NETRA: A Hierarchical and Partitionable Architecture for Computer Vision Systems

Alok N. Choudhary, Janak H. Patel, *Fellow, IEEE,* and Narendra Ahuja, *Fellow, IEEE*

***Abstract*— Computer vision is regarded as one of the most complex and computationally intensive problems. In general, a Computer Vision System (CVS) attempts to relate scene(s) in terms of model(s). A typical CVS employs algorithms from a very broad spectrum such as such as numerical, image processing, graph algorithms, symbolic processing, and artificial intelligence. This paper presents a multiprocessor architecture, called "NETRA," for computer vision systems. NETRA is a highly flexible architecture. The topology of NETRA is recursively defined, and hence, is easily scalable from small to large systems. It is a hierarchical architecture with a tree-type control hierarchy. Its leaf nodes consists of a cluster of processors connected with a programmable crossbar with selective broadcast capability to provide the desired flexibility. The processors in clusters can operate in SIMD-, MIMD- or Systolic-like modes. Other features of the architecture include integration of limited data-driven computation within a primarily control flow mechanism, block-level control and data flow, decentralization of memory management functions, and hierarchical load balancing and scheduling capabilities. This paper also presents a qualitative evaluation and preliminary performance results of a cluster of NETRA.**

***Index Terms*—Computer vision, parallel architectures, parallel algorithms, partitionable architectures, performance evaluation.**

I. Introduction

A. Computer Vision

COMPUTER vision has been regarded as one of the most complex and computationally intensive problems. A Computer Vision System (CVS) employs algorithms from a very broad spectrum such as numerical, signal processing, image processing, graph algorithms, symbolic processing, and artificial intelligence.

A typical CVS using color images requires a processor capable of handling 23 Megabytes of input data per second, interpreting it to construct a three-dimensional model of the environment [8], [38]. An interpretation may require hundreds of objects of different types to be identified [11]. Estimating the motion of and recognizing a moving object from a sequence of time varying images may further involve motion effects and employ a model based recognition in addition to the interpretation needed for static images [7], [9].

Vision researchers have shown that pattern recognition techniques and bottom-up processing alone is not adequate for the above tasks [16]. Vision also involves top-down and knowledge-based processing. Between these two levels of abstraction, another level, known as "intermediate level" is normally introduced. It involves symbolic processing. Symbols range from extracted image characteristics such as edges or regions through perceptually useful groupings such as geometric figures and surfaces. Hence, vision algorithms are normally classified into three levels: low (sensory, image processing), intermediate (symbolic processing), and high (knowledge-based).

B. Architectural Considerations

From a multiprocessor architecture perspective, an image understanding and computer vision tasks' computational requirements can be described considering different abstract levels of processing.

- *Low-Level Processing*—Tasks in this class exhibit massive spatial parallelism which is suitable for both SIMD and MIMD computations. Computations are normally simple and data independent. Computations mainly involve numeric processing and manipulation of simple data structures (such as pixels). Communication requirements are structured. Communication may be local or global in the sense that the output may depend on a spatially local neighborhood of data (e.g., convolution), or it may depend on the entire input image data (e.g., 2D-FFT). Also, communication requires efficient broadcast and synchronization.
- *Intermediate Level Processing*—Computations in this category manipulate symbolic (e.g., tokens) as well as numeric data [39]. Computations are normally data dependent and irregular. They are suitable for medium to coarse grain parallelism. The available parallelism is dynamic and data dependent. Communication patterns can be regular as well as unstructured, depending on the data. Both local and global communication (including broadcasts) are required. Since computations are data dependent, independent decision making capabilities and distributed control are required.
- *High-Level Processing*—Tasks in this level of processing are normally top-down (model directed). Computations require both numeric as well as symbolic processing and

Manuscript received July 16, 1991; revised August 10, 1992. This work was supported in part by National Aeronautics and Space Administration Under Contract NASA NAG-1-613.

A. N. Choudhary is with the Department of Electrical and Computer Engineering, Syracuse University, Syracuse, NY 13244.

J. H. Patel is with the Center for Reliable and High Performance Computing, University of Illinois, Urbana-Champaign, Urbana, IL 61801.

N. Ahuja is with Beckman Institute, University of Illinois, Urbana-Champaign, Urbana, IL 61801.

IEEE Log Number 9213476.

are suitable for MIMD coarse-grain parallelism. Computations are both data and model dependent, and are irregular. Communication is unstructured and irregular. Processors require accesses to shared data (which stores model and image information.) Furthermore, distributed control of processing as well as efficient mechanism to coordinate different activities is needed.

An architecture for CVS's should be capable of performing tasks from all levels of processing efficiently and synergistically. Hence, it needs to be flexible to be able to adapt to the required processing. Furthermore, a flexible communication structure is needed to allow different types of communication among various parts of an architecture. The architecture should allow a partition executing tasks from one level of vision to be reconfigured to perform a task from another level. This requires the architecture to be reconfigurable into the most suitable mode of operation (such as SIMD mode or MIMD mode) for a given task. Real-time vision and high performance requirements dictate that tasks from all levels exist and execute simultaneously in the system, and therefore, the architecture should be divisible into several partitions that can operate independently, yet interact with each other to exchange appropriate data and information. Top-down processing and load balancing requirements suggest a hierarchical and distributed control in the architecture [26]. Time-varying data or different sets of data may represent varying and unevenly distributed load. Therefore, efficient resource allocation, topology and data size independent mapping capabilities, and efficient load balancing capabilities are needed in the architecture. Finally, an architecture for such a complex problem should be modular.

In this paper, we present a parallel architecture called NETRA for CVS's. The architecture was originally proposed by Sharma, Patel, and Ahuja [28]. NETRA is a recursively defined tree-type hierarchical architecture, each of whose leaf nodes consists of a cluster of processors. Processors in a cluster are connected with a programmable crossbar with selective broadcast capability. The internal nodes of the architecture are scheduling processors whose functions are task scheduling, load balancing, and global memory management. The processors in clusters can operate in SIMD-, MIMD- or Systolic-like mode. Other features of the architecture include integration of limited data-driven computation within a primarily control flow mechanism, block-level control and data flow, decentralization of memory management functions, and hierarchical load balancing and scheduling capabilities.

C. Organization

Section II contains a review of architectures proposed for image processing and computer vision. A brief overview of hierarchical, partitionable, and reconfigurable architectures is also presented. Section III presents the architecture of NETRA and describes its components and their functions in detail. In Section IV, NETRA is critically examined with respect to the CVS architectural requirements. Section V contains preliminary results on the cluster performance. Finally, Section VI summarizes the paper.

II. Review of Architectures

A. SIMD Architectures

Massively parallel SIMD multiprocessors are well suited for low-level and well structured vision algorithms that exhibit spatial parallelism at the pixel level. However, such architectures are not well suited for high-level vision algorithms because these algorithms require nonuniform processing, more complex data structures, and data dependent decision making capabilities. Meshes, array processors, hypercubes, and pyramids are some of the most common SIMD parallel processors proposed for image analysis and processing. In meshes, the processing elements are arranged in a square array. Examples of mesh connected computers include CLIP4 [12] and the MPP [3]. The Connection Machine (CM) provides a NEWS network for local communication and a hypercube network for long distance communication [17], [36].

Pyramid architecture was proposed to mimic multidimensional divide-and-conquer computations [1]. However, it was discovered that while the pyramid structure was efficient for a large class of low-level image processing tasks, it was not efficient for higher level tasks [28]. Examples of pyramid architectures include PAPIA [6], SPHINX [24], MPP pyramid [27], and HCL Pyramid [35].

B. Hierarchical/Partitionable/Reconfigurable Architectures

Several hierarchical, partitionable, and reconfigurable architectures have been proposed (and some prototypes built). The following is a brief review of some of these architectures.

