial results from the copies are also critical in the design of a replicated-data algorithm.

We illustrate our technique using digital image convolution as an example. Digital image convolution by a two-dimensional kernel of weights is used in a variety of operations in image processing — for example, in smoothing and edge detection. Digital image convolution by a $k \times k$ kernel, for an odd number k, is defined as follows: Assume the indices of the kernel range from $-\lfloor k/2 \rfloor$ to $\lfloor k/2 \rfloor$. Then each pixel $P(u, v)$ of the image is mapped to a con-

Table 1. Comparison of the convolution algorithms on the Connection Machine. Timing of convolution is shown in seconds for random image and kernel.

Image Size	Kernel Size	VP Ratio	Replicated-Data Algorithm	One-Copy Algorithm
64 × 64	3 × 3	4	0.014	0.005
64 × 64	5 × 5	8	0.023	0.024
64 × 64	7 × 7	16	0.050	0.091
64 × 64	11 × 11	32	0.098	0.367
64 × 64	15 × 15	64	0.212	1.363
64 × 64	21 × 21	128	0.400	5.355

volved value $C(u, v)$ as given in

$$C(u,v) = \sum_{i=-k/2}^{k/2} \sum_{j=-k/2}^{k/2} P(u+i, v+j)K(i,j)$$

where P is the image array and K the kernel array.

The replicated-data algorithm to compute the convolution of an $n \times n$ image, where $n = 2^{m-1}$, by a $k \times k$ kernel of weights is given below. The algorithm uses k^2 copies of the image. These copies can be visualized as forming a two-dimensional square array whose indices range from $-\lfloor k/2 \rfloor$ to $\lfloor k/2 \rfloor$. The kernel weights are distributed one per copy such that copy (i, j) gets the value $K(i, j)$ for $-k/2 \le i \le k/2$ and $-k/2 \le j \le k/2$.

The algorithm's four steps are

(1) Using a scan operation with copy operator, copy the image stored in copy 0 to all copies.

(2) Broadcast the kernel weights to the copies in k^2 steps such that all processing elements (PEs) of the copy (i, j) get the kernel weight $K(i, j)$.

(3) The PE allocated to pixel (u, v) of copy (i, j) of the image performs the following computation: Obtain the pixel value from PE $(u + i, v + j)$ within the same copy (assume that all pixels outside the image have a gray level of 0) and store it in P as its pixel value. Multiply this value by the kernel weight allocated to the copy $K(i, j)$ and store this result in C.

(4) Perform a scan operation with "add" as the operator on C across the copies of each pixel of the image and store the result in a designated copy, say copy (0,0). The scan result gives the convolved image for each pixel.

The replicated-data convolution algorithm was implemented on a 16,384-processor Connection Machine CM2. Table 1 compares the algorithm's performance with that of a single-copy data-parallel algorithm. The virtual processor capability of the Connection Machine, in which each physical PE simulates multiple PEs, was used in the implementation, since the replicated-data algorithm needs many more processors than the physical array can support. The number of virtual PEs each physical PE simulates is called the virtual processor ratio, or VP ratio (given in the third column of the table). The fourth column gives the timing of the replicated-data algorithm. The timing for the single-copy algorithm is given in column five when the CM2 was configured to have the same VP ratio as the corresponding replicated-data algorithm, thus making the comparison fair. The front-end computation overhead was negligible in both cases. On the whole, the replicated-data algorithm achieves impressive speedup over the single-copy version for image convolution operations.

Replicated-data algorithms reduce the computation time by exploiting operation parallelism, but they incur overhead in distributing the data to the copies and in combining the partial results from them. This overhead is due to communications and computations across the copies of the data structure and depends on the efficiency of those operations when multiple copies are mapped onto the interconnection network. This efficiency is affected by the topology of the interconnection network and the time to perform near-neighbor communications and arithmetic operations.

We analyzed the replicated-data convolution algorithm on different common interconnection networks and determined the results for binary hypercubes, the underlying architecture of the CM2. The replicated-data algorithm performs better than the single-copy algorithm on hypercube architectures. For computing the convolution over a $k \times k$ neighborhood of an $n \times n$ image using k^2 copies, the speedup is given by

$$S = \frac{k^2}{2 \log k \dfrac{2t_c^n + t_a}{t_m + t_a + t_c^n} + \dfrac{t_c^h}{t_m + t_a + t_c^n}}$$

where t_c^n is the time for a near-neighbor communication, t_c^h is the time for a general hypercube communication, t_a is the time for an addition on the machine, and t_m the time for a multiplication. Figure 1 compares the speedup predicted by the above equation with the speedup obtained in practice on the CM2.

