Software Development Tools for Implementing Vision Systems on Multiprocessors

Alok Choudhary  
Department of Electrical and Computer Engineering  
Syracuse University  
Syracuse, NY, 13244

Sanjay Ranka  
School of Computer and Information Science  
Syracuse University  
Syracuse, NY, 13244

ABSTRACT

Parallel processing has been widely accepted as the approach to providing the necessary computational power to solve computer vision systems problems. Although several projects are underway to develop new architectures for computer vision, tools to effectively use those systems or commercially available multiprocessors are limited or non-existent. Unless we can develop efficient methods for mapping vision algorithms and developing programs on these architectures, the performance gains from parallel processing will be limited, and will be beyond the reach of a non-expert in parallel processing. This paper presents a design for a software development environment (SDE) for implementing vision systems applications on multiprocessors. The SDE design exploits characteristics of vision systems, and uses a classification scheme for vision algorithms to develop a parallelization and performance evaluation tool. These tools use databases that store knowledge of parallelization for different known computations on common architectures. The parallelization and performance evaluation tools use this knowledge to guide a user interactively parallelize algorithms. Some parts of SDE are currently operational and others still need to be developed.

1 INTRODUCTION

Computer vision has been regarded as one of the most complex and computationally intensive problems. A typical Computer Vision System (CVS) employs algorithms from a very broad spectrum of areas such as numerical, signal processing, image processing, graph algorithms, symbolic processing and artificial intelligence. Parallel processing has been widely accepted as the approach to providing the necessary computational power to solve CVS problems in a reasonable time.

A typical CVS using color images requires a processor capable of handling 23 megabytes of input data per second, interpreting to construct a three dimensional model of the environment [1]. An interpretation may require hundreds of objects of different types to be identified [2]. 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 [3, 4]. Complicating factors also include the presence of noise, occlusion, uncontrollable imaging environment, shadows, and motion [1]. Vision researchers have shown that pattern recognition techniques and bottom-up processing (sensory) alone is not adequate for the above tasks [5]. Vision also involves top-down and knowledge-based processing. Between these two levels of abstractions, another level is normally introduced, known as “intermediate level” [1]. 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.

Although several projects are underway to develop new architectures for computer vision, tools to effectively use those systems or commercially available general purpose architectures are limited or non-
existing. Unless we can develop efficient methods for mapping vision algorithms and programs on these architectures, the performance gains will be limited. Hence, there is a need to develop an environment and tools that assist users to design, implement and evaluate their applications on multiprocessor architectures.

For achieving the above goals, we need to provide tools that suggest efficient techniques to map an application program (given an algorithm and a general solution paradigm) on several architectures. This task includes examining the existing techniques, and suggest ways to improve upon them; propose new approaches and evaluate them. Several issues such as parallel algorithm selection (if not already given), partitioning, load balancing, communication and input-output of data and results need to be addressed. In conjunction with the mapping tools, we need software tools that will aid the user in predicting the performance of a program on various architectures. Using these tools the user may be able to redesign a program or algorithm to improve efficiency of their application implementation.

In this paper we present a design for a software development environment (SDE) for implementing vision systems applications on multiprocessors. The SDE design exploits characteristics of vision systems and uses a classification scheme for vision algorithms to develop databases that store knowledge of parallelizations for various known computations on common architectures. The parallelization and performance evaluation tools use this knowledge to guide a user interactively parallelize algorithms. Some parts of SDE are currently operational and others still need to be developed.

The rest of this paper is organized as follows. Section 2 presents characteristics of vision algorithms. The design of the Software Development Environment is presented in Section 3. An example problem implementation and its performance is presented in Section 4. Finally, summary and conclusions are presented in Section 5.

2 CHARACTERISTICS OF VISION ALGORITHMS

2.1 Parallelism in CVSs

Available parallelism in computer vision system algorithms can be placed in two broad categories: namely, Spatial and Temporal Parallelism.