TRAC is an experimental reconfigurable array computer proposed for scientific computations [22]. The available resources can be partitioned into several SIMD/MIMD partitions. The partitioning in TRAC is done by setting switches of the interconnection network to partition resources into blocks such that each resource is exactly part of one block.

PASM is a partitionable SIMD/MIMD architecture [31], [13]. PASM can be structured as one or more independent SIMD and/or MIMD machines of various sizes. PASM's multistage network is a generalized cube network. PASM provides hierarchical control. Partitioning of processors and networks is performed by explicitly setting switches to the desired configuration.

IUA (Image Understanding Architecture) has been developed to embed three abstract levels of vision processing into an architecture [39]. It has a hierarchical structure. At the high level, IUA is a MIMD parallel processor. The low level is a Content Addressable Array Parallel Processor (CAAPP) which operates in pure SIMD mode. It also has a reconfigurable mesh with a local broadcast capability. The intermediate level operates in synchronous-MIMD or MIMD mode. Communication and data transfer between different levels is achieved using a shared memory.

Cedar is a multiprocessor architecture with a hierarchical memory structure [20]. Cedar unifies distributed and shared memory paradigms. It consists of multiple clusters (each cluster being a multiprocessor) connected through an omega network to a global memory.

Other proposed multiprocessor architectures that have considered flexibility, partitioning and reconfiguration include CHiP [33], Non-Von [29], and REPLICA [23].

C. Other Architectures

The CMU *Warp* processor [14] is a systolic array machine proposed and built for image understanding and scientific computations. The machine has a programmable systolic array of linearly connected cells. *iWarp* (next in the sequence to Warp) is a two-dimensional systolic and distributed memory architecture considered for image understanding and scientific computations [4]. It supports *memory communication and systolic communication*. Another architecture called "VisTA" (Vision Tri-Architecture) has been proposed for integrated vision systems which attempts to explicitly embed three levels of vision in the architecture [34]. General purpose shared and distributed memory multiprocessors have also been considered and evaluated [37] for image understanding and computer vision.

III. Architecture of NETRA

Fig. 1 shows the architecture of "NETRA." NETRA consists of the following components:

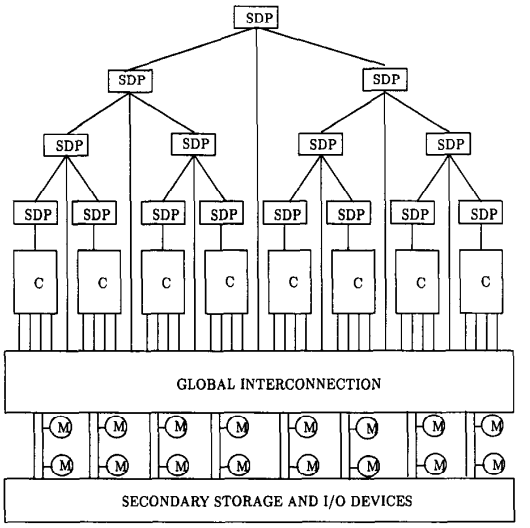
1) A large number (10^2–10^4) of *Processing Elements (PE's)*, organized into clusters of 16 to 64 PE's each.
2) A tree of *Distributing-and-Scheduling-Processors (SDP's)* that make up the task distribution and control structure of the multiprocessor.
3) A parallel pipelined shared *Global Memory*.
4) A *Global Interconnection* that links the PE's and SDP's to the Global Memory.

A. Processor Clusters

The clusters consist of 16 to 64 PE's, each with its own program and data memory. They form a layer below the SDP-tree, with a leaf SDP associated with each cluster. PE's within a cluster also share a common data memory. The PE's, the SDP associated with the cluster, and the common data memory are connected together with a crossbar switch. The crossbar switch permits point-to-point communications as well as selective broadcast by the SDP or any of the PE's.

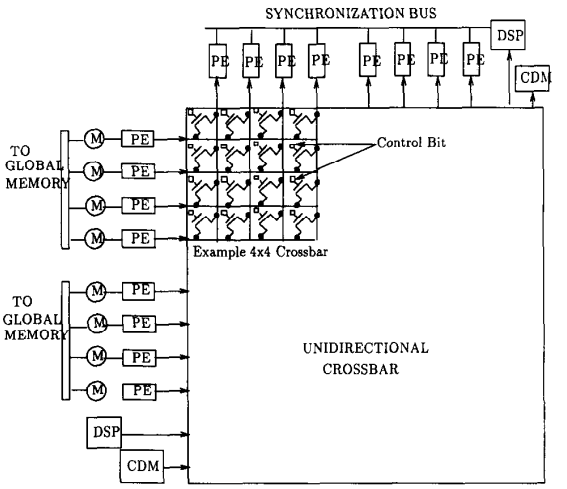
Fig. 2 shows the cluster organization. A 4×4 crossbar is shown as an example of the implementation of the crossbar switch. The switches are controlled by control bits indicating the connection pattern. If a processor of SDP needs to broadcast, then all the control bits in its row are made one. In order to connect processor P_i to processor P_j, control bit (i, j) is set to one and the rest of the control bits in row i and column j are off. Details of the crossbar are discussed later in this section.

Clusters can operate in a SIMD-, a systolic-, or an MIMD-like mode. Each PE is a general purpose off-the-shelf processor. In a SIMD mode, PE's in a cluster execute identical instruction streams from private memories in a lock-step fashion. Since instruction streams are supplied from the PE's private memory (as opposed to being broadcast by a controller), this type of execution represents is SPMD



SDP : Scheduling and Distributing Processor
C : Processor Cluster M : Memory Module

Fig. 1. Organization of NETRA.



PE : PROCESSOR M : LOCAL MEMORY
CDM : COMMON DATA MEMORY

Fig. 2. Organization of processor cluster.

(Single-Program-Multiple-Data) execution in lock-step. In the systolic mode, PE's repetitively execute a set of instruction on data streams from one or more PE's. In both cases, communication between PE's is synchronous. The advantage of providing these two modes of communication is that computations and communications can be overlapped and fine-grain communication among processors can be obtained. In the MIMD mode, PE's asynchronously execute instruction streams resident in their private memories. The streams may be different. In order to synchronize the processors in a cluster, a synchronization bus is provided which is used by processors to indicate to the SDP that a processor(s) has finished its

computation. The SDP can either poll the processors or the processors can interrupt the SDP using the synchronization bus.

1) Crossbar Design: An interconnection pattern between processors must be programmed in the crossbar before processors can communicate with each other. That is, there is no arbitration in the crossbar switch. Programming the crossbar requires writing a communication pattern into the control memory of the crossbar. In the SIMD mode, the SDP alters the communication patterns during the program execution as required by the communication pattern of the computations. In the MIMD mode, a processor can alter the communication pattern by updating the control memory as long as it does not conflict with the existing communication pattern. In case of conflicts, the SDP is responsible to resolve them. The SDP associated with the cluster can write into the control memory to alter the communication pattern. The most common communication patterns such as linear arrays, trees, meshes, pyramids, shuffle-exchanges, cubes, and broadcast can be stored in the memory of the crossbar. These patterns need not be supplied externally. Therefore, switching to a different pattern in the crossbar is fast because switching only requires writing the patterns into the control bits of the crossbar switches from its control memory.

The advantages of such a crossbar design are the following: Firstly, since there is no arbitration, the crossbar is faster than one which involves arbitration because switching and arbitration delays are avoided. Secondly, switches are simple, easy to design and implement because arbitration is absent. Such a crossbar is easily scalable. Unlike other interconnections (such as cubes, shuffle-exchanges etc.), the scalability need not be in powers of 2. A unit scalability is possible. In other words, it is possible to provide just one more processor and link in the crossbar, which can replace any other processor and link upon a failure. Hence, it is easy to provide fault-tolerance in a cluster. This is possible because there is no inherent structure that connects the processors. Each processor (link) is topologically equivalent to any other processor (link). Finally, the most commonly used communication patterns can be stored in the on-chip memory of the crossbar. That allows a single-cycle parallel load of a new pattern, and therefore, switching to a new pattern can be achieved in one cycle.

