High-Performance Computing for Vision

CHO-LI WANG, PRASHANTH B. BHAT, AND VIKTOR K. PRASANNA, FELLOW, IEEE

Invited Paper

Vision is a challenging application for high-performance computing (HPC). Many vision tasks have stringent latency and throughput requirements. Further, the vision process has a heterogeneous computational profile. Low-level vision consists of structured computations, with regular data dependencies. The subsequent, higher level operations consist of symbolic computations with irregular data dependencies. Over the years, many approaches to high-speed vision have been pursued. VLSI hardware solutions such as ASIC's and digital signal processors (DSP's) have provided good processing speeds on structured low-level vision tasks. Special purpose systems for vision have also been designed. Currently, there is growing interest in using general purpose parallel systems for vision problems. These systems offer advantages of higher performance, software programmability, generality, and architectural flexibility over the earlier approaches. The choice of low-cost commercial-off-the-shelf (COTS) components as building blocks for these systems leads to easy upgradability and increased system life.

The main focus of the paper is on effectively using the COTS-based general purpose parallel computing platforms to realize high-speed implementations of vision tasks. Due to the successful use of the COTS-based systems in a variety of high performance applications, it is attractive to consider their use for vision applications as well. However, the irregular data dependencies in vision tasks lead to large communication overheads in the HPC systems. At the University of Southern California, our research efforts have been directed toward designing scalable parallel algorithms for vision tasks on the HPC systems. In our approach, we use the message passing programming model to develop portable code. Our algorithms are specified using C and MPI. In this paper, we summarize our efforts, and illustrate our approach using several example vision tasks.

To facilitate the analysis and development of scalable algorithms, a realistic computational model of the parallel system must be used. Several such models have been proposed in the literature. We use the General-purpose Distributed Memory (GDM) model which is a simple but realistic model of state-of-the-art parallel machines. Using the GDM model, generic algorithmic techniques such as data remapping, overlapping of communication with computation, message packing, asynchronous execution, and communication scheduling are developed. Using these techniques, we have developed scalable algorithms for many vision tasks. For instance, a scalable algorithm for linear approximation has been developed using the asynchronous execution technique. Using this algorithm, linear feature extraction can be performed in 0.065 s on a 64 node SP-2 for a 512×512 image. A serial implementation takes 3.45 s for the same task. Similarly, the communication scheduling and decomposition techniques lead to a scalable algorithm for the line grouping task. We believe that such an algorithmic approach can result in the development of scalable and portable solutions for vision tasks.

Manuscript received October 15, 1995; revised March 18, 1996. This work was supported by NSF, ARPA, and AFOSR.

C.-L. Wang is with The University of Hong Kong, Hong Kong (e-mail: clwang@cs.hku.hk).

P. B. Bhat and V. K. Prasanna are with the Department of EE-Systems, University of Southern California, Los Angeles, CA 90089-2562 USA (e-mail: prabhat@halcyon.usc.edu and prasanna@ganges.usc.edu).

Publisher Item Identifier S 0018-9219(96)04992-4.

I. INTRODUCTION

Computer vision has applications in a wide variety of fields such as vehicle navigation, medical diagnosis, aerial photo interpretation, and automatic target recognition, among others. Some of these applications involve interaction with a human, and must complete within a few seconds. Other applications require a machine to act in real time, such as in automatic vehicle guidance. The maximum tolerable latency here is typically less than a fraction of a second. Most vision tasks are computationally intensive, involving complex processing. Thus for practical implementations of vision tasks to be possible, high performance computing support is essential.

Parallel processing is an attractive approach to satisfy the computational requirements of vision applications [45]. High levels of performance and fast turnaround times seem promising with general purpose parallel machines. However, simply replacing existing computing platforms with parallel machines does not guarantee that the stringent latency and throughput requirements of vision tasks will be met. To realize the full potential of parallel machines, efficient software is an important component. Parallel programs for the vision tasks must be developed. Although compiler technology is rapidly evolving, currently available parallelizing compilers can provide only limited support in automatically generating efficient parallel code from serial vision algorithms. This is because the computational characteristics of vision tasks are different from the structured number crunching computations that arise in most other grand challenge applications. A significant fraction of vision tasks consists of extensive symbolic computations, and therefore has irregular run-time data dependencies. Further, vision computations are heterogeneous, utilizing

techniques from a variety of disciplines such as signal and image processing, advanced mathematics, and artificial intelligence. It is therefore necessary to manually develop suitable parallel algorithms for vision computations.

The heterogeneity in vision tasks also arises because vision computations proceed in distinct phases [42]: low-level, intermediate-level, and high-level. The processing at each level has different computational characteristics. Low-level processing is mostly iconic, with regular and structured computations. The input to this stage is a two-dimensional (2-D) image array of pixels, representing the observed scene. Operations are performed on pixels using image data in the neighborhood of these pixels. At the higher levels, however, the operations performed on an item often have irregular data dependencies. Processing at these levels is performed on objects that are groups of features extracted at the lower level. On account of these complex and heterogeneous computational requirements, vision has been identified as a grand challenge application for high performance computing [20].

Several engineering factors also restrict the options in choosing computing platforms for vision applications. For example, many vision tasks arise in the context of robotic applications. Computing platforms for these applications must be easily embeddable on robot vehicles. Small sizes and low power requirements are important in such situations. A further consideration is that robotics applications require placing the computing resources in hazardous surroundings.

System cost is another consideration. In scientific computing environments, a single computing resource is shared by several applications. While this could lead to longer computing times, the costs incurred by each application are reduced. However, the stringent timing constraints associated with vision tasks often require that systems be dedicated entirely to a single task. Many vision application environments cannot afford these high system costs [63].

Several efforts have been directed toward providing high speed computing support for vision [4], [5], [21], [41], [53], [55]. A brief summary of leading research efforts in parallel computing for vision can be found in [63]. These efforts can be grouped into three categories, based on the nature of computing platforms they utilize: single chip (or board level) VLSI processors, specialized vision systems, and general purpose parallel machines.

1) VLSI Processors: Accelerator boards consisting of VLSI processing elements are well-suited for structured computations. These VLSI processors therefore yield excellent performance for low-level vision processing. The accelerators implement the computations in hardware, resulting in high-speed execution. For instance, VLSI chips for convolution and other low-level operations have been developed.

VLSI processors can also be programmable. Digital signal processors (DSP's) are an example of programmable VLSI processors. A DSP chip has an instruction set, and can execute any program composed of valid instructions. Due to this flexibility, efficient support for different tasks can be provided. Although VLSI processors perform well on structured computations, their architecture is not well suited for the irregular tasks in high level vision. Complete vision systems based on these components have therefore not been designed.

2) Specialized Vision Systems: Specialized vision systems are computing platforms specially designed to suit the requirements of vision tasks. The architectural features match the computational characteristics of vision processing. However, the architecture is adequately general, enabling efficient implementation of a variety of vision tasks. An example is the image understanding architecture (IUA) [64]. This architecture has three different layers of computational engines that suit the requirements of low, middle, and high-level vision tasks.

However, practical considerations make the special-purpose approach less attractive. Specialized machines tend to have a rather complex architecture, leading to considerable design and development effort. New operating systems, compilers, and other systems software must be developed for the special architecture. This design effort would result in higher system development costs. Many application scenarios cannot afford to use such expensive systems.

3) General Purpose Parallel Machines: Recent commercial parallel machines, such as the IBM SP-2, Meiko CS-2, Intel Paragon, SGI Power Challenge, and Cray T3D, have been successfully used for a variety of high performance applications. They offer hundreds of Gigaflops of computing performance. These machines are not designed for any specific application, but are meant to be general purpose systems. Most of them have a similar architecture consisting of processing elements, each with a local memory module, interconnected by a high speed network. These systems offer several advantages over the VLSI processors and the specialized vision systems:

- *Cost effectiveness:* A wide range of application domains use the same general purpose machines. The larger volume of production leads to lower system costs as compared to the specialized systems.
- *Programmability:* The general purpose machines are easily programmable. Availability of programming environments enable rapid and easy development of efficient software. VLSI processors do not provide these advantages.
- *Portability:* Standard communication libraries such as the Message Passing Interface (MPI) [39] have been defined and implemented efficiently on many parallel machines. Application software developed using these libraries can be ported from one parallel machine to another without extensive modifications.

This paper focuses on methodologies for the efficient utilization of general purpose parallel computing platforms for vision applications. In developing high-speed parallel solutions on these platforms, the following issues are important:

- Accessing the network incurs large software overheads. The interconnection network hardware typically consists of high bandwidth and low latency links. However, the time spent in executing the relevant operating system routines is an order of magnitude larger than the hardware latencies. By accounting for this overhead in the algorithm design process, faster solutions can be developed.
- Due to this expensive network access operation, most computations should ideally use data located in the local memory of the processor. Locality enhancing transformations such as data partitioning and mapping schemes are necessary to achieve good performance.