Spatial Parallelism is one in which similar operations are applied in all parts of the image data. That is, the data can be divided into many granules and distributed to subtasks which may execute on different processors in parallel. Most vision algorithms exhibit this type of parallelism. In an CVS, each task operates on the output data of the previous task in the system. Therefore, the type of data, and data structures may be different for each task in the system but each form of data can be partitioned into several granules to be processed in parallel. For example, consider an CVS that performs object recognition. The input image is smoothed using some filtering operation, then on the smoothed image an operator is applied for feature extraction, features with similar characteristics are grouped, then matching with the models is performed. Each of these tasks takes the output of the previous tasks as its input and produces an output which becomes the input for the next task. Note that within spatial parallelism, depending on the computation involved, an algorithm implementation may be suitable for data parallelism or task parallelism or both.

Temporal Parallelism is available when these tasks are repeated on a time sequence of images or on different resolutions of images. For example, the system in which motion of a moving object is estimated takes a sequence of images of the moving object and performs the same set of computation on all image frame(s). The processing of each frame or a set of frames can be done in parallel with the processing of frames of other time instances.
2.2 Data dependencies

Existence of spatial and temporal parallelism may also result in two types of data dependencies, namely, *spatial data dependency* and *temporal data dependency*. Spatial data dependency can be classified into intratask data dependency and intertask data dependency. Each task itself is a collection of subtasks which may be represented as a graph with nodes representing the subtasks and edges representing communication between subtasks. Intratask data dependencies arise when a set of subtasks needs to exchange data in order to execute a task in parallel. The exchange of data may be needed during the execution of the algorithm, or to combine the partial results, or both. Intertask data dependency denotes the transfer and reorganization of data to be passed onto the next task in the pipeline. The mode of communication may be subtasks of the current tasks to the subtasks of the next task, or collection and reorganization of the output data of the current task and then redistribution of the data for the next task. The choice of methods depend on the underlying parallel architecture, mapping of algorithms and input-output relationships between tasks. The set of algorithms which perform the reorganization of data are crucial to exploit the available parallelism. We will call these algorithms as Data Conversion Algorithms.

Temporal data dependency is similar to spatial data dependency except that some form of output generated by tasks executed on the previous image frames may be needed by one or more tasks executing on the current image frames.

2.3 A classification of vision algorithms

Vision algorithms are normally classified into three levels: low (sensory, image processing), intermediate (symbolic processing), and high (knowledge-based). From a computer architects point of view, the computational requirements can be summarized as shown in Table 1 [1].

Algorithms that constitute an integrated vision systems exhibit different characteristics, and therefore, require different data decomposition techniques and efficient load balancing techniques for parallel implementation. Since the input data of a task is produced as the output data of the previous task, this information can be exploited to perform knowledge based data decomposition and load balancing.

Most vision algorithms can be classified into the following four classes:

1. Data Independent and perform Local Operations - Low Level Algorithms
2. Data Independent and perform Global Operations - Low and Intermediate Level Algorithms
3. Data Dependent and perform local Operations - Intermediate level Algorithms
4. Data Dependent and perform Global Operations - Intermediate Level and High Level Algorithms