2) Crossbar Implementation: This crossbar design has been implemented and is currently being tested [32]. The crossbar chip is a 2-bit-sliced 8×8 crossbar fabricated using 2.0 μm CMOS technology with a die size of 4402 by 6602 μm This was packaged in a standard 64 pin DIP. The output ports are designed to be set to high impedance so that the chip can function as a building block for larger sized crossbars. Commands sent to the crossbar's opcode input pins instruct the crossbar as to which input ports are to be connect to which output ports. Opcodes exist for setting individual connections, pairs of connections, and connecting a single input to all eight output ports (i.e. the crossbar is set up for broadcast). Each time a new connection is made to an output port, the previous connection to that output port is over written.

The crossbar is capable of storing on chip the state of the input to output port connections for later access. The stored state can then be returned to through the execution of a single opcode. Storing current connections state is also done in a single instruction. Up to eight sets of connections can be stored at any one time. Each set is made up of eight subsets. The subsets consist of the address of one output port and one input port. Each subset's output port address is distinct. It should be noted that the number of patterns that can be stored on-chip depends on the amount of space available on the chip for memory. Given the current VLSI technology, enough memory can be put on the chip to store thousands of patterns on the crossbar itself. Since the first implementation was done to test the proof of concept, only a limited amount of memory was put on chip. Further, it should be noted that the size of the memory on chip is not a limitation on how many patterns the crossbar can allow, because the chip allows one to supply a new pattern externally. The only drawback is that it is much slower than using a pattern already stored on chip.

The chip is provided with a programmable chip address so that the opcode can be registered off a common bus. This functions as a chip select when compared with an incoming address off an address bus. If the address off the address bus does not match that of the chips, the registered opcode is ignored and a *no-op* is performed. The crossbar's chip address is set via scan. Because the address is 5 bits wide it will take five scan operations to set the chip's address. This is normally done at boot time.

Two additional operations not specified through the opcode, that the crossbar is capable of performing, are reset and test. There is a reset pin that when activated, tri-states all output ports and sets the chip address to 16. Reset is normally executed at boot time. Test is another operation that has a dedicated pin. When this signal is active several internal signals are routed to output pins to provide internal visibility during test.

A larger crossbar can be obtained using a concatenation of smaller crossbars. The crossbar chip has been built to provide this scalability. For example, four 32 × 32 crossbar chips can be used to obtain a 64 × 64 crossbar. However, if the crossbar becomes very large, it becomes difficult to support single-cycle transfer of data, or the cycle time must be increased, thereby reducing the bandwidth.

Some other architectures have employed programmable crossbar switches, most notably among them are the GF11 [18] and ICAP communication switch [25]. The GF11 was primarily designed for QCD (Quantum Chromodynamics) computations. The GF11 employs a three stage Benes network which connects 576 processors. The main switch of the network is a 24 × 12 one bit wide crossbar. Each node of the switch consists of a 24 × 24, nine bit crossbar. Each node contains memory to store 1024 switch settings. Before a job is run, the appropriate switch settings suitable for the problem are loaded. This programmability allows the switch settings to be changed after each word transfer, which takes four cycles. Although the GF11 crossbar switch is very similar to our crossbar switch, our switch allows both horizontal and vertical expansion, thereby allowing us to build larger crossbars from smaller chips.

The ICAP switch (ICAP is the intermediate level processor of the IUA architecture described earlier) prototype is a 64×64 network which uses 32×32 bit serial crossbar chips (called PARCOS). The chip consists of an on-chip control memory capable of storing 32 configurations. Therefore, the chip can hold up to 32 most frequently used connection patterns. Changing the connection pattern (which uses one of the patterns stored on the chip) requires a single write instruction with the address of the new pattern in the control memory. In this respect, the switch is similar to our crossbar. But in its usage to build larger networks, this switch is closer to the GF11 switch. Also note that, like our design, the number of on-chip control words is not a limitation on how many patterns can be used in the PARCOS chip. It allows loading of new patterns externally, when needed.

3) Significance of Crossbar for Reconfigurability: Several techniques for implementing reconfigurability between a set of PE's were studied [10], [30]. It was concluded that using a crossbar switch to connect all PE's was simpler than any other scheme. When designing communication networks in VLSI, the primary constraint is the number of pins and not the chip area. The number of pins is governed by the number of ports on the network and is independent of the type of network. Furthermore, it was realized that a crossbar with a selective broadcast capability was not only a very powerful and flexible structure, but was also simpler, scalable, and less expensive. However, it must be noted that the crossbar is centralized and tightly coupled. But at the same time, such a design allows single cycle data transfer across the crossbar. Therefore, we have limited a cluster size to 64 processors. It must be noted that a NETRA configuration containing 256 processors (4 clusters of 64 processors) will have 4 crossbars of size 64 and not 16 crossbars of size 64.

B. The SDP Hierarchy

The SDP-tree is an n-tree with nodes corresponding to SDP's and edges to bi-directional communication links. Each SDP node is composed of a processor, a buffer memory, and a corresponding controller.

The tree structure has two primary functions. First, it represents the control hierarchy for the multiprocessor. A SDP serves as a controller for the subtree structure under it. Each task starts at a node on an appropriate level in the tree, and is recursively distributed at each level of the subtree under the node. At the bottom of the tree, the subtasks are executed on a processor cluster in the desired mode (SIMD or MIMD) and under the supervision of the leaf SDP.

The second function is that of distributing the programs to leaf SDP's and the PE's. Low-level vision algorithms are characterized by a large number of identical parallel processes that exploit spatial parallelism and operate on different data sets. For global algorithms such as connected component labeling, multidimensional divide-and-conquer can be used. It involves two phases, 1) computations within partitions (e.g., labeling within partitions) and merging the partial results. The first phase involves execution of the same program for each processor on different data sets. The second phase involves execution of programs that merge partial results. The body of the programs is normally the same. The difference occurs in execution where the control flow is data dependent. It would be highly wasteful if each PE issued a separate request for its copy of the program block to the global memory because it would result in unnecessarily high traffic through the interconnection network. Under the SDP-hierarchy approach, one copy of the program is fetched by the controlling SDP (the SDP at the root of the task subtree) and then broadcast down the subtree to the selected PE's. Also, SDP hierarchy provides communication paths between clusters to transfer control information or data from one cluster to others. The SDP-tree is also responsible for Global Memory management.

The SDP hierarchy provides a hierarchical control and resource and process management functions, which is specifically useful for high-level vision algorithms. High-level vision exhibits functional parallelism where tasks have a loose coupling. For example, a collection of tasks may work on obtaining the best match of hypotheses with the models. A SDP controlling a cluster can dynamically schedule these tasks as and when necessary. Since the outcome of such computations is normally nondeterministic, and computations change depending on the data, scheduling and resource allocation cannot be done in advance. A SDP, therefore, can perform efficient resource management, scheduling, and coordination functions by controlling the initiation and execution of the tasks from its task queues.

C. Global Memory

The multiport global memory is a parallel-pipelined structure as introduced in [5]. Given a memory-access-time of T processor-cycles, each line has T memory modules. It accepts a request in each cycle and responds after a delay of T cycles. Since an L-port memory has L lines, the memory can support a bandwidth of L words per cycle.