These issues can be handled either through compiler transformations or through careful algorithm design.

a) Compiler transformations: In this approach, an optimizing compiler transforms a sequential user code into an efficient parallel code for a target architecture. This simplifies the task of the end-user, since no special efforts are necessary to generate efficient parallel code. On the other hand, the user does not have complete control over process scheduling, data layout, and data movement operations. Performance conscious users can optimize their code to some extent by providing compiler directives. Data parallel compilers allow users to specify initial data layouts. However, they yield good performance only when large amounts of data are processed in a structured manner. Currently, the compiler-based approach achieves good performance in parallelizing loops, especially in cases where regular data dependencies exist between the computations [2]. Recently, compilers have been developed that can automatically exploit optimal mixes of data and task parallelism, based on the particular application program and target platform combination. However, this effort has focused on applications that have a static and predictable computation structure [56]. Most high-level vision operations contain extensive symbolic operations, and therefore lack a structured and regular data access pattern. Automatic (compiler based) parallelization of these computations would therefore result in load imbalances and poor processor utilization.

b) Design of efficient algorithms: Contrary to the compiler-based approach, the user is in complete control of the parallelization process. The user designs efficient parallel algorithms, and then develops explicit parallel code for them. During the algorithm design phase, the parallelization overheads are carefully analyzed and it is ensured that these overheads are not significant. Parallel programs are developed using a high level programming paradigm. The message passing paradigm has been widely used on account of its excellent support for process control and data movement between processes. Message passing standards such as MPI [39] have been defined, making it convenient to develop portable code. We believe that this is a promising approach to the design of high speed parallel solutions to vision applications. In the following sections, the issues in pursuing this algorithmic approach are discussed in detail.

A primary goal of analyzing the parallelization overheads during the algorithm design phase is to design scalable solutions. Scalability ensures that the overheads due to parallelization do not dominate the performance gains. A scalable algorithm can thus realize increasing speed ups as larger parallel system configurations are utilized. Many algorithmic techniques have been developed to eliminate or reduce the parallelization overheads. Examples of these techniques include data partitioning and mapping, load balancing, and schemes to hide communication latencies.

Thus the algorithmic approach consists of the following steps:

1) Analysis of the computation and communication characteristics of the algorithm. This analysis can help identify the bottlenecks that would restrict scalability of the parallel solution.
2) Application of the appropriate algorithmic technique to overcome these bottlenecks, resulting in a scalable algorithm.

To support the analysis phase and to evaluate the algorithmic techniques, an analytical representation of the parallel system is necessary. A computational model is such a representation. It consists of parameters to represent the computing power of the nodes, hardware communication delays incurred in the network links, and software overheads in interfacing to the communication modules, among others. Several popular models for general purpose parallel computer systems are in use [14], [18], [27]. In choosing among these, a trade-off must be made between accuracy and simplicity. An accurate model can provide realistic and reliable estimates of overheads and execution times. However, the analysis becomes increasingly complicated, restricting applicability to only simple systems.

Many different techniques can be developed with the assistance of such an analytical model. A common goal of these techniques is to reduce overheads in parallelization of the applications. Data mapping techniques help to reduce the frequency of access to remotely located data. On the other hand, techniques such as overlapping communication with computation hide these overheads. Load balancing techniques lead to efficient utilization of all the computing nodes.

Our approach to high performance computing for vision is based on the algorithmic approach. The models and techniques developed by us result in efficient and scalable parallel solutions to many vision tasks. These results are discussed in Section IV.

The rest of the paper is organized as follows—Section II summarizes the computational nature of vision tasks. In Section III, we describe the architectural features of current general purpose parallel machines, and also survey the popular computational models. Section IV details our algorithmic approach of utilizing these parallel machines for vision computations and summarizes some performance results. Section V concludes the paper, where future research directions are identified.

II. The Nature of Vision Computations

To develop efficient parallel algorithms for vision, it is important to understand the computational characteristics of vision tasks. In this section, we summarize the salient features of vision computations.

A. Computational Characteristics

Vision tasks have a heterogeneous computational profile. The early phases consist of structured computations with localized data access patterns. This regularity is absent in the later phases, where extensive symbolic computations are performed. From a computational perspective, the vision process is generally divided into three distinct phases [42].

In the *low-level processing* phase, the raw input image is processed to extract primitive features, such as edges. The input image is represented as a 2-D array of pixels. Most of the pixel operations in this phase are performed using image data in the neighborhood of the pixels. The communication pattern is local and processing across the image is uniform. Such algorithms are easily parallelized.

In the *intermediate-level processing* phase, low-level features are grouped into a hierarchy of increasingly complex and more abstract descriptions. For example, edges are grouped into curves, curves into surfaces and surfaces into objects. These elements are combined into pairs, triples and larger structures, so the number of alternatives can grow to be very large. Of the many combinatorial groups possible, only some are meaningful. Efficient techniques are needed for forming and evaluating the groups.

In the *high-level processing* phase, the descriptions generated at the intermediate level are interpreted for finding semantic attributes of an image. This includes the task of object recognition and generation of inferences from collections of objects. Processing at this level tends to be highly irregular though the number of data objects is smaller. Typical algorithms consist of graph matching and rule-based analysis.

Thus it can be seen that the computations constitute a mix of varied methodologies. The regular data flow present in low-level processing is absent at the higher levels. A wide variety of data structures and representations are also employed. In low-level processing, imagelike arrays are used. Typically, input data sizes are in the range of 1 K×1 K pixels of data with each pixel represented by 16 b. As the computation proceeds, structures such as linked lists are used. Data is often organized in the form of complex data structures, and data accesses make extensive use of pointers.

The computations consist of both integer-based numerical operations and symbolic manipulations. Many of the high-level symbolic computations do not have localized data access patterns. This results in large communication overheads, when the computations are parallelized. In executing these high-level tasks, the flow of control and the load distribution on the processors are often unpredictable at compile time. Parallel algorithms, data partitioning and mapping methods, and load balancing strategies are needed to result in useful speed-ups. Load imbalances can arise since irregular triangular meshes are sometimes used to approximate surfaces of arbitrary topology. Often the techniques employed are not fixed, but evolve over a period of time. Thus reprogrammability is also required.

Most of the past work in the area of parallel processing for computer vision has addressed the use of parallelism for problems in low-level and some geometric problems in the interface between image processing and image understanding [43]. Design of parallel techniques for individual problems in low-level processing seems to be reasonably well understood [47]. However, this is not true of the higher level tasks. Implementations of these tasks on state-of-the-art parallel machines do not offer the performance levels demanded by interactive or real-time modes of execution.

Thus, vision tasks are computationally complex mainly because of their irregular dependencies, frequent interprocessor communication, and heterogeneous operations. These characteristics are different from the number crunching tasks in scientific applications, where the complexity arises primarily due to the large number of operations to be performed on extensive amounts of data. Most real vision applications use images whose sizes are atmost a few thousand pixels on each side. Larger parallel systems do not automatically yield higher speed ups for vision applications, as they do for many of the scientific applications. High processor efficiency for the irregular vision tasks is therefore difficult to achieve.

B. Example Vision Tasks

We illustrate the computational aspects of vision processing through some representative tasks. The examples chosen include tasks from low, middle, and high-level processing.

1) Segmentation: Segmentation identifies the line segments in the image, taking the 2-D pixel array as input. This is one of the first steps in most vision applications. During later stages, these segments can be further grouped into more complex shapes.

Segmentation techniques fall into two classes: edge-based and region-based. In edge-based techniques, "edgels" which represent local intensity discontinuities are first detected. They are then "linked" to form curves which may be further approximated by straight lines or other primitives such as B-splines [43]. In region-based techniques, regions of homogeneous intensity (or color) are found and then their boundaries are detected.

It should be noted that the segmentation step, while relatively easy, can consume significant computational resources. The amount of data to be handled here is rather large compared to the subsequent stages. It is not untypical for about 25% of the total application processing time to be devoted to this step. For both approaches above, the initial processing is iconic (image like) and consists of local neighborhood operations. This step is easily parallelized on virtually any architecture. However, the later steps, such as that of line finding, require the conversion of iconic

representations into symbolic lists. These operations are not so straightforward to parallelize.

2) Perceptual Grouping: In perceptual grouping, lower level features, such as lines and curves, are grouped into successively more complex and abstract features such as surfaces and parts of objects. These methods typically rely on geometric and other symbolic relations between the lower level features. Perceptual grouping techniques typically follow a "hypothesize, select, and verify" paradigm. Initially, grouping hypotheses are formed based on some geometric relations such as parallelism or simple symmetries. A large number of hypotheses may be generated at this step. Then stronger, but computationally more expensive, measures are used to select among the competing hypotheses. Finally, the selected hypotheses are "verified" based on all available image evidence [34].

Parallel implementation of such grouping operations is complicated as the computations are mostly symbolic, and not homogeneous for all features. The communication pattern is data dependent and irregular. At the hypothesis formation stage, a large number of hypotheses can be generated due to the combinatorial nature of the process.

