Parallelizing Image Feature Extraction on Coarse-grain Machines *

Yongwha Chung† and Viktor K. Prasanna

Department of EE-Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562
http://ceng.usc.edu/~prasanna

Abstract

In this paper, we present a fast parallel algorithm for feature extraction on coarse-grain MIMD machines. By maintaining algorithmic threads at each node, our algorithm enhances processor utilization and obtains large speed-ups. Our implementations show that, given an 1024×1024 *image, speed-ups of 27.6 and 56.0 on a 32-node SP2 and a 64-node T3D can be achieved.*

*This research was supported in part by NSF under grant CCR-9317301 and in part by DARPA under grant F49620-95-1-0522. A preliminary version of this paper appears at the IEEE International Conference on Computer Architectures for Machine Perception (CAMP), September 1995.

†Supported by a scholarship from ETRI.

1 Introduction

Over the years, low-level vision problems have been parallelized on fine-grain SIMD (Single Instruction Stream Multiple Data Streams) machines. These problems are characterized by regular and local computations. Such regular vision problems can be easily parallelized on fine-grain SIMD machines since the work load can be statically determined at compile time. In contrast, the time performance of most vision algorithms depends on the input image. The variation in the input image produces unbalanced workload and irregular data dependencies. This makes it highly inefficient to use SIMD machines for parallelizing such vision problems.

Coarse-grain MIMD (Multiple Instruction Streams Multiple Data Streams) machines employing commercial "off-the-shelf" Reduced Instruction Set Computer (RISC) processors are currently available. The typical number of nodes in these machines is in the range of tens to a few hundred nodes. These machines can process large images due to the powerful RISC processor and large available memory. Moreover, the Message Passing Interface (MPI) standard allows codes developed for these machines to be portable across multiple platforms. Due to these factors, such machines have been successfully deployed in a variety of applications [7]. Several groups have been developing efficient parallel algorithms for irregular vision problems on these machines [2, 4, 5, 6, 8, 14].

In this paper, we develop a fast parallel algorithm for feature extraction. Image feature extraction is a fundamental task in vision systems. In general, algorithms for feature extraction consists of two major subtasks: *contour-pixels detection* and *linear approximation* [13]. Contour-pixels detection involves detection of edges using a convolution operation, removal of false edges using a thinning operation, and connecting similarly oriented edges using a linking operation. Upon detection of contour-pixels, a linear approximation is performed to trace the detected contour-pixels and approximate each contour into line segments.

Design of parallel algorithms for extracting image features is challenging since the contour-pixels detection and the linear approximation tasks exhibit different types of parallelism. The contour-pixels detection is a computation intensive, low-level vision process. Data dependencies in this task are regular and computations are local to each pixel. After exchanging boundary information, each node can execute its operations independently of other nodes. Therefore, contour-pixels detection can be easily parallelized on coarse-grain machines. The regularity of the computational characteristics also allow the communication overhead to be reduced by overlapping the communication with the computation.

Linear approximation can be viewed as a bridge between low and intermediate level vision processes. It converts the iconic pixel data to a list of symbolic data structures. The computational characteristics of this task depends on the input image. The workload on each node can vary and data dependencies are irregular. Because of these irregular characteristics, it is nontrivial to obtain large speed-ups in parallelizing it.

Our parallel algorithm reduces the idle times caused by barrier synchronizations by allowing each node to run asynchronously without violating any data dependency. By computing the number of inter-processor data dependencies caused by non-local approximation tasks, each node terminates itself after completing its tasks. Such local termination does not affect the computational activities in other nodes. Furthermore, to enhance the utilization of the nodes, the algorithm overlaps communication with computation using a dynamic, priority-based scheduling. After specifying the computation into two algorithmic threads according to the inter-processor data dependencies, our algorithm switches between them dynamically based on the arrival of messages from other nodes. This algorithmic approach, can be viewed as a *message-driven* technique [1, 16]. It also provides a sound basis to develop parallel algorithms for irregular vision problems.

Our implementations show that, given an 1024×1024 image, features can be extracted in 1.172 seconds on a 32-node SP2 and in 0.423 seconds on a 64-node T3D. A serial implementation takes 32.350 seconds and 23.674 seconds on a single node of SP2 and T3D, respectively. The parallel code was specified using the MPI library and can be easily ported to other coarse-grain machines.

The organization of this paper is as follows. The computational steps in image feature extraction are outlined in Section 2. Section 3 describes several techniques for parallelizing the feature extraction problem. The details of our implementations and our experimental results are shown in Section 4. Finally, concluding remarks are made in Section 5.

2 Image Feature Extraction

In the past, several approaches have been proposed to extract image features [3, 13, 15]. The major difference in these approaches is the convolution masks used for edge detection and the method used to approximate the contours by linear segments. To illustrate our ideas, we use the Nevatia-Babu [13] *contour-pixels detection* algorithm and the *linear approximation* algorithm proposed by Roberge [15]. However, the techniques developed in this paper can be adapted to parallelize other approaches used for extracting image features. In the following,

for the sake of completeness, we briefly describe these algorithms [13, 15].

The input to the contour-pixels detection is a 2-D image array of pixels (grey levels). The output is a same sized array with directed contour-pixels embedded in the array. The main processing steps are [13]:

1. *Edge Detection*: convolve the input image with masks corresponding to ideal step edges in a selected number of directions.

2. *Thinning and Thresholding*: compare each edge with its neighbors (in the direction orthogonal to the edge's orientation) and retain the edge if it is of greater magnitude than its neighbors and the magnitude is also greater than a fixed threshold.