Each of these classes require similar data decomposition and load balancing techniques for mapping them on multiprocessors. For data independent and local operations, an obvious and simple method to implement a task in parallel is to decompose the data and equally among the processors. Such schemes perform well, and normally the processing time is comparable on all the processors. That is, efficient utilization and load balancing can be obtained. For example, regular algorithms such as convolution, filtering or FFT exhibit such properties. The amount of computation to obtain each output point is the same across all input data. Therefore, uniform decomposition of data results in load balanced implementation. This also results in communication patterns that are uniform and regular. Also, since the computations are local in nature the communication patterns required are also local in nature. For example, convolution requires communication between processors having proximate parts of an image. Thus embeddings of the partitions in an architecture which will maintain proximity of adjacent partitions will result in low communication cost.
<table>
<thead>
<tr>
<th></th>
<th>Low Level</th>
<th>Intermediate Level</th>
<th>High Level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation</strong></td>
<td>• Fine/medium grained</td>
<td>• Medium grained</td>
<td>• Coarse grained</td>
</tr>
<tr>
<td></td>
<td>• 256K 8-bit pixels</td>
<td>• Thousands of &quot;tokens&quot;</td>
<td>• Hundreds of &quot;volumes&quot;</td>
</tr>
<tr>
<td></td>
<td>• Integer arithmetic</td>
<td>• Integer arithmetic</td>
<td>• Integer arithmetic</td>
</tr>
<tr>
<td></td>
<td>• Limited real arithmetic</td>
<td>• Real arithmetic</td>
<td>• Real arithmetic</td>
</tr>
<tr>
<td></td>
<td>• Comparisons</td>
<td>• Maintaining lists of token relationships</td>
<td>• List travels</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>• Local neighborhood</td>
<td>• Local neighborhood</td>
<td>• Blackboard access</td>
</tr>
<tr>
<td></td>
<td>• Across connected components</td>
<td>• Long distance</td>
<td>• Control info to lower level</td>
</tr>
<tr>
<td></td>
<td>• Structured patterns</td>
<td>• Broadcast</td>
<td>• Queries to lower level</td>
</tr>
<tr>
<td></td>
<td>• Broadcast</td>
<td>• Down and up</td>
<td>• Data up from lower level</td>
</tr>
<tr>
<td></td>
<td>• Up</td>
<td>• Summary feedback</td>
<td>• Coarse grained</td>
</tr>
<tr>
<td></td>
<td>• Summary feedback</td>
<td>• Medium length messages</td>
<td>• Coarse grained</td>
</tr>
<tr>
<td></td>
<td>• High-speed I/O</td>
<td></td>
<td>messages</td>
</tr>
<tr>
<td></td>
<td>• Fine-grained messages</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>• SIMD</td>
<td>• SIMD</td>
<td>• MIMD</td>
</tr>
<tr>
<td></td>
<td>• Multi-SIMD</td>
<td>• Synchronous MIMD</td>
<td>• Distributed control</td>
</tr>
<tr>
<td></td>
<td>• Central control</td>
<td>• MIMD directed by higher level</td>
<td>• Attention focusing mechanism</td>
</tr>
<tr>
<td></td>
<td>• Local activity control</td>
<td>• Central and local control</td>
<td>• Coordination with central control of lower</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>levels</td>
</tr>
</tbody>
</table>

Table 1: Computation and communication requirement classification of vision systems algorithms

The second class of algorithms are data independent and require global operations. Typically, this requires some amount of data independent local computation followed by a global operation. Again, an obvious and simple method to implement a task in parallel is to decompose the data equally among the processors. The communication pattern for the global operation is also uniform and regular. For example, histogramming requires building up of local histograms and combining them to give the histogram of the whole image.

The algorithms in the other two categories do not exhibit a regular structure, and the computation is data dependent. Hence, the computation is not uniformly distributed across the input domain. In such cases, a simple decomposition does not provide efficient mapping and results in poor utilization and low speedups. Many image processing and vision algorithms exhibit this behavior. For example, in hough transform, the computation is proportional to the number of features (edges) or significant pixels in a granule. Therefore, equal size granules do not guarantee load balanced partitioning because of the data dependent nature of the computation.

If the data is not spatially related (e.g., in case of hough transform the spatial relationship between edges does not affect the computational requirements), the amount of work required on each granule can be estimated. This will typically depend on the size of granule (fixed overhead) and the number of significant data in the granule. The information for each granule can be used to estimate the total amount of work required. The load can then be uniformly distributed. This load balancing can be performed easily as the work can be distributed to every processor without caring about the spatial relationships. An example of hough transform implementation is described in Section 4.

When the computation in a granule not only depends on the number of significant data points in the input domain but also depends on their spatial relationships, then data distribution also needs to be taken into account as a measure of load to perform load balancing. For example, to perform stereo...
match, not only does the computation depend on the number of features but also depends on their spatial distribution. If the features are densely spaced then the computation will be more than that if the same number of features is sparsely distributed. Hence, the computation also depends on the spatial density (such as features/row if one-dimensional matching is performed [3]). While partitioning the data among processors, a weight can be assigned to each row as a function of the number of features in the row. This weight represents the feature density. This information can be used to calculate the amount of computation required in each partition.

Table 2 provides a summary of different partitioning and load balancing schemes for different classes of problems. Of course, this is not a complete list by any means. As algorithms are developed and new schemes are obtained, they will be added to this table.