3) Object Recognition: Object recognition consists of matching observed descriptions with stored models in a database. Often, moving objects must be visually tracked. Object recognition requires matching of attributed graph structures. Commonly used techniques are those of subgraph isomorphism and of maximal cliques. However, purely graph theoretical techniques are not sufficient to capture the variability that is inherent in observed scene descriptions. Apart from these techniques, some heuristic criteria need to be incorporated. Parallel implementations of such techniques have been reported [29].

Thus the above examples illustrate the differences in computational characteristics of vision tasks at low, middle, and high levels.

III. HPC Systems: Architecture and Models

This section describes the architectural characteristics, programming aspects, and performance issues of current generation high performance computing platforms. With this knowledge, the reader will be able to easily appreciate the role of the algorithmic techniques to be presented in Section IV.

A. The Architecture of HPC Systems

Traditionally, vector supercomputers have been the platforms of choice for high performance computing applications. Advances in semiconductor technology are rapidly changing this scenario. The performance of microprocessors as well as the capacity of memory chips have improved tremendously in recent years. These components are used in the large uniprocessor workstation market, resulting in a corresponding decrease in their cost. The ready availability of these low cost building blocks has led to their use in parallel computing systems as well. Several commercial general purpose parallel machines such as the IBM SP-2, Cray T3D, TMC CM-5, and Intel Paragon have been designed around these commercial-off-the-shelf (COTS) components. All these parallel systems have a similar architecture consisting of three main components: 1) Computing nodes based on state-of-the-art processor chips and DRAM modules, 2) a low-latency high-bandwidth network that interconnects the computing nodes (the network typically consists of high speed point-to-point links and routing switches), and 3) a network interface that couples each computing node to the network [14].

The COTS-based parallel systems provide several advantages over the monolithic vector supercomputer systems:

- The architectural configuration is flexible. The system can be easily expanded to include additional processor and memory modules into the system.
- The system can be easily upgraded to incorporate technological advances. As faster processors and memory components become available, existing modules can be replaced with the new components. The existence of standard hardware interfaces makes this especially easy.
- The available memory bandwidth scales with the system size. This is because the system memory is partitioned into several modules, each with its set of memory ports. The number of memory modules increases with the system size and so does the number of memory ports.

The COTS-based systems also offer advantages over single instruction multiple data (SIMD) stream architectures. SIMD systems consist of a central unit that fetches and interprets instructions and then broadcasts appropriate control signals to a number of processors operating in lock step. The architecture also consists of a high-speed network for interprocessor communication. The Connection Machine CM-200, the MasPar MP-2, and the forthcoming Cray 3/SSS Super Scalable System are examples of SIMD systems. Many applications with an embarrassingly parallel computational profile perform well on these systems. However, not all high performance applications are of an embarrassingly parallel nature. The generality, low cost, and increasing performance levels of 32-b and 64-b processors make the COTS-based systems very attractive. High-performance systems are therefore converging toward a multiple instruction multiple data (MIMD) stream architecture designed around the COTS components.

B. An Architectural Classification

The COTS-based MIMD systems are powerful platforms for many HPC applications. Several classification schemes can be used for the MIMD systems. We base our scheme on factors that affect the achievable performance. We classify the MIMD HPC systems based on three architectural features:

- *Memory partitioning*: Based on the organization of the physical memory modules, MIMD HPC systems can be classified into two classes. Systems in the *distributed memory* category are organized with a

memory module situated physically close to each processor module. The *centralized memory* systems are organized with all the memory modules situated at a single location.

The distributed memory systems offer potentially better memory access performance, especially in large systems. If many of the memory accesses by a processor can be satisfied at the local memory module, the interconnection network is accessed only occasionally. On the other hand, the centralized memory systems do not scale well to large system sizes. Since all memory accesses contend for the same interconnection network, it can lead to degraded memory system performance.

In distributed memory systems, the partitioned nature of the memory modules implies that some portions of the system memory can be accessed directly by a processor, while accesses to other portions of memory require use of the interconnection network. Many distributed memory architectures provide additional hardware support for remote data access, in the form of intelligent network interfaces. The interfaces can transfer blocks of data between memory modules without processor intervention.

- *Address space organization*: MIMD HPC systems can be further classified based on the memory management scheme. In *shared address space* systems, the memory is logically organized as a single address space. A processor can access any location of the memory using read and write operations. In *private address space* systems, each processor can directly access only its local memory module, which it controls. Access to a remote memory location requires *explicit* interaction and communication with the appropriate remote processor.

  When distributed memory systems are logically organized as shared address space systems, remote memory modules appear to a processor as portions of its address space. This organization results in a nonuniform memory access (NUMA) characteristic, with some portions of the address space having lower access times than other portions. In the centralized memory systems, hardware always supports a shared address space organization.
- *Interprocessor coupling*: Based on the physical proximity between the individual processing nodes, we distinguish between loosely and tightly coupled architectures. *Loosely coupled* architectures typically consist of independent uniprocessor systems, connected together with additional network links. *Tightly coupled* systems are typically organized as a single "box," which contains all the processor and memory modules interconnected by a special purpose interconnection network. A workstation cluster is an example of a loosely coupled architecture, while a massively parallel processor like the Cray T3D is an example of a tightly coupled architecture.

  The degree of interprocessor coupling must often be considered during the algorithm design phase. Interactions between processors are "expensive" in a loosely coupled system. These systems are therefore better suited for coarse grain parallel solutions. Algorithmic techniques to hide overheads in interprocessor interaction are particularly important in these systems.

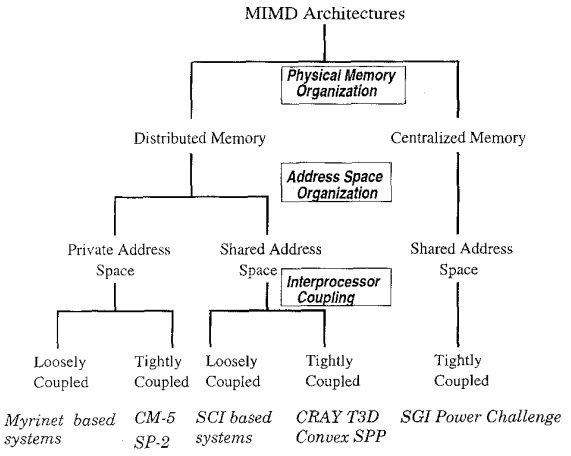


Fig. 1. Classification of current HPC systems.

Fig. 1 shows our classification scheme. We identify examples of each class at the bottom of the figure. Some of these examples are discussed in further detail next.

Workstation clusters interconnected by high-bandwidth low-latency networks are proving to be a low-cost approach to high performance systems. A typical example, *Myrinet*, consists of three main components: computing elements, namely the workstations, the network, consisting of links and switches to route data, and network interfaces between the workstations and network links. The network interface consists of a special processor that can transfer blocks of data, once transfer parameters are specified. This feature allows for the overlap of computation and communication in this loosely coupled system. One-way message latencies of 100 μs and bandwidths of 255 Mb/s have been observed in a Myrinet-based system interconnecting Sparc workstations [9].

Examples of distributed-memory, private-address-space, tightly-coupled systems are the *Thinking Machines CM-5, IBM SP-2, Intel Paragon*, and *Meiko CS-2*, among others. A CM-5 system contains between 32 and 16 384 processing nodes (PN's). Each node is a 32 MHz Sparc processor with upto 32 Mbytes of local memory. The PN's are interconnected by three networks: a data network, a control network, and a diagnostic network. The data network provides point-to-point data communication between any two PN's. Communication can be performed concurrently between pairs of PN's and in both directions. The data network is a 4-ary fat tree, whose bandwidth continues to scale linearly up to 16 384 PN's. The control network provides cooperative operations, including broadcast, synchronization, and scans (parallel prefix and suffix). The control network is a complete binary tree with all the PN's as leaves. Each partition consists of a control processor, a

collection of processing nodes, and dedicated portions of the data and control networks [58].

The Scalable Coherent Interface (SCI) standard specifies the implementation and operation of a shared address space, distributed memory system [51]. A SCI system consists of interconnected processor and memory nodes. The basic interconnection topology is that of a ring. However, other topologies can be realized using switches. SCI supports cache coherence in hardware, with a distributed directory-based coherence protocol. Unidirectional point-to-point links with differential signaling, as well as distributed cache sharing lists enable scalable performance. SCI can be used in the design of loosely coupled or tightly coupled systems. Bus interfaces and switches based on SCI allow for several workstations to be connected together as a distributed memory, shared address space, loosely coupled system [16].

The *Cray T3D* [13] and the *Convex SPP* [12] are examples of distributed memory, shared address space, and tightly coupled systems. Each processing element (PE) of the T3D consists of a processor, local memory, and additional support circuitry. Two PE's form a node and share a network interface. The nodes are organized in a three-dimensional (3-D) torus topology. Hardware access to remote memory is provided by the support circuitry and the block transfer engine (BTE), which is situated on the network interface module. The BTE's can transfer large chunks of data between memory modules. The destination processor can then access the data from its local memory. The support circuitry allows the processor to directly access remote memory, without making additional copies in the local memory module. The T3D also has other unique features, such as a cache which can be invalidated by a user command.