There are other low-level-vision operations that benefit from the technique of data replication. Previously, we presented analysis and implementation results of the replicated histogram algorithm.[1] Table lookup, Hough transform computations, and co-occurrence matrix computations are some of the other operations that can be speeded up using techniques similar to those used in the histogram algorithm.

Image algebra is an architecture-independent language that can describe a large class of image operations. Convolution belongs to the class of generalized template operations defined in image algebra. We developed a method to automatically generate a replicated-data algorithm for any image operation that can be described in terms of a generalized template operation.[1] We also developed replicated-data algorithms for rank-order filters, which are local nonlinear image operations. Rank-order filters are expensive to compute on SIMD machines, as they involve independent sorting of the neighborhood pixels within each processor. Sorting can be performed quickly if the neighborhood elements are distributed among many

processors using the sorting algorithms of the underlying interconnection network. For instance, k neighborhood pixels can be sorted in a replicated scenario on the hypercube in $O(\log^2 k)$ time, whereas independent sorting on each PE takes $O(k^2)$ time on SIMD architectures. By assigning different areas of the search space to different copies of the data structure, we are extending the replication technique to problems in intermediate- and high-level vision that contain a combinatorial component.

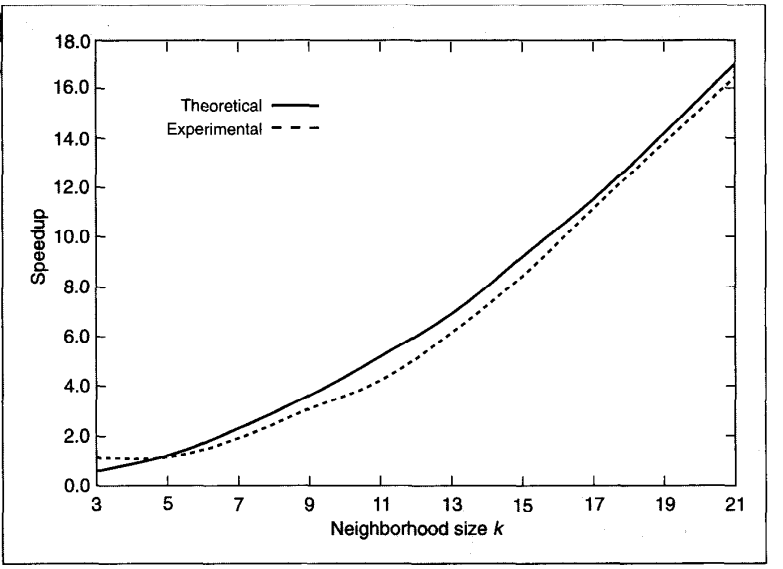

Figure 1. Comparison of experimental and theoretical speedups.

Image contour analysis. We turn now to problems associated with the efficient processing of image contours using hypercube-connected massively parallel computers. An important advantage of hypercube machines like the Connection Machine over the more common mesh network is that one can efficiently compute parallel prefix operations using the hypercube network. However, such operations can be applied only to a sequence of processors in which each processor can randomly access information from processors distance 2^i away from it, where i is an integer greater than zero. Since a contour can wind freely through an image, the sequence of processor addresses associated with the pixels on a contour will not generally have this property. Below we present an efficient algorithm for ranking the pixels on a contour. Once the pixels are ranked, the contour can be moved to a new set of processors whose addresses will form a monotonic continuous sequence (by simply moving the ith contour element to processor i), thus allowing random access between processors. The algorithm we present runs in $O(\log N)$ time (where N is the length of the contour) on either an exclusive read, exclusive write (EREW) parallel random access machine or a distributed-memory machine with EREW ability between memory modules. (See the literature for more details.[2])

The problem is as follows: We are given an $N \times N$ binary image that contains only thin curves. Each pixel on each curve has exactly two neighboring pixels adjacent to it, with the exception of endpoints, which have only one neighbor. Closed curves (curves that form a loop and have no endpoints) can also occur in the image. Our goal is to list rank all the curves in the image in parallel such that all open curves have one of their endpoints marked as the head and all other pixels on that curve determine their distance (along the curve) from that endpoint. For closed curves, any arbitrary pixel can be chosen as the head, as long as all pixels on the curve agree on which pixel is the head and every pixel determines its distance from that head in a consistent direction (clockwise or counterclockwise).

The algorithm starts with two pointers, P1 and P2, pointing to neighboring pixels. The algorithm uses these pointers to form an Euler tour path within each curve. Figure 2 illustrates this for a five-pixel open curve. If the Euler tour path is followed around an open curve, each pixel on the curve will be visited exactly twice, while the two endpoints are visited only once. Let's call each element (pointer plus distance value to be computed) in the Euler tour path a node. The Euler tour is easily initialized.

