New computers are supplying the massive power that vision applications require. How to program this hardware efficiently and naturally is the challenge facing the imaging community.

**COMPUTATIONAL APPROACHES**

Computationally, vision and image-processing problems have three levels: image-to-image transformations at the preprocessing level, image-to-features transformations at the intermediate level, and features-to-predicates transformations and manipulation, at the high level.

Many programmers have tried to extend high-level, sequential languages (like Fortran, Pascal, C, and C++ and other object-oriented languages) to handle image processing. All these extensions treat the full image as a data type and use a syntax based on matrix algebra to operate on arrays, which store the images.

Simple array operations, which are frequently implemented as primitives in language extensions, are data reduction (find the maximum value in an array), data mapping or permutation (shift, rotate, transpose), and data broadcasting (add a constant to all pixels). Radically differing paradigms have been suggested for high-level vision and manipulation.

Other approaches focus on query languages that support indexing to better retrieve images and manage the image database.

Finally, approaches that address industrial automation seek to integrate actuator commands with comprehensive vision-analysis software.

**VISION LANGUAGE EVOLUTION**

Machine vision's short history began in the '50s with pattern recognition and has progressed to include image processing. In the last 40 years it has had broad application, from industry to biomedicine to the military, and has already seen four generations of programming environments.

The first generation (1955-65) was essentially an application of Fortran and a few basic routines coded in assembly. Such routines handled I/O from core memory and secondary memory (where images were stored). Basic primitives included the convolution in the spatial domain and the butterfly operator, which implemented the fast Fourier transform in the frequency domain.

The second generation (1965-75) took into account new computing architectures, like cellular logic, for image processing. These environments treated the image as a single datum, not as a set of elements (pixels) ordered in a numerical array. Programmers coded basic operations directly with high-level language constructs (formally described by an image algebra). These environments solved the mismatch between the limited core memory and the correspondingly large secondary memory with special calls to the operating system.

The third generation (1975-85) worked on fine- and coarse-grained architectures...
and pipelined dataflow systems. Fine-grained architectures allowed so-called neighbor operations on all image pixels simultaneously (in single-instruction, multiple-data mode). Coarse-grained architectures had basic constructs, like iteration and recursion (on which most image-processing algorithms are based) embedded in the architecture. However, this approach compromised portability, because the parallel computers had different memory sizes, addressing strategies, and propagation mechanisms. This generation saw the addition of two primitives: a data-selection operation, which locates homogeneous components in an image, and a global-conditional operation, which tests the status of all pixels in an image.

The fourth and current generation (1985–?) addresses the issues of concurrency, multiple-resource management, loading variations, and dynamic dispatching. Today’s environments use many processing modes and so require different programming paradigms, including functional, procedural, object-oriented, and declarative. Their special constructs let the programmer organize the computation hierarchically. Operating systems that let both host and guest multiprocessor architectures manage concurrent processes are very complex because they must manage loosely orchestrated, partially shared system resources. Finally, fourth-generation environments must support the development of graphical user interfaces, a key feature in today’s applications.

IMAGE PROCESSING PROGRAMMING

The latest generation of environments supports the development and execution of image-processing programs more efficiently and considerably faster, but programmers also want to be able to code optimal solutions that exploit systems fully, without needing special skills.

Images involve large amounts of information, only a subset of which is relevant to any one part of the computation. Thus, when attacking an image-processing problem, a programmer may use a layered strategy, with different logical levels to reduce the amount of data to be handled. The approach the programmer chooses influences efficiency, so the combination of parallelization strategies, architecture, and language constructs determines execution speed.

Concurrency implies special system software to manage it, but such software creates an overhead because it must coordinate the activities of different processors and relieve the contention for common resources. This reality has led some researchers to claim that, as a practical matter, speedup is limited to the logarithm of the number of processors.

There are two ways to program parallel processors for computer vision. First, when a task’s parallelism is implicitly apparent, the parallelizing compiler segments the subtasks for each processor (like in a regular mosaic image in which each tile corresponds to a single processor). Second, the programmer can explicitly define the parallelism, allocating specific subtasks (usually those that operate on different regions of interest) to each processor. This distribution is not necessarily even, which may slow execution.