The centralized memory category has representatives only in the tightly coupled shared address space subclass. The *Silicon Graphics PowerChallenge* system is a typical example. It consists of up to 18 processing nodes (MIPS R8000) interconnected with a single system bus. All accesses to memory take place over the system bus [54]. Centralized memory systems do not scale to large system sizes. Our efforts toward high performance vision systems have therefore focused on distributed memory systems.

C. Parallel Programming Models

Several models of parallel computer systems have been proposed in order to design, use, and effectively reason about these systems. These models can be broadly classified as machine, architectural, computational, and programming models [22]. Machine models are specific to a particular machine and characterize execution of low level (assembly language) code. Architectural models characterize the features of a class of machines with a similar architecture. System organization details such as memory partitioning and interconnection network topology are addressed. While both these models are useful for efficient algorithm design on a particular target machine, they are not suitable for portable algorithm design. We therefore do not consider them in further detail.

Computational models provide an abstract view of a class of computers, while accurately reflecting costs of operations on these machines. We discuss computational models in Section III-D.

Programming models are a high level representation of a computer system's operation. They describe the system in terms of the semantics of a particular programming language. While a computational model describes memory as a sequence of locations, a programming model will describe it in terms of data structures [22]. A parallel programming model essentially contains abstractions for representing parallel computational activities, as well as for communication and interaction between the parallel activities. Hardware and system software layers on a parallel platform together implement these abstractions.

The existence of standard programming models permits architecture independent programming. Software developed in accordance with a programming model is portable across all platforms that support this model. The commonly used parallel programming models are the data parallel, shared memory, and message passing models.

1) The Data Parallel Model: A data parallel program looks similar to a sequential program, with additional compiler directives. Parallel threads execute the same program, although they operate on different portions of the problem data space. The directives are guidelines for partitioning the data elements, and for distributing them on to the processing elements. Primitives such as *shift, transfer, etc.* are used for interprocess communication. The data parallel programming model is suitable for structured parallel applications, where a large number of data elements are processed identically. Although the data parallel programming model matches the SIMD architectures closely, it can be implemented on the MIMD architectures as well. Unlike in the SIMD systems, the computations execute with a coarse granularity. They synchronize at intermediate points, rather than at every step. This is referred to as the single program multiple data (SPMD) mode of execution. A number of data parallel languages are in use, such as HPF [23] and Fortran D [10].

2) The Shared Memory Model: In the shared memory programming model, processes communicate through a common logical address space. A variable written to by one process can be read by any of the other processes, with the same variable name. This paradigm is thus very similar to the sequential programming paradigm. It has therefore been considered to be easy to understand.

The multiple processes can execute asynchronously, each performing different functions. This model is therefore applicable to a wider and more general class of problems than the data parallel model. New processes can be dynamically forked and terminated at any point in a program. Loop iterations can also be executed in parallel across a set of processes. In a self-scheduling loop, each

process is assigned a chunk of loop iterations. The first process which completes its work requests a new chunk. This continues until all the iterations are complete.

Often, portions of the shared data space must be accessible to only one process at a time. This is the mutual-exclusion problem. Several schemes have been developed to ensure this functionality. Examples are semaphores [15] and monitors [24]. These schemes require a process to obtain exclusive access rights before it executes the critical portion of code.

3) The Message Passing Model: In a message passing program, parallel processes execute with private copies of code and data. Each process can perform a different function, or can perform identical computations on different data. When intermediate results computed by the individual processes must be combined, explicit interaction between the processes must take place. One process "sends" a message containing the necessary information, and the other process "receives" it. Collective communication operations involving multiple processes are also feasible. Apart from such a message-based interaction, the contents of a process's data space are not accessible to other processes. For efficient execution, the programmer carefully partitions the data space between the processes so as to reduce the number of exchanged messages. The message passing model thus provides versatility and a higher degree of control in tuning parallel code for enhanced performance.

Programming using the message passing model has been considered to be tedious and error-prone as compared to the shared memory style. Several message passing libraries such as the Parallel Virtual Machine (PVM), the Message Passing Library (MPL), and CMMD have been developed [8], [26], [57]. These libraries consist of functions for thread creation and termination, sending and receiving messages between threads, and for collective communications such as broadcast, gather, and scatter. To enable the development of portable message passing programs, the MPI standard [39] has been recently defined.

4) Hybrid Programming Models: We have primarily used the message passing model in programming our vision applications. The versatility of this model and the provision for task parallelism are especially important for the irregular problems arising in intermediate and high level vision, which have only limited amounts of structured data parallelism.

The shared memory programming style is considered to be simple and elegant. However, this programming model often does not offer adequate flexibility when performance sensitive optimizations are to be made. A hybrid of the shared memory and message passing styles often offers benefits of both performance and programming ease. Consider the simple example of vector reduction by summation using shared memory programming. If a single shared variable is used to store all the intermediate results, updating the shared variable must be done using a lock. This would result in excessive idle waiting at the lock. Instead, a private variable could be used in each of the parallel processes to store partial results, as is typical in the message passing model. All the partial results can be then combined only at the final stage.

The message passing model can also be simplified by incorporating ideas from the shared memory style. Data communication in the message passing model requires the programmer to identify both the sending and receiving processes, and to insert appropriate communication calls in both of them. From the programmer's view, communication requires synchronization between the communicating processes. "Active Messages" [17] is a deviation from this style and provides asynchronous message passing. For instance, to send a message from one node to another, the sending node initiates a "put" operation. This creates a message consisting of the data, its destination node and address, and the address of a user level handler. The message is then sent to the destination node, where it interrupts the processor and starts the handler routine. The handler simply reads the data from the message and stores it into the destination address. At a later point, the process on the destination node reads this data which is now simply another variable in its private address space. This scheme thus decouples the sending and receiving processes, and obviates the need for synchronization between them. It is necessary for the sending process to know the address layout of the destination process. This requirement is met by programs executing in the single program multiple data (SPMD) mode, where all the processes have a uniform code image. The Cray T3D also supports a similar asynchronous style of communication in its SHMEM library [52].

D. Computational Models

A computational model is an analytical representation of a computer system. For a parallel system, the computational model consists of parameters to represent the computing power of the nodes, hardware communication delays incurred in the network links, and software overheads in interfacing to the communication modules, among others. Using the model, the designer can predict the performance of an algorithm-architecture pair, and also identify bottlenecks in achieving scalable parallel performance.

An effective model must balance conflicting requirements of *simplicity*, *accuracy*, and *broad applicability*. Simplicity allows for easier analysis, but the model should also accurately represent the salient features of the system architecture and enable realistic predictions to be made. Details specific to a small set of machines should be omitted, so that the model can be applied to a wider class of architectures.

A number of parallel computational models have been proposed and studied in the last 20 years. Some of these models are discussed next.

1) The Parallel Random Access Machine (PRAM): The PRAM model [18] is one of the earliest and most widely used computational models. It represents an ideal parallel machine with P processors and a single shared memory of unbounded size. Any processor can access any memory location with no access latency. All the processors operate with a common clock and synchronize at every step.

The PRAM model has been criticized as being unrealistic, since it does not address memory latency and memory contention issues. Some variations of the PRAM consider overheads due to contention at a single memory location. For instance, there are variations of the PRAM model based on whether simultaneous reads or writes to a single location by multiple processors is permitted. However, contention can also occur at memory ports. In distributed memory machines, the number of memory locations in a module is much larger than the number of memory ports to the module. The PRAM model does not account for such contention at a memory port. Accordingly, the PRAM model is not ideally suited for efficient algorithm design on distributed memory architectures. For instance, efficient data layout and data remapping techniques cannot be designed using this model.

2) Network Model: The network model represents the overheads due to specific topological organizations of the processors, such as rings, trees, meshes, and hypercubes [33]. Communication is allowed only between directly connected processors; other communication is explicitly forwarded through intermediate nodes. In current HPC systems, however, the hardware network latency is negligible in comparison with the software overheads. The network model does not capture this trait accurately.

3) Bulk Synchronous Parallel (BSP) Model: The BSP model [59] represents a distributed memory machine in terms of processor/memory modules, an interconnection network, and a *synchronizer* which performs barrier synchronization. Computations consist of a sequence of *supersteps*. In each superstep, every processor is allocated a task consisting of some combination of local computation steps, and send and receive operations. A message sent at the beginning of a superstep can only be used in the next superstep, even if the length of the superstep is longer than the message latency. After each period of L time units, a barrier-style synchronization is performed to determine whether the superstep has been completed by all the processors. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep. The value L must be carefully chosen based on the network capability and the amount of computation in the algorithm.

The model provides a restricted framework for algorithm designers to hide communication latency by overlapping computation and communication. However, it assumes special synchronization hardware, which may not be available on many parallel machines [14].