After the Euler tour path is initialized, each pointer in the path repeatedly does pointer jumping (distance doubling) while remembering the maximum node address seen so far, as well as the distance along the tour path to this maximal node. In Figure 2, node 8 would be the node with the maximum address. Thus, after pointer jumping has terminated, all nodes in the Euler tour path will have identified node 8 as having the maximum address and will have computed the distance along the path to node 8. At this point, a list ranking for the open curve can be computed by simply having each pixel compute the minimum value of the distance to node 8 for the two nodes associated with that pixel.

Many details have been omitted here, such as how to initiate the Euler tour path, when to terminate the pointer-jumping loop, and the changes necessary to handle closed curves; a more detailed discussion is available.[2]

The list-ranking algorithm was implemented on the Connection Machine. Table 2 shows the results obtained by running our EREW $O(\log N)$ list-ranking algorithm on the CM2 for different

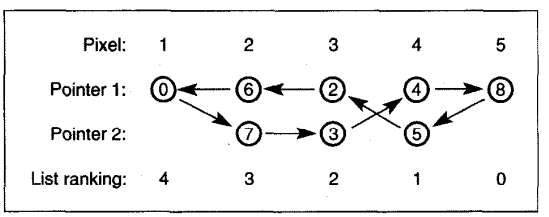

Figure 2. Example of Euler tour path on an open curve with five pixels.

Table 2. Result of CRCW and EREW $O(\log N)$ time algorithm versus $O(N)$ time algorithm (virtual processor ratio = 8).

Curve Length	CRCW Algorithm		EREW Algorithm		$O(N)$ Time Algorithm	
	Iterations	Time (ms)	Iterations	Time (ms)	Iterations	Time (ms)
64	5	193	8	236	64	605
128	5	201	9	283	128	1,219
256	6	244	10	334	256	2,438
512	6	262	11	387	512	4,859
1,024	7	334	12	476	1,024	9,696
2,048	8	487	13	726	2,048	19,410
4,096	8	715	14	1,162	4,096	38,827

longest curve lengths. An image can contain many curves, and the algorithm will rank them all simultaneously. However, the time needed to rank the entire set of image curves is determined by the length of the longest curve in the image. As a comparison, the running times required by the concurrent read, concurrent write (CRCW) algorithm, presented elsewhere,[2] are also shown, as are times for a trivial linear-time algorithm that propagates the list-ranking information along each curve pixel by pixel. We can see that the EREW algorithm is slower than the CRCW algorithm, but both are much faster than the linear-time algorithm. The algorithms were applied to a 512×512 image using 8,192 physical processors; thus, the VP ratio was 8 for all these experiments.

Next, we use piecewise linear approximation of curves as an example to illustrate how contours can be processed efficiently once they are linearized. Peucker devised a method for finding piecewise linear approximations of curves by breaking curves at points that are farthest from the line that connects the two endpoints of the curve. By repeatedly applying this curve-breaking method until all pixels of each curve are within a threshold distance from the line connecting the endpoints, we can obtain a good piecewise linear approximation.

This algorithm can be implemented easily on our monotone contiguous mapping between pixels and processors. The algorithm involves the following steps:

(1) Perform a reverse first scan on the x and y coordinates, so that the first processor of each curve segment has the x, y coordinates of the two endpoints of the segment.

(2) The first processor of each curve calculates the coefficients of the line that passes through its endpoints.

(3) A forward first scan is performed to broadcast the coefficients of this line to all pixels in this curve.

(4) All pixels calculate, in parallel, the distance between themselves and the line joining the endpoints.

(5) A reverse max scan is performed on this distance concatenated with the processor ID number of each pixel. This results in the first processor in each segment's knowing the processor ID number of the largest address processor having maximal distance from the line. Let m_i be the address of the processor having maximal distance in curve segment i.

(6) If this maximum distance is smaller than a threshold, the segment deselects itself and is idle through steps 7 and 8. If all segments in the image deselect, the algorithm terminates.

(7) A forward first scan is used to broadcast m_i to all processors in curve segment i.

(8) Processor m_i sets its segment flag to "True," thus splitting curve segment i for the next iteration.

(9) Steps 1 through 8 are repeated until the algorithm finally terminates at step 6.

We applied this piecewise linear approximation algorithm to three different 512×512 test images using the scan instructions available on the CM2. Table 3 shows the results. Edge detection and the list-ranking algorithm were applied to the three images, and the remaining pixels were packed into monotone contiguous processors. At this point each image contained C_{start} contours, and after applying the algorithm, each contained C_{end} linear segments. We can see that typical images will require five to six iterations of the algorithm before termination. On the CM2, each iteration takes roughly 7 milliseconds.