<table>
<thead>
<tr>
<th>Class</th>
<th>Sub-Class</th>
<th>Partitioning</th>
<th>Example Problem(s)</th>
<th>Load Balancing</th>
<th>Communication Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Independent</td>
<td>Computation $\propto$ Size of Data</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Relaxation, Template Matching, Smoothing, Thresholding, Sobel, Zero Crossings</td>
</tr>
<tr>
<td>Local Operations</td>
<td></td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td></td>
</tr>
<tr>
<td>Global Operations</td>
<td></td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td></td>
</tr>
<tr>
<td>Data Independent</td>
<td>Computation $\propto$ Size of Data, Spatial Constraints</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Load Balanced 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Thinning, Stereo Match, Surface Fitting</td>
</tr>
<tr>
<td>Local Operations</td>
<td></td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td></td>
</tr>
<tr>
<td>Global Operations</td>
<td>Computation $\propto$ Size of Significant Data, no Spatial Relationship</td>
<td>Uniform 1-D Array</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Hough Transform, No Spatial Constraints</td>
</tr>
<tr>
<td>Data Dependent</td>
<td>Computation $\propto$ Size of Significant Data, Spatial Density</td>
<td>Uniform 1-D or 2-D Array</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Connected Component Labelling, Region Growing, Static, Spatial Constraints</td>
</tr>
<tr>
<td>Local Operations</td>
<td></td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td></td>
</tr>
<tr>
<td>Global Operations</td>
<td>Computation $\propto$ Size of Significant Data, Models, Database</td>
<td>Uniform 1-D or 2-D Array</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td>Uniform 1-D or 2-D Blocks</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Partitioning and load balancing required for different classes of vision algorithms
3 SOFTWARE DEVELOPMENT ENVIRONMENT

Figure 1 illustrates various modules of the software tool for interactive parallelization of vision systems. The following is a brief description of the modules and their interaction with each other.

3.1 Application description interface (ADI)

The first step in developing a parallel program for an application is to specify the application. A user can specify the application using a specification language [6], an existing sequential code, a functional language [7, 9], or a graphics-based language [11]. In the above environment, the user can input a sequential program or task graphs. This has been used in parallelizing compilers. In this environment, we would support both shared and distributed memory machines. Inter-task dependencies define the order of execution and interaction between tasks for coarse-grain as well as fine-grain tasks. The ADI will have the capability to interactive guide the user to construct task flow and data flow graphs for the input algorithm or program. Also, the user will be able to interactively change the granularity of tasks by splitting or combining nodes of a task graph. Functional and dataflow languages (such as VAL, SISAL, LGDF) [7, 8, 9, 10] can be used to represent the application developed in this manner. The output of ADI will be input to the Interactive parallelization tool.

![Diagram of Software Development Environment]

Figure 1: Software development environment
3.2 Interactive parallelization tool (IPT)

The IPT has several inputs to interactively guide users parallelize their applications. The architecture database (to be described later) will input the characteristics of the architecture on which the user wishes to implement the task. The algorithm database will input the mapping information regarding the input task or a task with similar characteristics. With these inputs along with the input taskflow and dataflow information, the IPT will interactively guide the user to parallelize the task on the machine model obtained from the architecture database. The IDT will help the user partition the nodes of the task graphs onto the multiprocessor nodes. Initial mapping may be a crude one, and several iterations may be required to refine it. The refinement steps will also use information from the performance evaluation tool that will have capabilities to do simple performance analysis using the information from various databases. Once the mapping has been done, mapping verification and performance prediction will be performed to check the validity and efficiency respectively. Analytical models can be built into the performance evaluation tools to predict the performance [12]. Some tools have been developed for performance evaluation and prediction at Syracuse University [13] and will be incorporated in the above tool.