4) LogP Model: The $LogP$ model is a general purpose model for distributed memory machines [14]. The model has four parameters: L (latency), o (overhead), g (gap), and P (number of processors). The processors work asynchronously and communicate by point-to-point messages over a network. A message from a processor takes $L + 2o$ time to reach the destination processor, where L represents the hardware latency and $2o$ represents software overheads at the network interfaces of the sending and receiving processors. The network is assumed to have a *finite capacity*, such that at most $\lceil \frac{L}{g} \rceil$ messages can be in transit from any processor or to any processor at any time.

The $LogP$ model does not assume special hardware support to synchronize processors. All collective operations are done by point-to-point message passing. The basic $LogP$ model assumes all messages are of *small size*, containing a word (or small number of words). This assumption is true in the case of only a few parallel machines (e.g., CM5) that allow the use of low level communication primitives in user code. Moreover, software overheads such as end-to-end flow control, packet sequencing to preserve in-order delivery, and buffer management are not considered.

5) Postal Model: The postal model assumes a fully connected distributed memory system architecture [7]. The model is similar to the $LogP$ model in the sense that a communication latency λ is associated with each communication step. If at time t processor p sends a message M to processor q, then processor p is busy sending message M during the time interval $[t, t+1]$, and processor q is busy receiving message M during the time interval $[t+\lambda-1, t+\lambda]$. The communication latency λ is used to capture both the software and hardware overhead costs. In the postal model, the term *message* refers to an atomic piece of data (e.g., a packet) communicated between processors. A message cannot be broken into smaller pieces at the sending processor, at the communication network, or at the receiving processor.

6) The Block Distributed Memory (BDM) Model: The BDM model [27] characterizes a distributed memory architecture with a hybrid programming model. The programming model assumes a NUMA shared address space. Access to a remote memory location requires the programmer to copy the block of memory containing the desired data to a portion of the local memory module. The actual data movement is the same as in the message passing programming model. However, the programmer need not issue a send operation in the sending process. This transfer appears to the programmer as a block copy from one region of the shared address space to another.

The cost incurred in the data transfer is modeled using parameters representing memory latency, communication bandwidth, and spatial locality. The cost of accessing a remote location is $\tau + m\sigma$, where τ is the maximum latency for a requesting processor to receive a packet consisting of m consecutive words, and σ is the rate (time per word) at which a processor can inject a word into the network. The same cost $(\tau + m\sigma)$ is incurred even if a single word is moved. Thus the model encourages the use of spatial locality. The capability of interconnection networks to hide memory latency is modeled by pipelined prefetching. k prefetch read operations issued by a processor can be completed in $\tau + km\sigma$ time.

7) The GDM Model: The GDM model captures the features of the general purpose parallel systems organized as distributed memory machines with a message passing programming model. This model has been used in [47]–[49], [60], [62]. The high-bandwidth, low-latency interconnection network in these machines can be modeled

as a complete graph since the network hardware latencies are relatively small in comparison with the large software overheads in message passing [28]. The model consists of two parameters. The *startup time*, T_d, includes the software and communication protocol overheads. This is associated with each communication step. The *transmission rate*, τ_d, represents the time taken to transmit a unit of data.

Point-to-point communication of a message of size m between a pair of processors takes $T_d + m\tau_d$ time. Further, a permutation of data elements among the processors, assuming that each processor has m units of data for another processor, also takes $T_d + m\tau_d$ time. Many HPC systems provide support for overlapped communication and computation with an additional communication processor. The main processor only spends T_d time to activate the communication processor. Further data transmission is handled by the communication processor, during which time the main processor can continue with its computations. To perform a *global operation* such as a broadcast, a barrier synchronization, or a reduction operation (sum, min, max, prefix sum, etc.), τ_g time is required. The model assumes τ_g is a constant, if the machine has a control network dedicated to performing fast global operations, i.e., CM-5 [32]. Otherwise, the operation is implemented using point-to-point communication primitives, $\tau_g = O((\log P)T_d)$, where P is the number of processors.

We have used the GDM model as an analytical basis in the design of high performance parallel algorithms for vision. The model accurately reflects the performance of current generation general purpose parallel HPC systems. It has proved to be an effective model to predict achievable performance, to guide algorithm design, and to identify performance bottlenecks for our applications. In the next section, we present some of our research in this direction and our experimental results on state-of-the-art systems.

IV. An Algorithmic Approach to High Speed Vision

As discussed in Section III-A, current generation COTS based general purpose HPC platforms offer several advantages. Desired performance levels can be achieved by suitably scaling up system size. The architectural similarity between the systems in this class permits software solutions to be easily ported from one system to another. To facilitate easy software development, several programming environments have been developed for these systems. Currently, these platforms are widely used for scientific computing applications.

In developing parallel solutions for vision problems on these systems, large speed-ups and efficient processor utilization are often hard to realize. This is primarily due to the irregular communication patterns in vision tasks and the large communication costs of coarse-grained distributed memory systems. With the assistance of a computational model and careful analysis of the overheads, efficient parallel algorithms for these tasks have been designed. We discuss this algorithmic approach in this section.

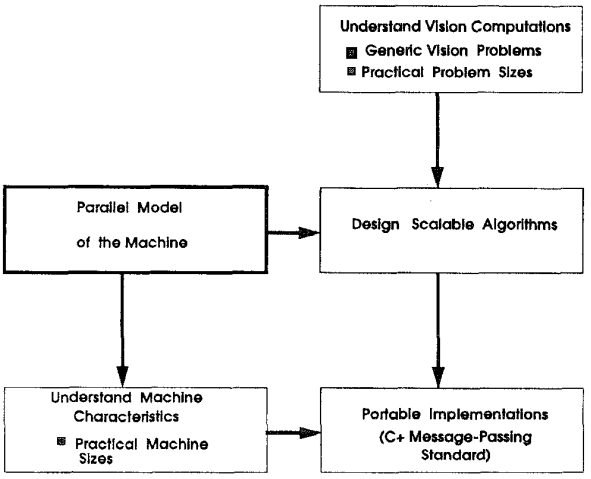


Fig. 2. Overview of the algorithmic approach.

A. Our Approach

The focus of our approach is on the development of general algorithmic techniques for generic vision problems, suitable for the large class of general purpose distributed memory machines. Fig. 2 shows the key steps in this approach.

- The architecture of the distributed memory machines is first studied and the salient computation and communication features of these machines are captured in terms of the GDM computational model described in Section II-D. The model constitutes a framework to design parallel algorithms and accurately predict their performance. For most distributed memory parallel machines, the ratio of T_d to τ_d is in the range of several hundreds to few thousands as shown in Table 1 [3].
- A number of *generic* problems arising in low, intermediate, and high level computer vision are then considered. The computational model is used to design scalable algorithms for these generic problems and to analyze their performance. With a known problem size, we can estimate the overheads and evaluate the performance of the parallel algorithms without actual implementation of the algorithms.
- We implement our algorithms on distributed memory machines. The algorithms are programmed using the message passing programming model (C and explicit message passing commands such as CMMD in CM-5 and MPL in SP-2). This programming model offers the desired amount of flexibility. Recently, we have been using the MPI message passing standard to ensure portability of code.

We have developed general techniques along these lines, and performed scalable implementations of several vision tasks [1], [11], [35]–[37], [40], [46]–[49], [60]–[62].

B. Scalable Portable Solutions

For the developed algorithms to be useful, it is important that the solutions be *scalable* and *portable*. Informally,

Table 1 Communication Parameters of Various Distributed Memory Machines

Time in Microseconds			
Machine	T_d	τ_d	1/clock rate
Intel iPSC/860	136-150	1.60-2.86	.0250
Intel Delta	85-250	.47-1.14	.0200
Intel Paragon	140-180	.40	.0125
nCube-2	60	1.60	.0500
TMC CM-5	88-260	.80-1.00	.0330
IBM SP-1	240-600	1.14	.0160
IBM SP-2	39-46	.14	.0148
Meiko CS-2	10-95	.24–.27	.0250
Cray T3D	1.0-2.0	.05	.0067

scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processors. Thus a parallel algorithm-architecture pair is considered scalable if the execution time of the algorithm on a machine with P processors varies as $\frac{1}{P}$ [30]. Scalability ensures that parallelization overheads do not dominate the performance gains due to parallel processing, as larger systems are used.

Different definitions of scalability have appeared in the literature. In [30], isoefficiency is used as a metric of scalability. Based on this definition, a scalable parallel system is one in which the efficiency can be kept fixed as the number of processors is increased, provided that the problem size is also increased. The isoefficiency function expresses quantitatively, in terms of the number of processors, the necessary increase in problem size to maintain the efficiency constant. A system with a smaller isoefficiency function is therefore more scalable than a system with a larger isoefficiency function.

A variety of parallel processing systems are currently in use. Porting software across different platforms is difficult due to the use of proprietary parallel programming libraries. The effort involved in developing efficient parallel programs suggests the need for software portability between the systems. The MPI standard [39] has been defined for this purpose. MPI specifies the semantics and function-call interfaces of a set of standard library routines. These routines can be called from Fortran or C programs. Programs developed using MPI function calls can be easily ported across platforms that implement the standard functionality. Efficient implementations of MPI are possible by optimizations at the hardware and systems software level.