We are working to develop practical curve-matching algorithms, as well as stereo-matching algorithms. We have also worked on an efficient parallel algorithm for computing the visibility graph of a polygon by using only parallel prefix operations for communication (closed curves can be transformed into polygons by the piecewise linear approximation algorithm).

The research discussed here focuses on the effective use of massively parallel computation for representative problems in intermediate-level vision. One

Table 3. Result of applying the algorithm.

Total		Splitting Algorithm			
Processors	Pixels	C_{start}	C_{end}	Iterations	Time (ms)
8,192	7,706	581	800	5	36
8,192	6,735	515	700	5	37
8,192	7,943	630	877	6	46

of the greatest challenges facing the image understanding community is to discover how to use parallelism to address problems in high-level vision — that is, image interpretation and scene analysis. While image understanding itself is the least developed aspect of the field, we can see several architectural solutions emerging in the current decade. These include the use of heterogeneous but tightly coupled systems like the Image Understanding Architecture, which attempts to capture one of the basic image understanding paradigms of the 1980s, and the use of homogeneous massively parallel systems, which use a single computational paradigm (for example, neural computing, connectionism, constrained combinatorial analysis) to address high-level-vision problems. These and alternative vision architectures deserve the attention of vision researchers in the 1990s.■

Acknowledgments

The support of the Defense Advanced Research Projects Agency (DARPA Order No. 6350) and the US Army Engineer Topographic Laboratories under contract DACA76-88-C-0008 is gratefully acknowledged.

References

1. P.J. Narayanan and L.S. Davis, "Replicated-Data Algorithms in Image Processing," Tech. Report CAR-TR-536/CS-TR-2614, Center for Automation Research, University of Maryland, College Park, Md., 1991.

2. L.T. Chen and L.S. Davis, "A Parallel Algorithm for List Ranking Image Curves in $O(\log N)$ Time," *Proc. DARPA Image Understanding Workshop*, 1990, pp. 805-815. Also Tech. Report CAR-TR-541/CS-TR-2629, Center for Automation Research, University of Maryland, College Park, Md.

P.J. Narayanan is a doctoral student in computer science at the University of Maryland at College Park, where he works in the Computer Vision Laboratory. His research interests include parallel algorithms and architectures, and computer vision.

Ling Tony Chen is a doctoral student in computer science at the University of Maryland at College Park. His research interests are computer vision, parallel algorithms, and shape analysis.

Larry S. Davis is a tenured professor in the Department of Computer Science and director of the University of Maryland Institute for Advanced Computer Studies. He has published extensively on topics in image processing and computer vision.

A Software Environment for Parallel Computer Vision

Leah H. Jamieson, Edward J. Delp, and Chao-Chun Wang,
Purdue University, West Lafayette, IN 47907

Juan Li, IBM, San Jose, CA 95193
and Frank J. Weil, Motorola, Schaumburg, IL 60196

We are developing a software environment tailored to computer vision and image processing (CVIP). Although obtaining highest performance on parallel systems will almost certainly require sophisticated knowledge of parallel processing (for example, see Stout[1]), it is both unrealistic and undesirable to expect a researcher in the CVIP area to be an expert in parallel problem-solving techniques or parallel architectures. It is essential to provide tools that let applications researchers achieve reasonably high performance at a reasonable level of programming effort. The software environment focuses on how information about the CVIP problem domain can make the high-performance algorithms and the sophisticated algorithm techniques being designed by algorithms experts more readily available to CVIP researchers.

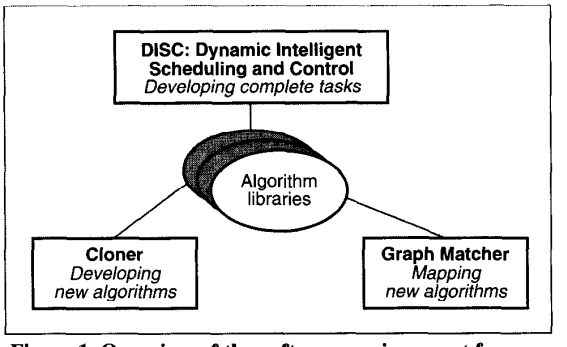

Figure 1. Overview of the software environment for computer vision and image processing.