3.3 Architecture database (AD)

The architecture database (AD) contains the machine level characterization of various machines. Information such as interconnection network, memory organization, communication or synchronization cost, computation cost etc. will be stored and accessible through IPT. Therefore, a user will be relieved of the burden of having an indepth knowledge of the architectures. Figure 2 shows the top level of the architecture characterization in AD. Each machine is characterized by its functional partitioning using Computation, Input/Output, Data Movement and Control [13]. Each of these categories is further classified using static (from benchmarks) characteristics and dynamic characteristics (from analytical models). For example, in a coarse-grain shared memory machine (such as encore multimax), the CPU time or computation time for each processor for primitive operations can be estimated using benchmark applications; however, bus latency, bus traffic etc. must be incorporated dynamically because they depend on the request rate and dynamic characteristics of an applications. This module is operational at Syracuse University and reader is referred to [13] for further details.

3.4 Algorithm characteristics database (ACD)

The Algorithm Characteristic Database contains a classification of algorithms, their computational and communication characteristics based on the characterization discussed in the previous section. Furthermore, it contains directions for parallelizing the algorithms and their suitability for certain types of parallelization (e.g., divide-and-conquer) paradigms, and suitable load balancing techniques. For most of the common algorithms, analytic performance measures will be stored for various parallelizations on different architectures. ACD will also include analytic measures for common primitives such as window based operators, transformations, simple image operations etc. in parameterized forms (e.g., as a function of window size for window based operators) so that if they are used in an input algorithms that is being parallelized, they can be simply incorporated in the parallelization process. New classes of algorithms and primitives can be added to ACD as and when they are identified and used. The classification will be done using the common characteristics of the primitives and algorithms. This classification will be described in a form which the user can read before parallelizing a new algorithm and make an informed decision to direct the tool to start with a classification that most likely describes the new algorithm. This will help provide the tool a better starting point for a particular parallelization.
Primitives

Many simple vision algorithms can potentially be represented by a primitive or a set of primitives. These primitives are sets of frequently used algorithms. The ACD also stores information about the runtime characteristics, input-output relationships and executable codes for a primitive. There can be many sets of executable codes for the same primitive depending upon the underlying architecture and input-output relationships. An example of a primitive is given in Table 3. The codes of many primitives may be similar (and, or, sum, synchronization) with different parameter values.

3.5 Performance measurements

The function of this tool is to provide execution time performance measurements of an application. The execution profile will be stored in the form of a trace that will be analyzed by the performance evaluation tool. The run-time traces can be obtained at several levels of a program such as program level, processor level, process level, procedure or function level and primitive activity level [14]. The trace can be obtained by inserting probes in the application interactively by an user. The probes are normally compiler directives. For example, if the performance measurements are to be taken at procedure or function level, the probes can be inserted just before a procedure or function is entered and just after it is exited. Therefore, the trace will contain the timing as well as other information (such as frequency) about all the procedures and functions, their execution order, the amount of time spent each time it is entered. The hierarchy of performance measurements allows a user to obtain only that information that may be of interest rather than being overwhelmed with all possible performance data. For example, if the user is only interested in send and receive primitives for a distributed memory multiprocessor program, then the probes can be inserted before and after all send and receive primitives. The execution trace will contain computation and communication phases for all the processes in the system. This trace can be subsequently analyzed by the performance evaluation tool to study the communication behavior of the parallel application.
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Sum</th>
<th>Sum</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Size</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Network</td>
<td>Mesh</td>
<td>Hypercube</td>
<td>Tree</td>
</tr>
<tr>
<td>Computation Time</td>
<td>$k(\log P + N/P)$</td>
<td>$k(\log P + N/P)$</td>
<td>$k(\log P + N/P)$</td>
</tr>
<tr>
<td>Communication Time</td>
<td>$2\log P$</td>
<td>$t \cdot \log P$</td>
<td>$t \cdot \log P$</td>
</tr>
<tr>
<td>Maximum Number of</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Processors</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input: Array $[1 \ldots N]$  
Output: One Element  
Program: Program for a Mesh Hypercube for a Tree  

| Table 3: A simple primitive |

### 3.6 Performance evaluation tool (PET)

Performance Evaluation Tool has several functions such as performance analysis and prediction, performance evaluation using measurement traces, and providing feedback to the user through visualization. Performance analysis and prediction provides some initial and rough predictions when an application is interactively parallelized [12].