Data and programs are organized in memory in *blocks*. Blocks correspond to "units" of data and programs. The size of a block is variable and is determined by the underlying tasks, their data structures, and data requirements. A large number of blocks may together constitute an entire program or an entire image. Memory requests are made for blocks. The PE's and SDP's are connected to the Global Memory with a multistage interconnection network. Each line also incorporates a secondary storage device, thus supporting a large paged virtual memory.

The global memory is capable of queueing requests made for blocks that have not yet been written into. Each line (or port) has a Memory-line Controller (MLC) which maintains a list of read requests to the line and services them when the block arrives. It maintains a table of *tokens* corresponding to blocks on the line, together with their length, virtual address, and *full/empty* status. The MLC is also responsible for virtual memory management functions.

Two main functions of the global memory are: input-output of data and program, to and from the SDP's and processor clusters; to provide intercluster communication between various tasks as well as within a task if a task is mapped onto more than one cluster.

D. Global Interconnection

The PE's and the SDP's are connected to the Global Memory using a multistage circuit-switching interconnection network. Data is transferred through the network in pages. A page is a unit of data or instruction. A page is transferred from the global memory to the processors which is given in the header as a destination port address. The header also contains the starting address of the page in the global memory. When the data is written into the global memory, only the starting address must be stated. In each case, end-of-page may be indicated either by using an extra flag bit appended to each word (which may be expensive but is the most flexible), or by containing the length of the page in the header (which requires a capability to count, and therefore, additional logic, in the MLC).

IV. CVS Requirements and Architectural Features of NETRA

A. Reconfigurability (Computation Modes)

The clusters in NETRA provide SIMD, MIMD, and systolic capabilities. It is important to provide these modes of operations in a multiprocessor system for CVS's so that processor configuration can be adapted to suit the best implementation for each algorithm. Consider matrix multiplication. We will show how it can be performed in SIMD and systolic modes. Let us assume that the computation requires obtaining the matrix $C = A \times B$. For simplicity, let us assume that the cluster size is P and the matrix dimensions are $P \times P$. In general, any arbitrary size computation can be performed independent of the data or cluster size.

1) SIMD Mode: The algorithm can be mapped as follows. Each processor is assigned a column of the B matrix, i.e., processor P_i is assigned column B_i ($0 \leq i \leq P-1$). The SDP broadcasts each row to the cluster processors which compute the inner product of the row with their corresponding column in lock-step fashion. Note that the elements of the A matrix can be continuously broadcast by SDP, row by row without any interruptions, and therefore, efficient pipelining of data input, multiply, accumulate operations can achieved. Fig. 6(a) illustrates a SIMD configuration of a cluster. The following pseudo code describes the SDP and processor (P_k's program, $0 \leq k \leq P-1$) program.

SIMD Computation SDP	P_k
1. FOR i=0 to i=P-1 DO	1. -
2. connect(SDP,P_i)	2. NO-OP
3. out(column B_i)	3. in(column B_i)
4. END_FOR	4. -
5. connect(SDP, all)	5. NO-OP
6. FOR i=0 to i=P-1 DO	6. $c_{ik} = 0$
7. FOR j=0 to j=P-1 DO	7. FOR j=0 to j=P-1 DO
8. out(a_{ij})	8. in(a_{ij})
9. END_FOR	9. $c_{ik} = c_{ik} + a_{ij} \times b_{jk}$
10. END_FOR	10. END_FOR

In the above code, the computation proceeds as follows. In first three lines, the SDP connects with each processor through the crossbar and writes the column (one word at a time) on the output port. That column is input by the corresponding processor. In statement 5, the SDP connects with all the processors in a broadcast mode. Then from statement 6 onwards, the SDP broadcasts the data from matrix A in row major order and each processor computes the inner product with each row. Finally, each processor has a column of the output matrix. It should be mentioned that the above code describes the operation in principle, and does not give exact timing of operations.

2) Systolic Mode: The same computation can be performed in systolic mode. Fig. 3 illustrates a linear systolic configuration of a cluster. The SDP can reconfigure the cluster in a circular linear array after distributing columns of matrix B to processors as before. The SDP is not shown in the figure. The SDP assigns row A_i of matrix A to processor P_i. Each processor computes the inner product of its row with its column. At the same time, a processor writes the element of the row on the output port. This element of the row is input to the next processor (through the programmed crossbar connections). Therefore, each processor receives the rows of matrix A in a systolic fashion and the computation is performed in a systolic fashion. Note that the computation and communication can be efficiently pipelined. In the code, statements 7–10 illustrate the systolic computation. Each element of the row is used by a processor and immediately written on to the output port. At the same time, the processor receives an element of the row of the previous processor (in the circular linear array) on its input port. Therefore, every P cycles a processor computes a new element of the C matrix.

Systolic Computation SDP	P_i
1. FOR i=0 to i=P-1 DO	1. -
2. connect(SDP,P_i)	2. NO-OP
3. out(column B_i)	3. in(column B_i)
4. out(row A_i)	4. in(column A_i)
5. END_FOR	5. -
6. connect(P_i to $P_{i+1} mod P$)	6. $c_{ii} = 0$
7. -	7. FOR j=0 to j=P-1 DO
8. -	8. $c_{ii} = c_{ii} + a_{ij} * b_{ji}$
9. -	9. out(a_{ij}), in(a_{i-1j})
10. -	10. END_FOR
11. -	11. repeat 7–10 for each new row

B. Partitioning and Resource Allocation

There are several tasks with vastly different characteristics in a CVS. The required number of processors for each task may be different as well as each task may need a different computational mode and partition. Hence, partitionability and dynamic resource allocation are keys to high performance.

Partitionability of interconnection networks has been studied by many researchers [40], [30], [22], [10], [19]. These approaches are, however, relevant only to systems of tightly

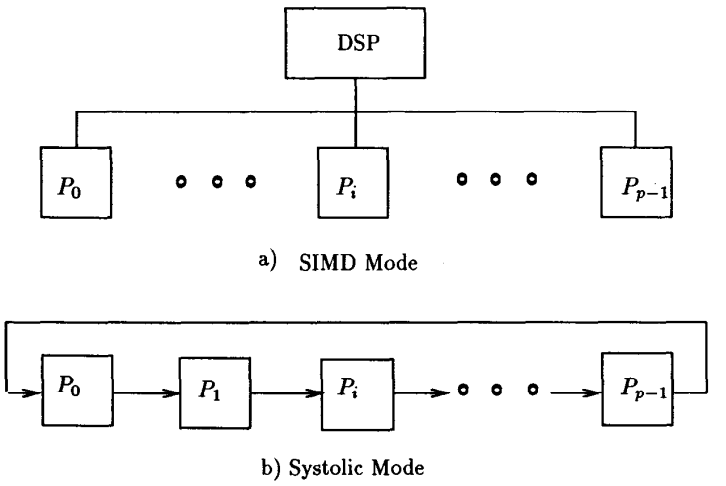


Fig. 3. An example of SIMD and systolic modes of computation in a cluster.

coupled processes wherein tasks require specific interconnection patterns. In the above cases, links are reserved for specific point-to-point communication while a process executes. Whenever a new process is instantiated, the required resources should be free and linked together in a specified manner. A partition is, in effect, isolated from the rest of the system.

Partitioning in NETRA is achieved as follows. When a task is to be allocated, the set of subtrees of SDP's is identified such that the required number of PE's is available at their leaves. One of these subtrees is chosen on the basis of load balancing (discussed later), locality considerations, and characteristics of the task. The chosen SDP represents the root of the control hierarchy for the task. Together with the SDP's in its subtree, it manages the execution of the task. Once the subtree is chosen, the processes may execute in SIMD, MIMD, or systolic mode when they get to the head of the Ready Queues at the PE's or clusters. Further, MIMD processes may exhibit widely varying execution times as processing required often depends on input data characteristics. If rigid partitions are used, processors would have to wait until all complete processing before they start executing another task. Finally, locality is maintained within the control hierarchy, which limits the intratask communication to within the subtree.