The software environment consists of three principal components — DISC, Cloner, and Graph Matcher — shown in Figure 1. At the heart of the environment, and key to the operation of all three components, is a set of algorithm libraries, along with a metalevel of algorithm characteristics that abstract information about the library programs. The environment also includes traditional compilers, debuggers, and operating systems components. However, our goal is to exploit the special characteristics of CVIP to achieve easier algorithm development and better performance than can be expected with general-purpose tools; therefore, we focus on the subsystems in Figure 1. Each com-

ponent addresses a different aspect of the problem of rapid prototyping for CVIP algorithms and tasks:

- *DISC* (dynamic intelligent scheduling and control) supports experimentation at the CVIP task level by creating a dynamic schedule from a user's specification of the algorithms that constitute a complex task.
- *Cloner* is aimed at the algorithm development process and is an interactive system that helps a user design new parallel algorithms by building on and modifying existing library algorithms.
- *Graph Matcher* performs the critical step of mapping new algorithms onto the target parallel architecture.

We have completed initial implementations of DISC[2] and Graph Matcher[3]; work on Cloner is in progress. The remainder of this article summarizes the components of the CVIP software environment.

DISC: A dynamic scheduler for executing computer vision tasks. The DISC system is designed to facilitate system prototyping, the experimental process during which a user tests strategies for performing a complex task by trying different component algorithms, different orderings of algorithms, and different strategies for controlling the selection and sequencing of algorithms. DISC is implemented as an expert system that uses a library of low-, mid-, and high-level-vision algorithms and alternative parallel implementations, a database of execution characteristics of CVIP algorithms, rule-based heuristics, and the current system state to produce and continually update a schedule for the subtasks (algorithms) that constitute the overall task. The scheduler keeps track of what subtasks are potentially executable and chooses the best candidate by considering the relative importance of finishing the subtask quickly and the extent to which the current allocation of data in the machine partitions (subsets of processing elements) matches the data allocation needed by the subtask. DISC also controls repartitioning and compaction of the system.

Figure 2 is a graph representing data dependencies among the algorithms constituting a sample task. The primitives of the DISC language are the library algorithms. The graph is derived from a sequential listing of the algorithms and their arguments. The data dependencies are derived from the input/output specifications for the parameters for each algorithm. Once a subtask is chosen for execution, the scheduler selects the most suitable implementation of that subtask from the library. Implementations may differ by

- the way data is allocated to processing elements (for example, pixel data allocated by rows versus by square subimages, and contours allocated by object versus by coordinates);
- the format of the input and output parameters (for example, binary image versus edge list);
- mode (single instruction, multiple data, or SIMD; or multiple instruction, multiple data, or MIMD); and
- range of number of processors usable by the implementation.

DISC selects an implementation based on how well its characteristics coincide with the current data allocation, data format, and mode of the chosen system partition, and based on the expected relative speedup for the size of the partition. The scheduling is performed dynamically to handle situations common in vision applications: algorithms for which the execution time depends on the input image (for example, boundary tracing) and tasks in which the actual sequence of algorithms executed may vary depending on characteristics of the image (for example, the Edge linking/Edge continuity test loop in Figure 2).

DISC has been implemented with the PASM (partitionable SIMD/MIMD) parallel-processing system as its target architecture. Evaluation to date has consisted of simulations of tasks covering a spectrum of dependency graphs. In these tests, each library algorithm was represented by at least 12 implementations — typically two or three different implementation strategies executable on six different partition sizes. For algorithms with image-dependent execution times, the simulated times were randomized by the simulation controller, and at least 100 runs were performed.

Performance was evaluated using measures including utilization and scheduling overhead. Utilization is measured as the percentage of the processor-time space during which processors are not idle. Figure 3 shows a processor-time diagram for the task in Figure 2. Over the nontrivial tasks on which DISC has been tested, a 77-percent average utilization was achieved, and on the tasks in the test suite for which the optimal schedule was known, DISC obtained the optimal schedule.

Scheduling overhead is used to assess the amount of overhead DISC incurs in creating a schedule. The scheduling overhead is counted as the number of rules

Algorithm characteristics

The ability to characterize algorithms is key to the DISC and Cloner systems. The characteristics used are derived from the following general set of parallel-algorithm characteristics:

- Nature of parallelism: data or function.
- Data granularity or module granularity: the size of the data items processed as a unit or the size of independent modules.
- Degree of parallelism.
- Uniformity of operations, expressed as the smallest data granularity at which uniform operations are performed.
- Synchronization requirements and precedence constraints.
- Static/dynamic character of the algorithm, in terms of the pattern of process generation and termination.
- Data dependencies and related issues of data allocation and memory access patterns.
- Five characteristics shared by serial and parallel algorithms: fundamental operations, data types and precision, memory requirements, data structures, and I/O.