The most difficult decisions to be made by a programmer in distributed memory parallel machines are those involving data partitioning strategies and message-passing communication. For the shared memory parallel machines, the decisions are regarding partitioning, synchronization and shared variables. Performance Evaluation Tool (PET) gives the user assistance in this matter. It can be useful in discriminating between different partitioning strategies and message passing schemes. It can allow a programmer to explore different algorithmic techniques and assess their relative performance impacts at program development time. This will provide the user to focus on selected program segments and improve their performance in incremental steps.

The performance prediction function uses information from the algorithm characteristic database, architecture database and performance database. Information from ACD and AD helps the performance prediction process to characterize the given application to one of the classes or algorithms (or to the most similar algorithm) stored in ACD. The ACD information, in conjunction with the architectural characteristics and the performance information from the performance database about the chosen algorithm, is used to predict the performance of the given application.

The performance evaluation function uses the trace information from performance measurements of the given application execution and shows the dynamic behavior of the application to the user. The level of detail depends on the level at which the measurements were obtained as specified by the user. The evaluation can help the user identify the bottlenecks and refine the parallelization using this feedback mechanism. Hierarchical evaluation enables the user to perform refine the parallelization in a coarse-to-fine manner because the user can start with the application level performance evaluation and go down to primitive level performance evaluation, each time refining the parallelized version using the feedback from PET.
3.7 Machine interface

The machine interface module provides to bridge the gap between logical implementation produced by IPT and the syntax of the target machine implementation. Furthermore, it also provides and interface to the vendor supplied operating system and software. For this purpose, we plan to use some of the existing platforms such as Express [16] that provides syntax and other machine and programming language dependent features for several multiprocessor machines.

4 EXAMPLE PROBLEM

This section presents steps in parallelization of and performance results for hough transform algorithm which is a global data dependent algorithm. Hough Transform is used to transform the edges to another space, called the Hough space, with the property that the desired group of edges cluster together in the transformed space. Let \( I[0 \ldots N - 1, 0 \ldots N - 1] \) be an \( N \times N \) image such that \( I[x, y] = 1 \) iff the image point \([x, y]\) is a possible edge point. \( I[x, y] = 0 \) otherwise. The \( p \) angle Hough transform of \( I \), to detect straight lines in an image, is the array \( H \) such that

\[
H[i, j] = |\{(x, y) | i = [x \cos \theta_j + y \sin \theta_j], \theta_j = \frac{\pi}{p}(j + 1) \text{ and } I[x, y] = 1\}|
\]

(1)

\( j \) takes on the integer values \( 0, 1, \ldots, p - 1 \). These correspond to the \( p \) angles \( \theta_j = \frac{\pi}{p}(j + 1), 0 \leq j < p \). Hence \( 0 < \theta_j \leq \pi \). For \( \theta_j \) in this range and \( x \) and \( y \) in the range \( 0 \ldots N - 1 \), \( [x \cos \theta_j + y \sin \theta_j] \) is in the range \( -\sqrt{2}N \ldots \sqrt{2}N \). Hence \( H \) is at most a \( 2\sqrt{2}N \times p \) matrix. In the following, parallelization for hough transform for distributed and shared memory machines is described. Furthermore, performance results for several implementations are presented.

4.1 Distributed memory

The size of the image and the Hough array is \( O(N^2) \) and \( O(Np) \) respectively. These sizes are comparable for typical values of \( N \) and \( p \) (\( N \) is typically 1024 or 2048, while \( p \) varies from 45 to 180). We will assume that hough array is large enough, that it needs to be distributed across several processors. Thus On completion, the \( 2\sqrt{2}N \times p \) Hough array \( H \) is distributed over the nodes in blocks of size \( 2\sqrt{2}N \times p/P \).

Let there be \( P \) processors available on the parallel processors. In an integrated vision system, hough transform will typically be performed after edge detection. We assume that that edge detection was performed using uniform (1-D or 2-D) blocks, as it is a data independent local operation. Thus a Thresholding function has already been applied to the pixels and each node has a list of pairs (\( x, y \)) such that \( I[x, y] \) passes the threshold. We call this list the edge list for the node. The size of this edge list will be different for different nodes. For ease of presentation, we assume that the number of hypercube nodes \( P \) divides the number of angles \( p \) as well as the image dimension \( N \).