C. Parallel Algorithmic Techniques

To develop high performance vision solutions, we design efficient solutions for some generic vision problems. These problems arise in many vision applications, and therefore, efficient solutions to the generic problems lead to efficient execution of a large class of vision applications as well. We design scalable parallel algorithms for these problems and develop portable code for these algorithms using MPI.

We use several parallel algorithmic techniques to design scalable algorithms. These techniques improve load balance between the processors, reduce interprocessor communication overheads, and reorganize data layouts among the processors.

Imbalanced workloads can often occur in vision tasks when input images with a nonuniform feature distribution are divided among the processors in a regular way. In such cases, the achievable speed-up is determined by the most heavily loaded processor. *Load balancing* techniques reduce imbalances in the amount of computation to be performed by each processing node [6], [19]. Data structures such as line segments and their associated computations are redistributed among the processors to achieve load balance [11].

Message vectorization, communication scheduling, overlapping of computation and communication, and decomposition techniques are some of the algorithmic techniques that reduce overheads in interprocessor communication.

Message vectorization or message packing reduces message startup overheads [60]. Several short messages with the same source and destination are combined and sent as a single large message block. Although this does not reduce the total amount of data transferred, the communication overhead is significantly reduced because the number of startups is fewer. In current generation parallel machines, the startup overhead is much larger than the unit transmission cost.

Decomposition techniques constitute a different approach toward reducing the number of communication steps in an algorithm, and therefore the total startup costs [31]. For example, to broadcast a message, the processor array of size P can be logically partitioned into a $\sqrt{P} \times \sqrt{P}$ mesh. Data is first broadcast along a single column and then broadcast concurrently along each row. Instead of sending messages individually from a processor to all the other $(P-1)$ processors in $P-1$ communication steps, $2 \times (\sqrt{P}-1)$ communication steps are sufficient using this technique.

Node contention can often lead to poor performance. When many processors attempt to communicate with a single processor simultaneously, the limited communication hardware at this processor results in a serialization of these communication events. With communication scheduling techniques, the communication events can be distributed over time, so that traffic among the processors is smooth. Thus node contention overheads can be reduced [35], [50]. Efficient schedules for frequently occurring patterns (e.g., many-to-many personalized communication) are useful in a variety of situations.

Many general purpose machines provide hardware support for message passing in the form of advanced communication adapters or intelligent network interfaces. These adapters can perform communication related operations such as transmission, reception, and protocol processing. The main processor places the message to be sent in its memory and issues a command to the network interface. While the network interface performs the actual send operation, the main processor can continue with its computations. This hardware feature can be exploited at the software level using nonblocking send commands. Such a send operation incurs only the start-up cost T_d, and the data transmission cost is hidden. With a little reorganization of the control flow, algorithms can benefit from this feature [14].

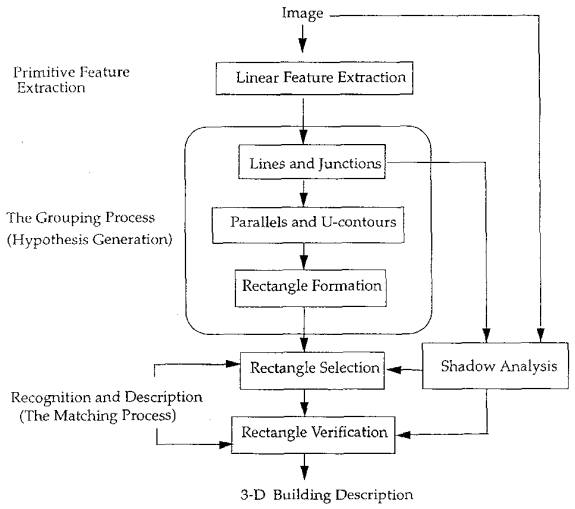


Fig. 3. An overview of the building detection system.

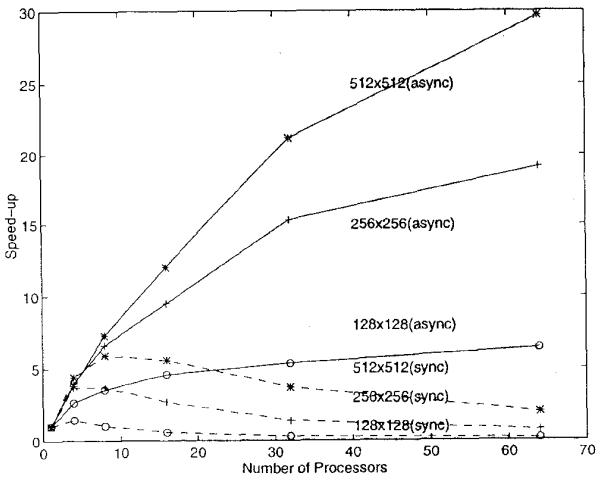


Fig. 4. Speed-up of linear feature extraction on SP-2.

Vision tasks often exhibit irregular data dependencies. Due to this, large synchronization overheads are incurred. Processors idle-wait while other processors compute results required for further processing. The *asynchronous algorithm technique* can reduce such overheads. In an asynchronous algorithm, processing nodes iterate without synchronizing between iterations. This elimination of synchronization overhead can improve processor utilization by reducing processor idle times [11].

When coarse-grained distributed memory architectures are used, accesses to remote data are much more expensive than local memory accesses. It is therefore important for data items that are needed by a computation to be located at the respective processing nodes. The data must be partitioned and distributed in such a manner that the overall communication time is minimized, and the workload is balanced [36], [44].

Since vision tasks consist of multiple distinct phases, the data are accessed differently in each phase. In such situations, data remapping aims to rearrange the data layout in between the phases so as to reduce the total execution time. In some scenarios, the overhead incurred in performing the remapping operation might be larger than the performance benefit due to remapping. These tradeoffs can be estimated using the computational model.

The following subsection illustrates the applicability of many of these techniques using a real world vision application as an example.

D. An Example Vision Application

This section explains the computational steps in an example vision application; a building detection system [25]. This is an integrated vision system, consisting of low, middle, and high-level phases. We choose such a complete application because it illustrates the applicability of our techniques to irregular problems arising in intermediate and high level vision. Most previous research efforts in using general purpose parallel machines have only considered low-level tasks.

1) Overview of the Computations: The input to the building detection system is an aerial image of buildings in urban and suburban environments and the output is a collection of buildings modeled as compositions of rectangular blocks [25]. The computational steps in this application can be divided into three main phases, as shown in Fig. 3. In the initial phase, primitive features such as line features are detected in the input image. The computation consists of structured processing such as, for instance, convolution operations. In the second phase, these primitive features are grouped together into higher level structures. For instance, line segments that are a very small distance apart, and run parallel to one another are grouped into a single line. In the third phase, high-level objects (rectangles) in the scene are recognized.

A serial implementation written in LISP produces good results, but the computational times on current workstations are large. The processing for a 1 K $\times$ 1 K image takes nearly 65 min to extract the line segments and 10 min to generate building hypotheses in the perceptual grouping phase, while processing a large image may take several hours [25]. To make the system really usable, computation times of at most a few seconds for small images, and of a few tens of minutes for large images are desired.

2) Parallel Techniques for Fast Line Finding: The linear feature extraction algorithm consists of two major phases: *contour-pixels detection* and *linear approximation*. Contour detection is performed to identify intensity boundaries (contours) in the 2-D raw image. Linear approximation is then performed to approximate the contours by line segments.

The contour-pixels detection phase consists primarily of *window operations* with localized data dependencies. Parallelizing this phase on a coarse-grain machine such as SP-2, can be easily done. The $n \times n$ image array is divided into P blocks, where P is the number of processors available. Each block is of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$. Initially, the boundary data

Table 2 Execution Time for Line Grouping on CM-5

Image ($n \times n$)	Number of Processors	Number of Segments	Execution Time (in *seconds*)		
			Communication Algorithm		
			Five Stage	Two Stage	One Stage
Air ($n = 128$)	64	320	.287	.341	.186
AP1 ($n = 256$)	64	1594	.460	.504	.361
J512 ($n = 512$)	64	3256	.934	.981	.814
J3 ($n = 1K$)	64	7219	1.436	1.462	1.295
Air ($n = 128$)	256	320	.363	.961	.477
AP1 ($n = 256$)	256	1594	.408	1.051	.539
J512 ($n = 512$)	256	3256	.524	1.177	.648
J3 ($n = 1K$)	256	7219	.633	1.255	.758

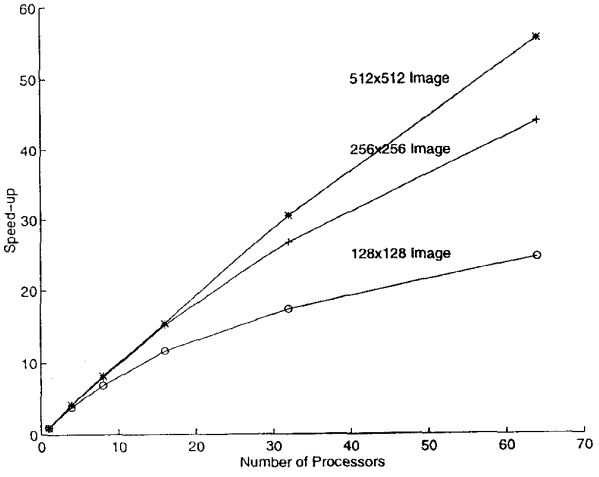


Fig. 5. Speed-up of the linear approximation phase on SP-2.

between neighboring image blocks is exchanged and stored in a local buffer. The window operations then proceed without further interprocessor communication.