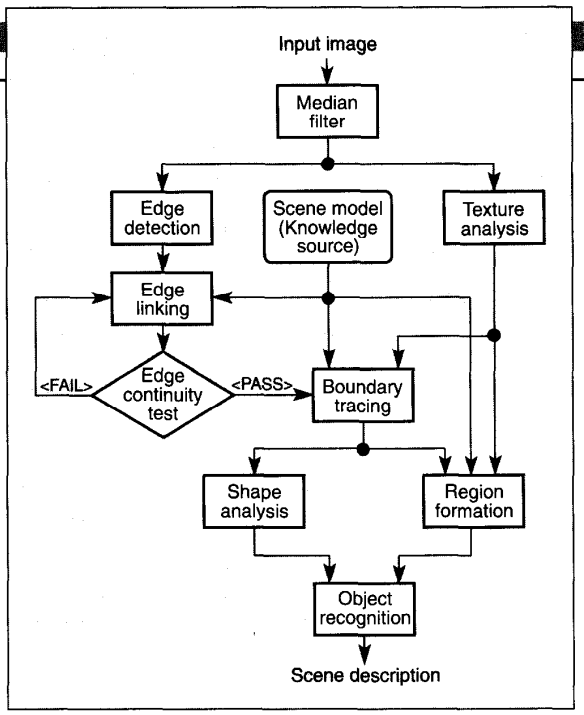

Figure 2. Algorithm data-dependency graph for a CVIP task to locate tanks in forward-looking infrared radar images.

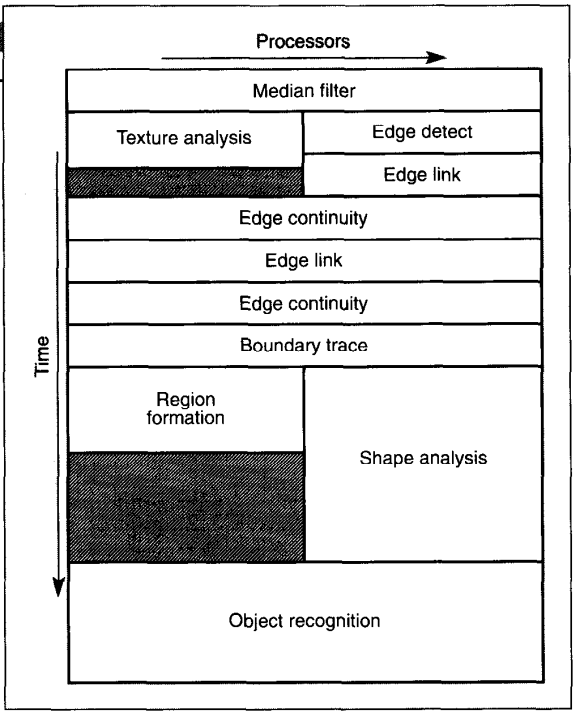

Figure 3. Example processor-time diagram for the task in Figure 2. (Time axis is not to scale.)

fired to begin the execution of an implementation of each algorithm. Using realistic assumptions to relate simulation time units and real time, scheduling overhead was measured to be less than 0.1 percent of the overall execution time.

Current work on DISC includes expansion of the library, refinement of the scheduling heuristics, performance analysis based on stochastic modeling of image-dependent execution characteristics, and packaging as a tool portable to a variety of parallel architectures.

Cloner: An interactive environment for developing parallel algorithms. Cloner is a software reuse tool that helps a user design parallel algorithms by building on and modifying existing library algorithms. It takes advantage of the fact that CVIP algorithms, especially for low-level vision, are often highly structured and that many image- and graph-based algorithms share the same, or a similar, structure. Cloner's dual goals are rapid prototyping and improved code quality.

The library forms the heart of Cloner. The user is provided with information about what algorithms and programs the system "knows about" and is guided through the process of developing a new program by relating it to existing programs. This may be done by a combination of operations, including the adoption and/or modification of library data abstractions (objects), the composition of library code segments, and the modification of library code templates. Emphasis is on providing a characterization of the library algorithms and kernels to act as an interface between the library and the user. Cloner exploits reusable code at several levels:

- the data-structure/data-abstraction level,
- the control structure level, and
- the algorithm level.

Interaction is via menus and queries that lead the user through an adaptive series of displays and questions designed to let either the user or Cloner select code templates. The questions are based on characteristics of the algorithm in terms of attributes such as data versus function parallelism and dominant data structures. Graphical displays show data-dependency patterns associated with a library program or kernel (for example, an operation structured around a 3-by-3-pixel window) and highlight potential reusable code fragments (for example, an optimized looping structure displayed to delimit the control structure and the replaceable loop body).