Hough transform is an example of data dependent global operation. However, the data is not spatially related i.e. the algorithm to compute Hough transform does not depend relationship between different edges. Each edge pixel can potentially contribute to every partition of the hough array. Thus global communication is required. One way of achieving this global communication is by viewing the \( P \) nodes of the hypercube multicomputer as forming a ring. For any node \( i \), let left (\( i \)) and right (\( i \)), respectively, be the node counterclockwise and clockwise from node \( i \). Let logical (\( i \)) be the logical index of node \( i \) in the ring.
procedure UpdateH partition \((H)\)
  for each \((x, y)\) in edge list do
    for \((j := j_{\text{Begin}}\) to \(j_{\text{Begin}}\) + \(\text{size} - 1\) do
      \(\theta = \frac{x}{y}(j + 1)\)
      \(i = x \cos \theta + y \sin \theta\)
      increment \(H[i, \theta]\) by 1
    end;
  end;
end; \{cf UpdateH partition\}
\(\ell := \) logical index of this node, \(\text{size} := \frac{p}{P}\);
\(j_{\text{Begin}} := \text{size} \times \ell\)
initialize own \(H\) partition to zero;
for \(i := 0\) to \(P - 1\) do
  UpdateH partition;
  send own \(H\) partition to node on right;
  receive \(H\) partition from node on left;
  \(j_{\text{Begin}} := (j_{\text{Begin}} - \text{size}) \mod p;\)
end;

Figure 3: Algorithm to compute \(H\)

This algorithm (Figure 3) is run on each node. As remarked earlier, each node has an edge list and an \(H\) partition. The \(H\) partitions move along the ring one node at a time. When an \(H\) partition reaches any node, the edge list of that node is used to update it, accounting for all contributions these edges make to this \(H\) partition. Procedure UpdateH Partition does precisely this. \(j_{\text{Begin}}\) is the \(j\) value corresponding to the first angle (column) in the \(H\) partition currently in the node. \(\text{size} = \frac{p}{P}\) is the number of columns in an \(H\) partition. During the compute phase, an \(H\) partition is updated.

The size of the edge list in each node is different and this difference significantly impacts the performance of the algorithm. The node with the maximum number of edges may thus become a bottleneck. To reduce the run time, one may attempt to obtain an equal or near equal distribution of the edges over the \(P\) nodes. The load is defined as the size of the nodes' edge list. A heuristic to balance the load is given in Figure 4. Load balancing is accomplished by averaging over the load in processors that are directly connected. This heuristic can be used because the computation of hough transform does not utilize any spatial relationship between edges. The variables used have the following significance:

\[
\begin{align*}
\text{MyLoad} &= \text{current load in the node processor} \\
\text{HisLoad} &= \text{load in a directly connected node processor} \\
\text{MyLoadSize} &= \text{size of the load in the node processor} \\
\text{HisLoadSize} &= \text{size of the load in a directly connected node processor} \\
\text{avg} &= \text{average size of the load of the two processors}
\end{align*}
\]

The hough transform algorithm, as well as the load balancing heuristic were programmed in C and run on an NCUBE/7 hypercube with 64 nodes [15]. The percentage of pixels in an \(N \times N\) image that passed the threshold was fixed at 5\%, 10\%, or 20\%. The number of edge pixels in each nodes partition was determined using a truncated normal distribution with variance being one of 4\%, 10\%, and 64\% of the mean. In all cases, we arbitrarily set \(p = 180\).
procedure LoadBalance();
for i := 0 to CubeSize do
    Send MyLoadSize to neighbor processor along dimension i;
    Receive HisLoadSize from neighbor processor along
    dimension i;
    avg=(MyLoadSize+HisLoadSize+1)/2;
    if [MyLoadSize > Avg][
        Send extra load (MyLoadSize-Avg) to neighbor
        processor along dimension i;
        MyLoadSize = Avg; ]
    else if [HisLoadSize>Avg][
        Receive extra load (Avg-HisLoadSize) from neighbor
        processor along dimension i
        MyLoadSize+ =HisLoadSize-Avg;]
end;

Figure 4: Load balancing heuristic

Preliminary experiments suggested that the time to load balance is less than 2% of the overall run time (load balance followed by Hough transform computation). The run time of the algorithm, both with and without load balancing, is given in Table 4 for the cases of \(P = 64\). We see that as the load variance increases from 4% to 64%, the run time of the algorithm without load balancing increases significantly. In fact, it almost doubles. With load balancing, however, the run time is quite stable. Furthermore, it is always less than the run time for 4% variance without load balancing. When the variance in load is 64%, load balancing results in a 25% to 53% reduction in run time!