Since all tasks are assigned in this manner, the partitioning is only virtual. The PE's are not required to be physically isolated from the rest of the system. Therefore, unlike physical partitioning of a network, in the above approach, communication and data exchange is possible between tasks operating in different partitions. For example, suppose there are two tasks in the system executing on different partitions, one working on matching models of objects to the models developed from the image data (a high-level vision task), and the other working on probing the image data to resolve disambiguities (using low-level vision tasks, e.g., Hough transform) [38]. For normal operation both tasks can execute within their respective partitions. But they need to provide feedback to each other as the computation progresses. If partitions are isolated, it will be very difficult to achieve this cross-communication between tasks of two partitions. TRAC architecture provides "shuttle-memory" which can be used for such communications [22], [21]. In NETRA, cross-partition communication is provided through shared memory for partitions on different clusters, and through the common data memory if different partitions are on the same cluster.

C. Flexible Communication

Availability of flexible communication is critical to achieving high performance. For example, when a partition operates in a SIMD mode, there is a need to broadcast the programs. When a partition operates in an MIMD mode, where processors in the partition cooperate in the execution of a task, one or more programs need to be transferred to the local memories of the processors. Performing the above justifies the need for selective broadcast capability. In order to take advantage of spatial parallelism in vision tasks, processors working on neighboring data need to communicate quickly amongst themselves in order to obtain high performance. The programmability and flexibility of the crossbar provides fast local communication. A large number of vision algorithms need a broad range of processor connectivities for efficient execution. These connectivities include arrays, pipelines, several systolic configurations, shuffle-exchanges, cubes, meshes, pyramids etc. Each of these connectivities may perform well for some tasks and badly for others. Therefore, using a crossbar with a selective broadcast capability, any of the above configurations can be achieved, and consequently, optimal performance can be achieved within clusters.

The need for global communication is relatively low and infrequent. Global communication is needed for intertask communication in a CVS executing on different clusters. It is also needed to input and output data, to transfer data within subtasks of a task when a task is executed on more than one cluster, and finally, it is needed to load programs. The global communication is performed through the global

```
Compute_System_Load;

  If ROOT_SDP
    System_Load = 0;
    For i = 1 to num_child(ROOT_SDP) do /* num_child is the number of children of the
ROOT_SDP */
      Receive Load[child(i)];
      System_Load = System_Load + Load[child(i)];
    End_For
    Compute Average_Load;
    Broadcast Average_Load;
  Else If LEAF_SDP /*SDP associated with one cluster*/
    Send Cluster_Load to Parent SDP;
    Receive Average_Load;
  Else /*Internal SDP*/
    Sub_Tree_Load = 0;
    For i = 1 to num_child(This_SDP) /* num_child is the no.  of children of this
SDP*/
      Receive Load[child(i)];
      Sub_Tree_Load = Sub_Tree_Load + Load[child(i)];
    End_For
    Send Sub_Tree_Load to Parent SDP;
    Receive Average_Load;

End Compute_System_Load.
```

Fig. 4. Algorithm for periodic computation of system load.

memory using the interconnection network. Global memory is used for coarse-grain communication where data is transferred in blocks as described below.

D. Load Balancing and Task Scheduling

In NETRA, a hierarchical load balancing scheme is proposed. Here, a "load balancing system" executes on the SDP-tree as a hierarchy of identical processes. Two levels of load balancing are employed, namely, *global load balancing* and *local load balancing*. Global load balancing aids in partitioning and allocating the resources for tasks as discussed earlier. Local load balancing is used to distribute computations (or data) to processors executing parallel subtasks of a task. Local load balancing is a centralized scheme in which the cluster SDP is responsible for local load balancing. The local load balancing can be either static or dynamic or a combination of both.

Using the information from local load balancing and other measures of computations, global load balancing is achieved hierarchically by using the SDP hierarchy. A similar approach has been proposed in [15]. In this scheme, each controller SDP maintains the following.

1) A measure of load on the subtree below it. For example, an average number of processes in the Active Queues of PE's and cluster in the subtree can serve as the measure.
2) A measure of average load over the entire system. This is computed periodically over the entire tree and broadcast to all the SDP's.

Each SDP sends its measure of load to its parent SDP and the root SDP receives the load information for the entire system. The root SDP then broadcasts the measure of load of the entire system to the SDP's. The procedure for periodic computation of system load is illustrated in Fig. 4. When a task is to be allocated, these measures can be used to select a subtree for its execution as follows:

If any subtree corresponding to the child of the current SDP has an adequate number of processors, then the task is transferred to a child SDP with the lowest load, else if the current subtree has enough resources and the load is not significantly greater than the average system load, then the task is allocated to the current subtree, else the current SDP transfers the task to the parent SDP.

In NETRA, a SDP is not confronted with a large volume of information to schedule a task since it needs to consider the average load on the subtree below it and the overall average load of the system. In systems like PASM, REPLICA or PM4, fragmentation can be minimized only if scheduling is static or done considerably in advance of execution. This is because scheduling would involve global considerations such as partitionability of the network and availability of resources. However, since the scheduler cannot determine in advance, what resources will be available at a later time, processes cannot be easily prescheduled. NETRA allows for easy allocation of dynamically created tasks because they are generated on the basis of load balancing and locality considerations alone.

E. Block-Level Data and Control Flow

A CVS system consists of a collection of tasks, each of which can be executed in parallel in an SIMD, systolic or MIMD mode over a number of processors. Each task can be considered a functional block. A *Functional Block* thus corresponds to a block of instructions executed on one or more clusters, copies of which are broadcast to several PE's to be executed as a set of distributed processes. Similarly, data is also organized as *Data Blocks*, which represent "units" of data. For example, in a graph matching algorithm, a record containing all the information about one node can be considered a block.

Each function block requires one or more input data blocks and produces one or more output data blocks. *Tokens* are used to specify both function blocks and data blocks. A token is composed of the following fields:

$$< \text{Job ID} > < \text{Task ID} > < \text{Block Number} >$$

Since the token corresponding to given tasks will differ in their less significant bits, these bits are used to specify port numbers for the global memory. Blocks corresponding to a task are, thus, uniformly distributed over the global memory, and therefore, can be accessed with minimal conflicts.

Processes are explicitly assigned to clusters and PE's. When a task is to be executed in an SIMD-like mode on a cluster, the corresponding token is sent to the leaf SDP controlling the cluster. For MIMD tasks, tokens are assigned to the individual PE's. Tokens corresponding to the tasks are, however, transferred only to an *Active Queue* at the PE or the leaf SDP controlling the cluster. Function blocks are broadcast to the selected PE's or clusters by parent SDP's. Simultaneously, requests for input-data-blocks are issued. The tokens are moved to the Ready Queue only after all input-data-blocks are available at the PE or cluster. An explicit control flow scheme is used here because we believe that at the function level, control flow is simple and data dependencies are easily recognized. An interesting approach is that of combining explicit control flow and block level data flow schemes. The memory can queue requests for "empty" blocks and service them when blocks are "full."

F. Intelligent Memory

NETRA requires that a PE or a cluster issue requests for input-data-blocks as soon as a process token enters its Active Queue. The required data may not be available in the global memory at that time. Instead of waiting for the data, the processor should proceed with tasks already in the Ready Queue. Therefore, the memory should be capable of queueing requests and responding when data is available.

The scheme employed is similar to the I-structure storage technique used for dataflow computers [2]. Each block has associated with it a *full/empty* bit. The bit is set to 1 if the required data has been written into the page and is set to 0 otherwise. When a request is made for the block, this bit is examined. If the block is marked full, the request is serviced; otherwise it is queued. There is a controller on each line called Memory Line Controller (MLC). MLC is responsible for accepting requests, queueing them if required, and selecting them for service when appropriate. For this purpose, the MLC maintains a table containing the following information for each block on the line:

< Virtual Address > <Length> <full/empty status >

Higher order bits of the block are used to index into the table. Lower order bits are used to select the global memory port.