We are now building *Xcloner*, an X window-based implementation of Cloner. Figure 4 on the next page shows an example of the main Xcloner menu for image-processing operations. The categories in the main menu bar identify the major operations provided by Xcloner. Each category provides a pull-down menu. At the highest level, Xcloner is machine independent. Information specific to the target parallel system resides within the menu options and in the coding and mapping of the library algorithms. The initial implementation focuses on image processing and low-level-vision algorithms because the obvious common structures for these algorithms facilitate the development of the user interface and library access tools. However, many algorithms performed in mid- and high-level vision use well-defined structures such as graphs

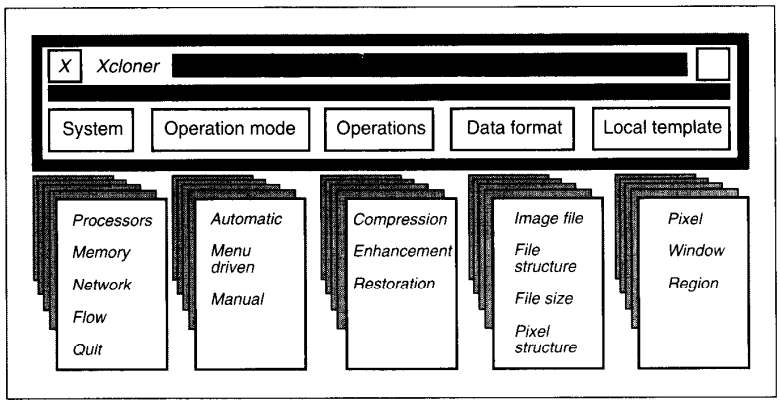

Figure 4. Example of Cloner display for image-processing algorithms.

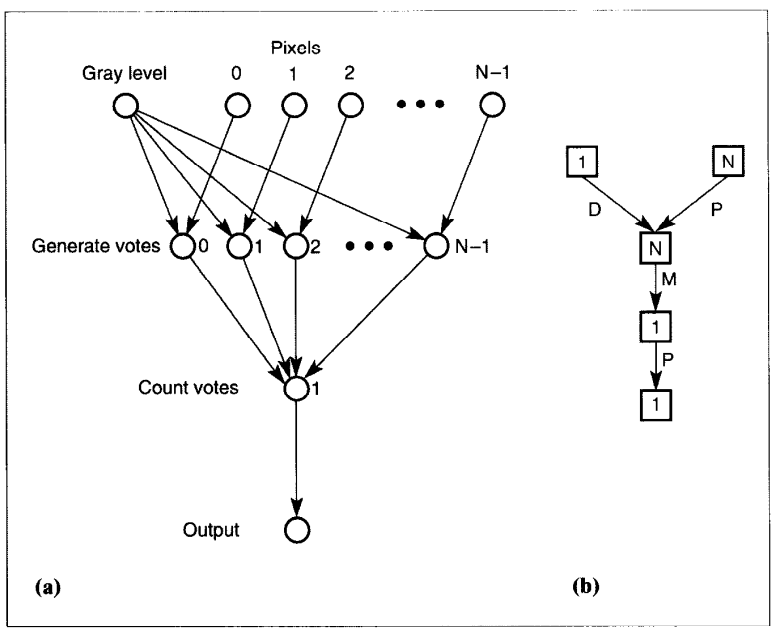

Figure 5. Dependency graph (a) and hypergraph (b) for a histogramming algorithm. *N* = number of pixels. Hypernode labels show the number of DG nodes combined to form the hypernode. Hyperedge labels characterize the pattern of the edges merged to form the hyperedge: *D* = Distribute (one to many); *P* = Parallel (one to one).

to represent regions and objects. Future work includes expansion of Cloner to include these algorithms.

Graph Matcher: Mapping CVIP algorithms onto parallel architectures. If a user has specified a new algorithm directly (for example, by a data-dependency graph or dataflow graph, or as a program that is transformed to such a graph), then the critical step in mapping the algorithm onto the parallel architecture involves the assignment of algorithm steps to processors. The objective is to equalize the work done by the processors and to minimize overhead from interprocessor communication. General-purpose compilers and mapping tools are designed to perform this function. However, in many algorithms with regular structure, it is possible to exploit prior experience to obtain mappings that might be difficult or prohibitively time-consuming to derive using general-purpose tools.

Graph Matcher uses pattern-matching techniques to recognize the data-dependency structure of a new algorithm as an instance of a dependency structure for which an algorithm-to-architecture mapping is already known. At the data-dependency level, especially in low-level vision and image processing, many algorithms share communications patterns:

- patterns typical for window-based algorithms,
- patterns typical for block-based algorithms,
- patterns characteristic of transforms, and
- patterns typical for collecting image statistics.

At the process level, algorithms based on the same paradigm (for example, divide and conquer) may exhibit similar communications requirements.