Defining parallelism explicitly requires a good knowledge of both the task and the hardware so that subtasks are optimally scheduled and processors are given equal loads. When it is difficult to evaluate the task’s timing, as it is in high-level vision, overhead rises accordingly. Nevertheless, it is possible to make parallel both the algorithm and data, particularly on low-level tasks, where fixed pixel regularity and cardinality make it easy to partition resources in advance.

On the other hand, a flexible parallelizing approach should accommodate different data dimensions with software, so a single pixel, a connected object, a region of interest, and a full image or set of images may all become operands of the primitives available to the user.

An important feature in characterizing parallel image-processing systems is their granularity, which has been defined as the ratio between communication and computation: A fine-grained program spends more time exchanging data among processors than a coarse-grained one.

The computational nature of the problem strongly conditions the granularity level and the corresponding processes determine the computational weight. Low-level vision tasks correspond to fine-grained systems; high-level vision tasks are coarse-grained. It is diffi-
cult for a single software environment to accommodate these extremes, because fine-grained tasks rely on Boolean constructs and coarse-grained applications rely on message-passing constructs.

ARTICLES

The two articles that follow present different, sometimes conflicting, ideas on new languages and environments to handle computer vision's imposing requirements.

In the first article, "Mechanism to Capture and Communicate Image-Processing Expertise," Bertrand Zavidovique, Veronique Serre, and Christian Fortune outline the problems designers face when they use existing programming environments and languages to solve real-world image-processing tasks.

Zavidovique and his colleagues propose a method for abstracting, storing, and sharing image-processing knowledge that involves decomposing complex tasks into primitives (atoms) and then integrating them into new functional tools (molecules).

In the second article, "Parallel Programming for Computer Vision," Anthony P. Reeves describes Visx and Paragon, two Unix environments he and his Cornell University colleagues developed for programming parallel computers to handle image-processing and vision applications.

Viss is a portable environment for the development of vision applications that has been used for many years on serial computers in research. Reeves has adapted Visx to run on a multiprocessor with modest parallelism by using functional decomposition and standard operating-system capabilities to exploit the parallel hardware.

Paragon is a high-level environment for multiprocessor systems that has facilities for both functional decomposition and data partitioning. It provides primitives that will work efficiently on several parallel processing systems. Moreover, you can use Paragon's primitives to build special image-processing operations, so you can grow your own programming environment naturally. Paragon has been implemented on several testbeds—Reeves reports speedups, data-redistribution strategies, and performance degradations.

WHAT'S AHEAD

Although technological advances have already provided high-speed processors, high-definition monitors, low-cost, huge memories, and various parallel-processing architectures, work remains to be done in:

- algorithms, especially the reconfiguration of known sequential algorithms;
- programming, in which the emphasis must be on reusability and primitives that are easily integrated in different systems;
- environments, in which system and application details must be hidden from the user, to aid program writing, editing, and debugging.

The following articles take steps in these directions, illustrating state-of-the-art image-processing languages and environments.

ACKNOWLEDGMENTS

We thank Renato M. Capocelli, Tadao Ichikawa, Ted Lewis, and Stefano Tarrinton, who made valuable suggestions that helped us select and improve these articles.

Virginia Cantoni is a professor of computer engineering at Pavia University, Italy, and director of its Department of Informatics and Systems Science. He is coordinating a nationwide project to design and construct a pyramidal system for image analysis. His research interests include object recognition and parallel architectures for image processing and computer vision. He has written more than 100 papers.

Cantoni received a PhD with honors in electronic engineering from Pavia University. He is an IEEE senior member.

Stefano Levialdi is a professor of computer science at the University of Rome. He has written more than 100 papers on integrated technology for multiprocessor computers, parallel architectures, high-level languages for image manipulation, and multiprocessor computer vision.

Levialdi received a degree in telecommunications engineering from the Faculty of Engineering, Buenos Aires. He is an associate editor of Pattern Recognition, Signal Processing, Computer Vision, Graphical and Image Processing, Image and Computer Vision, Pattern Recognition Letters, and the Journal of Parallel Computation and Machine Recognition and Applications and a coordinator of Journal of Visual Languages and Computing. He is a vice president of the International Association for Pattern Recognition and an IEEE fellow.

Address questions about this special section to Levialdi at Scienze dell'Informazione, Università di Roma, Via Salaria 133, 00198 Rome, Italy.