4.2 Shared memory

As discussed earlier, the hough transform is a global data dependent algorithm. Therefore, equal distribution of significant features among processors is important for load balancing. We implemented hough transform on encore multimax (a shared memory machine) using static edge partitioning (SEP) to balance the load (as detected by algorithm characteristics). Table 5 shows performance of hough transform using SEP for image size of 256 with edge concentration of 5%, 10% and 20%. “SEP CG16” indicates a coarse-grain hough array update algorithm, “SEP LA” indicates using arrays of locks to update the hough array and “PP” implies using parameter partitioning as the computation decomposition scheme. It is clear that most of the time, a coarse-grain update strategy performs better than the others. Note that all these schemes exploited the information that hough transform is global data dependent algorithm for initial computation partitioning. Dynamic load balancing schemes were also used for partitioning computations but the results are omitted due to space limitations.

5 SUMMARY AND CONCLUSIONS

Although several projects are underway to develop new architectures for computer vision, tools to effectively use those systems or commercially available general purpose architectures are limited or non-existent. Unless we can develop efficient methods for mapping vision algorithms and programs on these architectures, the performance gains will be limited. Hence, there is a need to develop an environment and tools that assist users to design, implement and evaluate their applications on multiprocessor architectures.
<table>
<thead>
<tr>
<th>Image Size</th>
<th>%</th>
<th>No Load Balancing</th>
<th>Load Balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4%</td>
<td>10%</td>
<td>64%</td>
</tr>
<tr>
<td>128 x 128</td>
<td>5</td>
<td>0.3462</td>
<td>0.4200</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.6512</td>
<td>0.7735</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>1.2052</td>
<td>1.5239</td>
</tr>
<tr>
<td>256 x 256</td>
<td>5</td>
<td>1.2038</td>
<td>1.5377</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2.4770</td>
<td>2.9324</td>
</tr>
<tr>
<td>512 x 512</td>
<td>5</td>
<td>4.9077</td>
<td>5.8004</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>19.3672</td>
<td>23.9020</td>
</tr>
</tbody>
</table>

Number of nodes = 64

Table 4: Comparisons of the algorithm with and without load balancing

<table>
<thead>
<tr>
<th>P</th>
<th>5% Edge</th>
<th>10% Edge</th>
<th>20% Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SEP CG16</td>
<td>SEP LA</td>
<td>PP</td>
</tr>
<tr>
<td>1</td>
<td>218.48</td>
<td>218.48</td>
<td>218.48</td>
</tr>
<tr>
<td>2</td>
<td>108.11</td>
<td>107.35</td>
<td>111.31</td>
</tr>
<tr>
<td>4</td>
<td>55.18</td>
<td>55.18</td>
<td>55.18</td>
</tr>
<tr>
<td>8</td>
<td>28.23</td>
<td>32.45</td>
<td>29.24</td>
</tr>
<tr>
<td>12</td>
<td>19.5</td>
<td>20.68</td>
<td>20.52</td>
</tr>
<tr>
<td>16</td>
<td>15.39</td>
<td>25.47</td>
<td>18.52</td>
</tr>
</tbody>
</table>

Table 5: Computation times for static edge partitioning for different edge concentrations

In this paper we presented a design for a software development environment (SDE) for implementing vision systems applications on multiprocessors. The SDE design exploits characteristics of vision systems, and uses a classification scheme for vision algorithms to develop a parallelization and performance evaluation tool. These tools use databases that store knowledge of parallelization for different known computations on common architectures. Some parts of SDE are currently operational and others still need to be developed.

6 ACKNOWLEDGEMENTS

Alok Choudhary would like to acknowledge the support of the Engineering Foundation Grant No. RJ-B-90-23 for this work.
References