Graph Matcher consists of a library of known data-dependency structures and mappings of these structures onto architecture configurations. The input to Graph Matcher is a directed acyclic graph representing a new algorithm. The process of matching the input dependency graph (DG) to one of the library graphs can be formulated as a graph isomorphism problem; however, the complexity of graph isomorphism makes direct and exhaustive comparison to the library infeasible. Graph Matcher therefore relies on heuristics that use easy-to-compute graph attributes and that take advantage of the regular structure of many image-processing algorithms.

In a candidate elimination step, easily measured features of the input DG are compared to features of the library DGs to eliminate library graphs from further consideration. Features include

Table 1. Comparison of dependency-graph sizes and hypergraph sizes.

Algorithm	Parameters	No. of Nodes	No. of Edges	No. of Hypernodes	No. of Hyperedges
Histogram	No. of pixels: N	$3 + 2N$	$3N + 1$	5	4
Hough transform	No. of pixels: N No. of parameters: P	$2N + P + 2$	$NP + N + P + 1$	5	4
Threshold	No. of pixels: N	$3N + 1$	$3N$	4	3
Smoothing	8-connected	11	10	3	2
Erosion and dilation	8-connected	10	9	3	2

graph size, simple connectivity properties, and properties of particular vertices. Isomorphism testing against the remaining library graphs uses an approach based on hypergraphs. Attributed hypergraphs have been used for three-dimensional object recognition. For the mapping problem, they are used to reduce the size of regular DGs by letting a single hypergraph node (a hypernode) represent a set of nodes and letting hyperedges represent connectivity between hypernodes.

Figure 5 shows the data-dependency graph and hypergraph for a histogramming algorithm. Table 1 compares the DG sizes and hypergraph sizes for a number of algorithms. The use of hypergraphs does not alter the asymptotic time complexity of graph isomorphism. However, it can yield graphs of such reduced size that the test for isomorphism becomes feasible. For graphs with little or no regular structure, the hypergraph approach may yield no reduction in graph size, and therefore no savings in the isomorphism testing. However, for graphs with regular structure, substantial savings are realized.

We have proven that the heuristic procedure for generating hypergraphs from DGs preserves isomorphism. Under conservative assumptions, the algorithm has worst-case time complexity $O(n^4)$, where n is the number of nodes in the original DG. Under realistic assumptions the complexity is $O(n^2)$.

Graph Matcher has been implemented and tested on the Purdue Image Processing Library. Areas of future work on Graph Matcher include incorporation of techniques to calibrate "closeness" of matches between library and input DGs, and the use of these measures to apply Graph Matcher to high-level-vision algorithms where graph similarity is a more appropriate criterion than graph isomorphism.

Conclusions. Anecdotal reports abound about researchers with scientific and engineering problems who have tried to make use of parallel-processing systems, and who have been almost fatally frustrated in the attempt. Our experience with parallel-algorithm design suggests that the regular structure of many CVIP algorithms can form the basis for an effective specialized parallel-programming environment. Our goal is to exploit the special characteristics of CVIP to achieve easier algorithm development and better performance than can be expected with general-purpose tools. DISC and Graph Matcher and the in-progress Cloner are "proof-of-concept" implementations demonstrating the effectiveness of this specialized environment.■

References

1. Q. Stout, "Mapping Vision Algorithms to Parallel Architectures," *Proc. IEEE*, Vol. 76, No. 8, Aug. 1988, pp. 982-995.
2. F.J. Weil, L.H. Jamieson, and E.J. Delp, "Dynamic Intelligent Scheduling and Control of Reconfigurable Parallel Architectures for Computer Vision/Image Processing," *J. Parallel and Distributed Computing*, Vol. 13, No. 3, Mar. 1992, pp. 273-285.
3. J. Li and L.H. Jamieson, "A System for Algorithm-Architecture Mapping Based on Dependence Graph Matching and Hypergraphs," *Fifth Int'l Parallel Processing Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2167, Apr. 1991, pp. 513-518.

Leah H. Jamieson is a professor of electrical engineering at Purdue University. Her research interests include parallel algorithms; the application of parallel processing to speech, image, and signal processing; and speech recognition.

Edward J. Delp has been a professor of electrical engineering at Purdue University since 1984. His research interests include image coding, medical imaging, ill-posed inverse problems in computational vision, and nonlinear filtering using mathematical morphology.

Chao-Chun Wang is a graduate student in the School of Electrical Engineering at Purdue University. His research interests include parallel processing, parallel algorithms, and distributed-system design.

Juan Li is an advisory engineer with IBM. She has worked primarily in the areas of parallel and distributed processing, fault-tolerant computer systems, and object-oriented design.

Frank J. Weil is a technical leader in the Software Engineering Research Laboratory at Motorola's Software Technology Center. His research interests include software engineering, artificial intelligence, scheduling systems, parallel processing, and speech and image processing.