G. Distributed Memory Management

The task of managing the global memory is distributed over the SDP-tree and the MLC's. Two factors greatly simplify the memory management task. First, the blocks are distributed over the memory ports by using LSB's of tokens to select the port. This represents block level interleaving of data. For low-level vision algorithms, where equal blocks of data are assigned to tasks, data blocks corresponding to a task are expected to be similar in size, this distribution is expected to be very even. For other tasks such as high-level vision tasks, where the distribution of data sets may not be even, interleaving in the manner described above scatters the data uniformly among memory modules, thereby reducing the correlation in access patterns Second, a large locally managed virtual space is provided at each port. The local controller is free to place a block anywhere in its virtual space. Specifically, blocks corresponding to the same task may be allocated in contiguous virtual space. A request for allocation of storage for data blocks is made by the SDP that initiates a task. When the task is complete, requests for deallocation are made.

H. Ability to Tolerate Large Memory Access Latency

A large multiprocessor implies that response times to memory requests can be large and variable in a nondeterministic manner due to conflicts. Therefore, it is required that PE's in such a multiprocessor be able to issue multiple requests in advance and accept responses out of order.

NETRA is a multiprogrammed system with a large number of processes active at any time. A process becomes active when a token corresponding to the process is entered into the Active Queue of a PE (MIMD mode) or a cluster (SIMD-like mode). Data requests for the input-data-blocks are immediately issued. When all input-data-blocks are available, it is transferred to the Ready Queue. However, while these requests are being serviced, the PE continues to execute processes already in its Ready Queue. Access to memory for one process is thus overlapped with execution of another.

V. Preliminary Results on Cluster Performance

In this section we present initial performance results based on the implementation of some algorithms on a cluster simulator. The total processing time for a parallel algorithm consists of the following components: Program load time onto the cluster processors (t_{pl}), data load and partitioning time (t_{dl}), computation time of the divided subtasks on the processors (t_{cp}), which is the sum of the processing time on a processor P_i and intra-cluster communication time (t_{comm}), and the result report time (t_{rr}). t_{dl} consists of three components: 1) data read time from the global memory (t_r) by the cluster SDP, 2) crossbar switch setup time (t_{sw}) and, 3) the data broadcast and distribution time onto the cluster processors (t_{br}). The total processing time $\tau(P)$ of the parallel algorithm on a P processor cluster is given by

$$\tau(P) = t_{pl} + t_{dl} + t_{cp} + t_{rr} \tag{1}$$

where,

$$t_{dl} = t_r + t_{sw} + t_{br}. \tag{2}$$

If the computation and communication do not overlap then,

$$t_{cp} = \max_{1<=i<=P} t_{Pi} + t_{comm} \tag{3}$$

else if computation and communication can completely overlap then,

$$t_{cp} = \max\left(\left(\max_{1<=i<=P} t_{Pi} \right), t_{comm} \right). \tag{4}$$

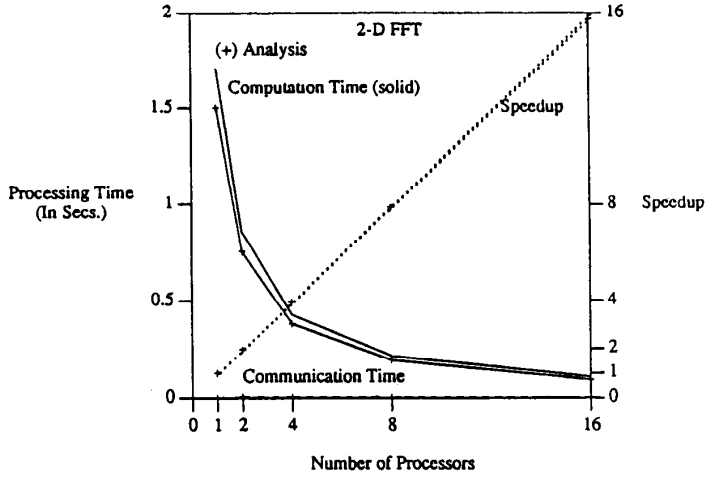


Fig. 5. Performance of 2D-FFT on a cluster.

In the above equations, t_r depends on the effective bandwidth of the global interconnection network.

A. Cluster Simulator

A cluster was simulated on the Intel iPSC/2 hypercube multiprocessor. In order to obtain accurate computation results, the processors (Intel 80386 processors) of the hypercube were used as the cluster processors. The crossbar communication was explicitly simulated in which communication cost was computed based on the amount of data transferred between nodes. Note that since there are no conflicts in the crossbar (because the switch must be set before communication), the amount of data transferred between nodes can provide sufficient information to simulate the communication in the crossbar. Each crossbar link was assumed to provide 20 Megabytes/s using 8-bit wide data paths. Details are presented in [9].

B. Two-Dimensional Fast Fourier Transform

Two-Dimensional Fast Fourier Transform (2D-FFT) was implemented on a 16 processor cluster simulator. For a P processor cluster and $N \times N$ image, the steps of the algorithm were as follows: 1) Each processor was assigned N/P rows of input data. Each processor computed the one-dimensional transform of each row of its own partition. 2) The intermediate results were transposed by all processors communicating with each other. This step required $P - 1$ switch settings of the crossbar. 3) Each processor computed the column transform on the intermediate results producing the 2D-FFT of the input image. It should be noted that in step 2) each switch setting permits P parallel communications. Hence, the entire transpose can be achieved in $P - 1$ distinct switch settings.

Fig. 5 shows the performance results for a 2D-FFT on a cluster varying in size up to 16 processors. Both analytical and implementation results are shown. As we can observe, analytical and implementation results are very close to each other. Almost linear speedups can be obtained for the 2D-FFT. Fig. 6 shows the communication time as a function of number of processors in a cluster. An important observation from the figure is that the communication time decreases as the number of processors increases. This is very important to obtaining almost linear speedups. The communication time decreases as a function of number of processors because there are no conflicts in the crossbar for the transpose phase of the algorithm, and hence, as the number of processors increases, each processor communicates smaller amount of data in each switch setting. Specifically, each processor communicates $(P - 1) \times N^2/P^2$ amount of data in the transpose phase. Therefore, since there are no conflicts in communication, the communication time is a decreasing function of the number of processors.

C. Median Filtering

Median filtering of an image using a $w \times w$ filter involves replacing each pixel of the image with the median of its $w \times w$ neighborhood window. Median filtering was implemented in the MIMD mode on the test data provided with the DARPA Image Understanding Benchmark [37]. Table I shows the performance results for the data set "test" of the benchmark. Each component of program execution such as processing time, data load time, result output time, program load time, and total time is shown. It can be observed that almost linear speedups can be obtained after incorporating all the overheads of various phases of program execution.