In the linear approximation phase, irregular runtime data dependencies occur. The contours can be nonuniformly distributed over the image and approximation of a contour can proceed only after the previous segment of the contour at the neighboring processor has been processed. This results in load imbalances and poor processor utilization.

The asynchronous algorithm technique can improve load balance and processor utilization in this situation. We have developed an asynchronous algorithm for linear approximation, allowing each processing node to run independently of other processing nodes without violating any data dependency [11]. Fig. 4 shows the performance improvement achieved by such an asynchronous algorithm.

Our implementation shows that, linear feature extraction on a 512×512 image can be performed in 0.065 s on a 64 node SP-2. A serial implementation takes 3.45 s on a single processing node of the SP-2. A previous implementation on CM-5 takes 0.1 s on a partition having 512 processing nodes. Fig. 5 shows the speedup diagram for this task on a SP-2. The code was written using MPI [39] and is therefore portable to other parallel machines. (Intel Paragon, Meiko CS-2, Cray T3D, etc.) We have ported it to SP-2 and CM-5 [11].

3) Parallel Techniques for Perceptual Grouping: The perceptual grouping process operates on the set of primitive features detected in the low-level analysis and groups them to form structural hypotheses. These hypotheses are further used for high-level reasoning.

Perceptual grouping consists of four stages: line grouping, junction grouping, parallel grouping, and U-contour grouping. The line grouping operation groups line segments which are closely bunched, overlapped, and parallel to each other to form a *line*. For each line segment, a search is performed within a constant width window on both sides of it, to find other parallel line segments. The input to each processor is an image block of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ along with line segments extracted during linear feature extraction. A segment token with starting coordinates (x_1, y_1) and ending coordinates (x_2, y_2) is stored in the processor which has the image block containing (x_2, y_2). The grouping process is performed in a scatter-and-gather fashion. In the scatter phase, each processor computes the search window for each token stored locally in the processor. Search requests are then sent to the processors which have index array elements that overlap the search window. The search requests look for line segments which are approximately parallel to the local segment. The detected segments are then grouped and sent back in the gather phase. A *merge* operation is then performed to form *lines*.

The message packing technique can reduce the communication overhead in this operation. All requests from a processor with the same destination are packed into a single message. The resulting communication pattern is a many-to-many personalized communication with message length variance. Using decomposition and communication scheduling techniques, we have developed a five-stage algorithm which can reduce the communication time for many-to-many data communication with high message length variance [62]. This performs better than a straightforward one-stage algorithm and a two-stage algorithm proposed in [50], especially for larger number of processors and for higher variance in the message length. In Table 2, we present sample performance results of the line grouping operation on CM-5.

Thus it can be seen that the algorithmic approach can yield significant performance improvements, and thereby lead to scalable parallel implementations of computational tasks in various stages of the vision process. Careful

analysis of the overheads and structuring of the algorithm to overcome these overheads are important steps to achieving high performance.

V. Conclusion

This paper has given an overview of the state-of-the-art in high performance computing for vision. We observe that vision tasks have computational characteristics significantly different from those of Grand Challenge problems in scientific application domains. The vision process consists of a sequence of tasks, each with distinct computational characteristics. Low-level vision has a structured computational profile, with regular data dependencies. At the higher levels, extensive symbolic operations with irregular data access patterns are present. This heterogeneous nature of the computations makes it difficult for a single computing platform to achieve desired performance levels for the entire vision process.

Over the years, researchers have used several different approaches based on various computing platforms for high-performance vision. Custom VLSI based solutions have been successfully used primarily for low level vision problems. Specialized vision systems have also been designed with architectural features that suit the requirements of vision tasks. However, such systems have not been widely used due to high costs and long design times.

Economic and engineering considerations point to the use of general purpose distributed memory parallel machines as computing platforms for vision applications. Current generation general purpose machines are designed around low-cost COTS components and have a MIMD architecture. These systems offer advantages of architectural flexibility and programmability. Due to their general purpose architecture, they provide suitable computing support for a variety of vision tasks at each of the different levels of vision processing. Parallel solutions for low-level tasks have been developed and implemented on these systems. However, developing high speed parallel solutions for the higher level tasks is not straightforward. The irregular data dependencies of these tasks lead to excessive communication overheads, when they are implemented on the distributed memory systems. These overheads result in inefficient processor utilization and low speed ups.

We have used an algorithmic approach to develop high performance parallel solutions for vision tasks on the general purpose machines. A computational model of the parallel system is used to analyze parallelization overheads, and thereby design scalable algorithms. With the assistance of the model, we have designed scalable parallel algorithms for many generic tasks in vision processing. We have implemented these algorithms on current generation parallel machines, with promising results.

Many challenging issues remain to be solved before practical real time vision systems can be realized. The application environments for real-time systems have special constraints such as low-power consumption and small system size. Most currently available general purpose parallel machines have large power requirements and are bulky. Research in multichip modules (MCM's) and in low power microsystems design aims to advance COTS component technology so that they can be used in such embedded parallel systems. On the software side, standards such as real-time MPI are currently evolving. Efficient implementations of these standards on the computing platforms must be developed, so that efficient parallel algorithms can be designed in a truly portable manner.

Acknowledgment

The authors thank the Vision group, headed by R. Nevatia at the University of Southern California, for many helpful discussions and for their assistance in understanding vision computations. The implementations reported in this paper were performed at the Minnesota Supercomputer Center, the Pittsburgh Supercomputing Center, and the Maui High Performance Computing Center. The authors are grateful to the authorities at these organizations for providing access to their computing resources. They also thank Y. Chung and C.-C. Lin for helpful discussions.

References

[1] H. Alnuweiri and V. Prasanna, "Parallel architectures and algorithms for image component labeling," *IEEE Trans. Patt. Anal. and Mach. Intell.*, pp. 1014–1034, 1992.

[2] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *Proc. ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation*, pp. 112–125.

[3] G. Astfalk, "Parallel programming on the convex MPP," in *Proc. IEEE Oceans*, 1994, pp. 1–6.

[4] P. Athanas and A. Abbott, "Addressing the computational requirements of image processing with a custom computing machine: An overview," in *Proc. Workshop on Reconfigurable Architectures*, Santa Barbara, CA, 1995.

[5] D. Bader and J. Já Já, "Parallel algorithms for image histogramming and connected components with an experimental study," Tech. Rep., Univ. Maryland, Dec. 1994.

[6] I. Banicescu and S. Hummel, "Balancing processor loads and exploiting data locality in irregular computations," *IBM Res. Rep.*, Feb. 1995.

[7] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," in *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, 1992, pp. 13–22.

[8] A. Beguelin *et al.*, "A users' guide to PVM: Parallel virtual machine," Tech. Rep. ORNL/TM-11826, Oak Ridge Nat. Lab., July 1991.

[9] N. J. Boden *et al.*, "Myrinet—A gigabit-per-second local-area network," *IEEE Micro*, Feb. 1995.

[10] Z. Bozkus *et al.*, "Compiling fortran 90D/HPF for distributed memory MIMD computers," *J. Parallel and Distrib. Comput.*, Special Issue on Data Parallel Algorithms and Programming, vol. 21, no. 1, Apr. 1994.

[11] Y. Chung, V. K. Prasanna, and C.-L. Wang, "A fast asynchronous algorithm for linear feature extraction on IBM SP-2," in *IEEE Workshop on Computer Architectures for Mach. Perception*, Sept. 1995.

[12] Convex Computer Corp., "SPP 1000 systems overview," 1994.

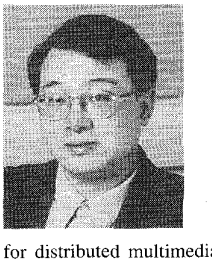
[13] Cray Res., Inc., "Cray T3D System architecture overview," Sept. 1993.

[14] D. Culler *et al.*, "*LogP*: Toward a realistic model of parallel computation," in *Proc. 4th ACM SIGPLAN Symp. Principles and Practices of Parallel Programming*, 1993, pp. 1–12.

[15] E. W. Dijkstra, "Cooperating sequential processes," *Programming Languages*. New York: Academic, 1966.

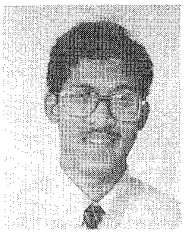
[16] Dolphin Interconnect Solutions, Inc., "1 Gbit/sec SBus-SCI cluster adapter card," White Paper, Mar. 1995.