D. Sobel Edge Detection

Sobel edge detection was also implemented using the data from the Image Understanding Benchmark. Sobel edge detection essentially involves computing a 3×3 convolution of the image. The results are shown when implementation on the simulator is done in a SIMD mode, but computation and communication are not overlapped. Table II shows the performance results for the data set "test." Only sublinear speedups are obtained for sobel edge detection. This occurs due to the following reason. The amount of computation

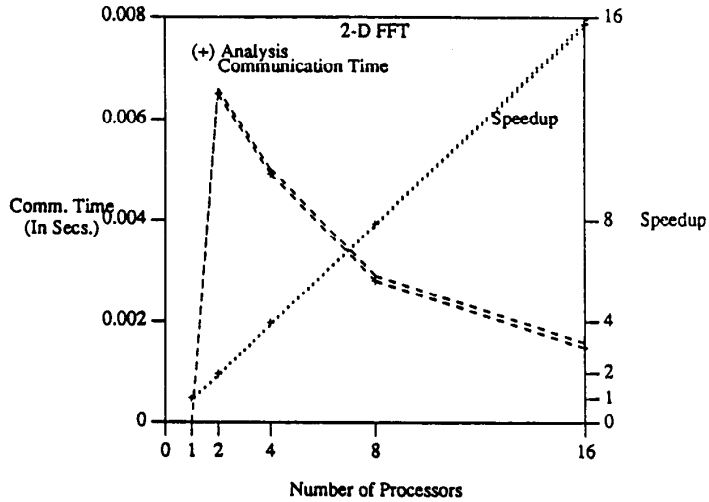


Fig. 6. Communication time for 2D-FFT on a cluster.

TABLE I
PERFORMANCE FOR MEDIAN FILTERING

Median Filtering (Test)							
No. Proc.	Proc. Time(sec.)	Data load Time(Sec.)	Result Output Time(sec.)	Prog. Load Time(sec.)	Data Input Time(sec.)	Total Time(sec.)	Speed up
1	60.36	0	0	0	0.008	60.37	1
2	30.17	0.056	0.056	0.001	0.008	30.30	1.99
4	15.19	0.056	0.056	0.001	0.008	15.31	3.94
8	7.72	0.056	0.056	0.001	0.008	7.85	7.70
16	3.99	0.056	0.056	0.001	0.008	4.11	14.68
32	1.90	0.056	0.056	0.001	0.008	2.02	29.93

TABLE II
PERFORMANCE FOR SOBEL EDGE DETECTION

Sobel (Test)							
No. Proc.	Proc. Time(sec.)	Data load Time(Sec.)	Result Output Time(sec.)	Prog. Load Time(sec.)	Data Input Time(sec.)	Total Time(sec.)	Speed up
1	4.04	0	0	0	0.008	4.05	1
2	2.02	0.056	0.014	0.001	0.008	2.1	1.92
4	1.01	0.056	0.014	0.001	0.008	1.09	3.70
8	0.51	0.056	0.014	0.001	0.008	0.589	6.91
16	0.26	0.056	0.014	0.001	0.008	0.33	12.13
32	0.13	0.056	0.014	0.001	0.008	0.21	19.71

per pixel is small, and amount of computation per processor decreases linearly as the number of processors decreases. At the same time, other measures such as data load time, data input-output time remain constant, and hence, the overhead as a fraction of total time increases. Further details are presented in [9].

VI. SUMMARY AND CONCLUSIONS

NETRA is a hierarchical and partitionable architecture for computer vision systems. NETRA is a recursively defined tree-type hierarchical architecture whose leaf nodes consist of cluster of processors connected with a programmable crossbar with selective broadcast capability to provide for desired flexibility. We presented a qualitative evaluation of NETRA. The programmable crossbar has been implemented and is currently being tested. Furthermore, some preliminary results on the performance of a cluster of NETRA were presented using 2D-FFT, median filtering, and sobel edge detection algorithms. We have done extensive performance evaluation of clusters as well as inter-cluster communication of NETRA. The details are presented in [9].

NETRA also provides a control hierarchy that can be employed to develop heterogeneous architectures for computer vision systems. In such architectures, some clusters can be replaced by special purpose processors (as briefly proposed below) and machines to perform specific tasks efficiently. Most parallel architectures provide a host interface and an attached multiprocessor. The leaf SDP can provide the functions of the a host processor with all the responsibilities described in the paper. Therefore, a mix of special purpose processors and clusters proposed in the paper can synergistically provide a powerful and flexible architecture in which the SDP hierarchy will provide a hierarchical control and system management functions.

ACKNOWLEDGMENT

We would like to thank M. Sharma and J. Simonson for their contributions. We would also like to thank the referees for their comments which have helped improve this paper.

REFERENCES

[1] N. Ahuja and S. Swamy, "Multiprocessor pyramid architectures for bottom-up image analysis," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-6, pp. 463–475, July 1984.

[2] Arvind and A. Ianucci, "A crtique of multiprocessing von Neumann style," in *Proc. 10th Int. Symp. Comput. Architecture,* 1983, pp. 426–437.

[3] K. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.,* vol. C-29, pp. 836–840, 1980.

[4] S. Borkar *et al.*, "Supporting systolic and memory communication in iwarp," in *Proc. 17th Annu. Int. Symp. Comput. Architecture,* May 28–31, 1990, pp. 70–81.

[5] F. A. Briggs and E. S. Davidson, "Organization of semiconductor memories for parallel-pepelined processors," *IEEE Trans. Comput.,* pp. 162–169, Feb. 1977.

[6] V. Cantoni, S. Levialdi, M. Ferretti, and F. Maloberti, "A pyramid project using integrated technology," *Integrated Technology for Parallel Image Processing,* pp. 121–132, 1985.

[7] A. N. Choudhary and J. H. Patel, *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems.* Boston, MA: Kluwer Academic, 1990.

[8] A. N. Choudhary and S. Ranka, "Parallel processing for computer vision and image understanding," *IEEE Computer,* vol. 25, no. 2, pp. 7–10, Feb. 1992.

[9] A. N. Choudhary, "Parallel architectures and parallel algorithms for integrated vision systems," Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, Aug. 1989.

[10] D. Degroot, "Partitioning job stractures for SW-banyan networks," in *Proc. Int. Conf. Parallel Processing,* 1979, pp. 106–113.

[11] B. A. Draper *et al.*, "The Schema System," *Int. J. Comput. Vision,* vol. 2, pp. 207–218, Jan. 1989.

[12] M. J. B. Duff, "CLIP 4: A large scalc integratcd circuit array parallcl processor," in *Proc. IEEE Int. Joint Conf. Pattern Recognition,* Nov. 1976, pp. 728–733.

[13] T. L. Casavant E. C. Bronson, and L. H. Jamieson, "Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system," *IEEE Trans. Parallel Distributed Syst.,* vol. 1, no. 2, pp. 195–205, Apr. 1990.

[14] M. Annaratone *et al.*, "The Warp computer: Architecture, implementation, and performance," *IEEE Trans. Comput.,* Dec. 1987.

[15] D. G. Feitelson and L. Rudolph, "Distributed hierarchical control for parallel processing," *IEEE Computer,* vol. 23, no. 5, pp. 65–77, May 1990.

[16] A. R. Hanson and E. M. Riseman, "A methodology for the development of general knowledge-based vision systems," *Vision, Brain, and Cooperative Computation,* 1986.

[17] D. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.

[18] M. Denneau J. Beetem, and D. Weingarten, "The GF11 Parallel Computer," in *Experimental Parallel Computing Architectures,* J. J. Dongarra, Ed. Amsterdam: North-Holland, 1987.

[19] M. Jeng and H. J. Seigel, "A distributed management scheme for partitionable parallel computers," *IEEE Trans. Parallel Distributed Syst.,* vol. 1, no. 1, pp. 120–126, Jan. 1990.

[20] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh, "Parallel supercomputing today and the Cedar approach," *Science,* vol. 231, pp. 967–974, 1986.

[21] G. J. Lipovski, private communication on shuttle memory, Sept. 1990.

[22] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons.* New York: Wiley, 1987.

[23] Y. W. Ma and R. Krishnamurti, "The architecture of REPLICA-A special-purpose computer system for active multi-sensory perception of 3_dimensional objects," in *Proc. Int. Conf. Parallel Processing,* 1984, pp. 30–37.

[24] A. Merigot, B. Zavidovique, and F. Devos, "SPHINX, A pyramidal approach to parallel image processing," in *Proc. IEEE Workshop Computer Architecture for Pattern Analysis and Image Database Management,* Nov. 1985, pp. 107–111.