[17] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proc. 19th Annu. Int. Symp. on Computer Architecture*, 1992, pp. 256–266.
[18] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proc. 10th ACM Symp. on Theory of Computing*, 1978, pp. 114–118.
[19] D. Gerogiannis and S. Orphanoudakis, "Load balancing requirements in parallel implementations of image feature extraction tasks," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 4, pp. 994–1013, Sept. 1993.
[20] "Grand challenges: High performance computing and communications," The FY 1992 U.S. Res. and Develop. Program, Comm. on Physical, Mathematical, and Engineering Sci., Federal Coordinating Council for Sci., Engineering, and Technol., Office of Sci. and Technol. Policy, Washington, DC, 1992.
[21] C. Guerra, "Survey of parallel algorithms for structural pattern matching," *Int. Conf. on Patt. Recog.*, Sept. 1994, pp. 275–278.
[22] T. Heywood and S. Ranka, "A practical hierarchical model of parallel computation: The model," *J. Parallel and Distrib. Computing*, vol. 16, pp. 212–232, 1992.
[23] High Performance Fortran Forum, "High performance Fortran language specification," Jan. 1993.
[24] C. A. R. Hoare, "Monitors: An operating system structuring concept," *CACM*, vol. 21, no. 8, Aug. 1978.
[25] A. Huertas, C. Lin, and R. Nevatia, "Detection of buildings from monocular views of aerial scenes using perceptual grouping and shadows," *Image Understanding Workshop*, 1993, pp. 253–260.
[26] IBM, *Parallel Programming Subroutine Reference Release 2.0*, 1994.
[27] J. Já Já and K. Ryu, "The block distributed memory model for shared memory multiprocessors," in *Proc. Int. Parallel Process. Symp.*, 1994, pp. 752–756.
[28] V. Karamcheti and A. Chien, "Software overhead in messaging layers: Where does the time go?" in *Proc. Int. Parallel Processing Symp.*, ASPLOS VI, 1994, pp. 51–60.
[29] A. Khokhar, "Scalable data parallel algorithms and implementations for object recognition," Ph.D. dissertation, Dept. EE-Syst., Univ. S. Calif., Apr. 1993.
[30] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*. New York: Benjamin/Cummings, 1994.
[31] S. Kumaran and M. Quinn, "Divide-and-conquer programming on MIMD computers," *Int. Parallel Process. Symp.*, 1995, pp. 734–741.
[32] T. Kwan, B. Totty, and D. Reed, "Communication and computation performance of the CM-5," in *Proc. of Supercomputing '93*, 1993, pp. 192–201.
[33] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, CA: Morgan Kaufmann, 1992.
[34] C. A. Lin, A. Huertas, and R. Nevatia, "Detection of buildings using perceptual organization and shadows," in *Proc. 1994 Conf. Computer Vision and Patt. Recog.*, pp. 62–69.
[35] C.-C. Lin and V. K. Prasanna, "Analysis of communication costs of parallel machines with various communication mechanisms," in *Proc. 5th Symp. on Frontiers of Massively Parallel Comput.*, 1995.
[36] C.-C. Lin, V. K. Prasanna, and Y. Chung, "Data remapping for intermediate level analysis in image understanding on distributed-memory mahines," in *Workshop on Solving Irregular Problems on Distrib. Memory Machines, IPPS'95*, 1995, pp. 35–42.
[37] W. Lin and V. K. Prasanna, "Parallel algorithms and architectures for discrete relaxation technique," in *IEEE Conf. Computer Vision and Patt. Recog. (CVPR)*, 1991.
[38] H. Lu and J. K. Aggarwal, "Applying perceptual organization to the detection of man-made objects in nonurban scenes," *Patt. Recog.*, pp. 835–853, 1992.
[39] Message Passing Interface Forum, "MPI: A message-passing interface standard," Tech. Rep. CS-94-230, Univ. Tennessee, Knoxville, TN, May 1994.
[40] R. Miller, V. K. Prasanna, D. Reisis, and Q. Stout, "Parallel computations on reconfigurable meshes," *IEEE Trans. Computers*, pp. 678–692, June 1993.
[41] P. Narayanan, L. Chen, and L. Davis, "Effective use of SIMD parallelism in low- and intermediate-level vision," *IEEE Computer*, vol. 25, no. 2, pp. 68–73, 1992.
[42] R. Nevatia, *Machine Perception*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
[43] R. Nevatia and K. Babu, "Linear feature extraction and description," *Computer Graphics and Image Processing*, vol. 13, pp. 257–269, 1980.
[44] C. Ou and S. Ranka, "Parallel remapping algorithms for adaptive problems," in *Proc. Symp. Frontiers of Massively Parallel Computation*, 1995, pp. 367–374.
[45] V. P. Kumar, *Parallel Algorithms and Architectures for Image Understanding*. Boston: Academic, 1991.
[46] V. K. Prasanna and V. Krishnan, "Efficient parallel algorithms for template matching on hypercube SIMD machines," in *Proc. Int. Conf. on Parallel Process.*, Aug. 1987.
[47] V. K. Prasanna and C.-L. Wang, "Image feature extraction on connection machine CM-5," in *Image Understanding Workshop*, Nov. 1994, pp. 595–602.
[48] ——, "Scalable parallel implementations of perceptual grouping on connection machine CM-5," in *Int. Conf. on Patt. Recog.*, 1994.
[49] V. K. Prasanna, C. Wang, and A. Khokhar, "Low level vision processing on connection machine CM-5," in *Workshop on Computer Architectures for Machine Perception*, 1993, pp. 117–126.
[50] S. Ranka, J. Wang, and M. Kumar, "Irregular personalized communication on distributed memory machines," *J. Parallel and Distrib. Computing*, vol. 25, pp. 58–71, 1995.
[51] *SCI-Scalable Coherent Interface Standard*, ANSI/IEEE 1596-1992.
[52] *SHMEM Techn. Note for C*, Cray Res. Inc., 1994.
[53] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer vision related tasks onto reconfigurable parallel processing systems," *IEEE Computer*, pp. 54–63, Feb. 1992.
[54] Silicon Graphics, Inc., *Power Challenge Technical Report*, 1994.
[55] R. M. Haralick *et al.*, "Proteus: A reconfigurable computational network for computer vision," in *Int. Conf. on Patt. Recog.*, 1992.
[56] J. Subhlok *et al.*, "Communication and memory requirements as the basis for mapping task and data parallel programs," *Proc. Supercomput.*, 1994.
[57] Thinking Machines Corp., *CMMD Ref. Guide Version 3.0*, 1992.
[58] ——, "CM-5: Technical summary," Tech. Rep., 1991.
[59] L. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[60] C.-L. Wang, "High performance computing for vision on distributed memory machines," Ph.D. dissertation, Univ. S. Calif., Aug. 1995.
[61] C.-L. Wang, V. K. Prasanna, H. Kim, and A. Khokhar, "Scalable data parallel implementations of object recognition using geometric hashing," *J. Parallel and Distrib. Comput.*, Mar. 1994.
[62] C.-L. Wang, V. K. Prasanna, and Y. Lim, "Parallelization of perceptual grouping on distributed memory machines," in *IEEE Conf. Computer Architec. for Machine Perception*, Sept. 1995.
[63] J. Webb, "High performance computing in image processing and computer vision," in *Int. Conf. on Patt. Recog.*, Sept. 1994, pp. 218–222.
[64] C. C. Weems, S. P. Levitan, A. R. Hanson, and E. M. Riseman, "The image understanding architecture," *Int. J. Computer Vision*, vol. 2, pp. 251–282, 1989.



Cho-Li Wang received the B.S. degree is computer science and information engineering from the National Taiwan University in 1985. He received the M.S. and Ph.D. degrees in computer engineering from the University of Southern California in 1990 and 1995, respectively.

He is now a Lecturer in the Computer Science Department at Hong Kong University. His research interests include parallel processing for image understanding, high performance computer architectures, and software environments for distributed multimedia applications.



Prashanth B. Bhat received the B.Tech degree in computer engineering from the Karnataka Regional Engineering College, India, in 1992. He received the M.E. degree in computer science and engineering from the Indian Institute of Science, Bangalore, in 1994. He is presently working toward the Ph.D. degree in computer engineering at the University of Southern California at Los Angeles.

His research interests include computer architecture, parallel algorithm design for signal and image processing applications, heterogeneous computer systems, and configurable computing.

Viktor K. Prasanna (Fellow, IEEE) received the B.S. degree in electronics engineering from Bangalore University and the M.S. degree in automation from the Indian Institute of Science. He received the Ph.D. in computer science from the Pennsylvania State University in 1983.

He is currently Professor of Electrical Engineering at the University of Southern California at Los Angeles. His research interests include parallel computation, computer architecture, VLSI computations, and high performance computing for signal and image processing, and vision. He has served on the editorial boards of the *Journal of Parallel and Distributed Computing* and the IEEE TRANSACTIONS ON COMPUTERS.

Dr. Prasanna is the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the general co-chair of the 1997 IEEE International Parallel Processing Symposium.