[25] D. Rana and C. Weems, "The ICAP Parallel Processor Communication Switch," COINS Tech. Rep. 89–02, Univ. of Massachusetts, Amherst, 1989.

[26] A. Rosenfeld, private communication, Nov. 1990.

[27] D. H. Schaefner, D. H. Wilcox, and G. C. Harris, "A pyramid of MPP processing elements - experience and plans," in *Proc. Hawaii Int. Conf. Syst. Sci.,* 1985, pp. 178–184.

[28] M. Sharma, J. H. Patel, and N. Ahuja, "NETRA: An architecture for a large scale multiprocessor vision system," in *Proc. Workshop Computer Architecture for Pattern Analysis ans Image Database Management,* Nov. 1985, pp. 92–98.

[29] D. E. Shaw, "Organization and operation of a massively parallel machine," in *Parallel Computing: Theory and Comparisons,* G. J. Lipovski and M. Malek, Eds. New York: Wiley, 1987.

[30] H. J. Siegel, "Partitioning permutation networks: The underlying theory," in *Proc. Int. Conf. Parallel Processing,* 1979, pp. 175–184.

[31] H. J. Siegel *et al.* "PASM: A reconfigurable parallel system for image processing," in *Parallel Computing: Theory and Comparisons,* G. J. Lipovski and M. Malek, Eds. New York: Wiley, 1987.

[32] J. Simonson, "A programmable crossbar switch for multiprocessor systems," in preparation. Master's thesis, Univ. of Illinois, Urbana-Champaign, Feb. 1991.

[33] L. Snyder, "Organization and operation of a massively parallel machine," in *Parallel Computing: Theory and Comparisons,* G. J. Lipovski and M. Malek, Eds. New York: Wiley, 1987.

[34] M. H. Sunwoo and J. K. Aggarwal, "A vision tri-architecture (VISTA) for an integrated computer vision system," in *Proc. Image Understanding Benchmark Workshop,* 1988.

[35] S. L. Tanimoto, T. J. Ligocki, and R. ling, "A prototype pyramid machine for hierarchical cellular logic," in *Parallel Hierarchical Computer Vision,* L. Uhr, Ed., 1987.

[36] L. W. Tucker, "Architecture and application of the Connection Machine," *IEEE Computer,* pp. 26–38, Aug. 1988.

[37] C. Weems *et al.*, "A report on the results of the DARPA integrated image Understanding benchmark exercise," in *Proc. Image Understanding Workshop,* 1989, pp. 165–183.

[38] C. Weems, A. Hanson, E. Riseman, and A. Rosenfeld, "An integrated image understanding benchmark: Recognition of a 2 1/2 d mobile," in *Proc. Int. Conf. Comput. Vision and Pattern Recognition,* June 1988.

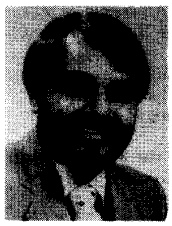
[39] C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, J. G. Nash, and D. B. Shu, "The image understanding architecture," *Int. J. Comput. Vision,* vol. 2, pp. 251–282, 1989.

[40] A. S. Youssef and B. Narahari, "The banyan-hypercube networks," *IEEE Trans. Parallel Distributed Systems,* vol. 1, no. 2, pp. 160–169, Apr. 1990.

Alok N. Choudhary received the Ph.D. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1989, the M.S. degree from the University of Massachusetts, Amherst, in 1986, and the B.E.(Hons.) degree from the Birla Institute of Technology and Science, Pilani, India, in 1982.

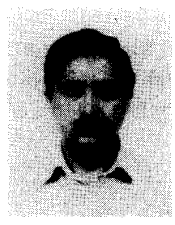
He joined the faculty of the Department of Electrical and Computer Engineering at Syracuse University in 1989. He was a visiting scientist at IBM T. J. Watson Research Center during the summers of 1987, 1988, and 1991. He was a research and teaching assistant with the Electrical and Computer Engineering Department and the Center for High Performance and Reliable Computing at the University of Illinois, Urbana-Champaign, from 1986 to 1989. He worked as a system analyst and designer from 1982 to 1984. His main research interests are in parallel computer architectures, software development environments, and applications for parallel computers, computer vision, and performance evaluation. He has co-authored a book *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems* (Boston, MA: Kluwer Academic). In addition, he has written two book chapters on parallel architectures and algorithms.

Dr. Choudhary was awarded an IEEE Engineering Foundation Award for research in parallel processing. He served as a guest editor for IEEE COMPUTER Special Issue on Parallel Processing for Computer Vision and Image Understanding, published in February 1992. He is also a guest editor of *Journal of Parallel and Distributed Computing* for a Special Issue on Parallel I/O Systems published in January 1993. He is a member of the IEEE Computer Society and the Association for Computing Machinery. He is a subject area editor of JPDC. He received the NSF Young Investigator Award in 1993.



Janak H. Patel (S'73–M'76–SM'87–F'89) was born in Bhavanagar, India. He received the B.Sc. degree in physics from Gujarat University, India, the B.Tech. degree from the Indian Institute of Technology, Madras, India, and the M.S. and Ph.D. degrees from Stanford University, Stanford, CA, all three in electrical engineering.

He is with the University of Illinois at Urbana-Champaign, where he is currently a Co-Director of the Center for Reliable and High Performance Computing and a Professor of Electrical and Computer Engineering and Computer Science, and a Research Professor with the Coordinated Science Laboratory. He has made seminal contributions in the areas of multiprocessor cache memories, pipeline processing, and multiprocessor interconnections. His cache coherence protocol, the Illinois protocol, is used in several commercial multiprocessors. His pipeline scheduling methods are now being applied in super-scalar compilers. He is currently engaged in research, teaching, and consulting in the areas of cache performance, architecture based automatic test generation, and synthesis for testability.



Narendra Ahuja (S'79–M'79–SM'85–F'92) received the B.E. degree with honors in electronics engineering from the Birla Institute of Technology and Science, Pilani, India, in 1972, the M.E. degree with distinction in electrical communication engineering from the Indian Institute of Science, Banglaore, India, in 1974, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 1979.

From 1974 to 1975 he was Scientific Officer in the Department of Electronics, Government of India, New Delhi. From 1975 to 1979 he was at the Computer Vision Laboratory, University of Maryland, College Park. Since 1979 he has been with the University of Illinois at Urbana-Champaign where is currently (since 1988) a Professor in the Department of Electrical and Computer Engineering, the Coordinated Science Laboratory, and the Beckman Institute. His interests are in computer vision, robotics, image processing, image synthesis, and parallel algorithms. He has been involved in teaching, research, consulting, and organizing conferences in these areas. His current research emphasizes integrated use of multiple image sources of scene information to construct three-dimensional descriptions of scenes, the use of integrated image analysis for realistic image synthesis, the use of the acquired three-dimensional information for navigation, and multiprocessor architectures for computer vision.

Dr. Ahuja was selected as a Beckman Associate in the University of Illinois Center for Advanced Study for 1990–1991. He received University Scholar Award (1985), Presidential Young Investigator Award (1984), National Scholarship (1967–1972), and President's Merit Award (1966). He has co-authored the books *Pattern Models* (New York: Wiley, 1983), with Bruce Schachter, and *Motion and Structure from Image Sequences* (Springer-Verlag, to be published) with Juyang Weng and Thomas Huang. He is Associate Editor of the journals IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, *Computer Vision, Graphics, and Image Processing*, *Journal of Mathematical Image and Vision*, and *Journal of Information Science and Technology*. He is a fellow of the American Association for Artificial Intelligence, and a member of the Association for Computing Machinery and the Optical Society of America.