3. *Edge Linking*: compare each edge with its neighbors in the direction of the edge's orientation and form a link to the neighbors if they are of similar orientation.

The above steps are *window operations*. The output at a pixel depends on the value of the input pixel and its neighboring pixels. The neighborhood of a pixel is defined by a given window size.

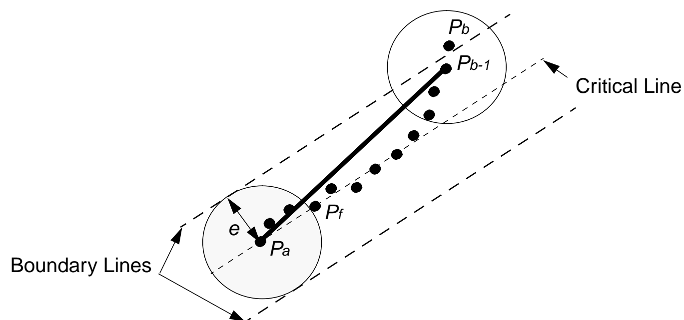


Figure 1: Strip-based linear approximation

Linear approximation is a data reduction technique to extract line segments from a contour-pixel array. The input is a 2-D contour-pixel array and the output is a set of line segments approximating the contours. It consists of contour tracing and approximation operations. Several heuristics are available for approximating a contour by a set of piecewise line segments. In this paper, we have employed a *strip-based* algorithm [15]. However, our framework can be used to implement the approximation step using other heuristics in the

literature. In the following, we briefly describe the strip-based algorithm. Details can be found in [15].

Given a starting pixel P_a of a contour and an error bound e for controlling the quality of the approximation, the algorithm selects a pixel P_f on the contour which is at a distance $> e$ from P_a. If it can not find such a pixel, the algorithm stops and forms a line from P_a to the last pixel of the contour. Otherwise, it draws a line using P_a and P_f. This line is referred to as the *critical line*. Next it forms two lines parallel to the critical line at a distance e. These lines are referred to as the *boundary lines* (see Figure 1). Beginning with pixel P_{f+1}, it successively examines the remaining pixels on the contour until a pixel P_b is found, where P_b is the first pixel lying outside the region formed by the boundary lines or P_b is the last pixel on the contour. If P_b is outside the region formed by the boundary lines, then the line segment $\overline{P_a P_{b-1}}$ is the approximation to the contour from P_a to P_{b-1} and the same procedure is applied starting at P_{b-1}. If P_b is the last pixel, then $\overline{P_a P_b}$ is the approximation to the contour from P_a to P_b and the procedure stops.

Our parallel algorithm as well as its implementation can be easily modified to suit other linear time heuristics for linear approximation leading to similar time performance. The key feature of the heuristic exploited in the parallel algorithm is a scan of the contour pixels starting at one of the end-points of the contour.

Note that, given an $n \times n$ image and a fixed number of windows of size $w \times w$ to be applied at each pixel, extraction of image features (including contour-pixels detection and linear approximation) can be performed in $O(n^2 w^2)$ time on a serial machine. This assumes that the linear approximation is performed by traversing the contours using a linear time heuristic.

3 Parallel Algorithms

Let P and $w \times w$ denote the number of nodes and the window size, respectively. In this section, we present parallel algorithms for the contour-pixels detection and the linear approximation tasks described in Section 2.

3.1 Contour-pixels Detection

Contour-pixels detection consists of three main steps: edge detection, thinning and thresholding, and edge linking. Contour-pixels detection can be parallelized by partitioning the

$n \times n$ image array into P blocks of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ and performing the window operations corresponding to a partitioned block in a node. However, at the beginning of this step, some communication is required to exchange the boundary data to perform the window operation. We call this *boundary padding*. Since each window operation can be performed independently of each other (without additional communication) after boundary padding, we can exploit data parallelism naturally.

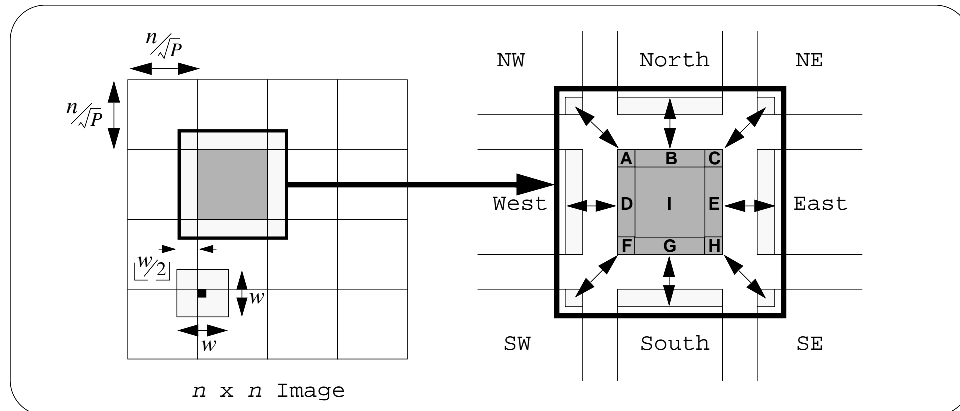


Figure 2: Data exchange between nodes for a boundary padding operation

Note that thinning and linking steps contain "if-then-else" statements. The time needed to execute these steps on a parallel machine depends on the input image. On SIMD machines, execution of the if-then-else statement partitions the nodes into two groups. One group remains idle until the other group completes its execution. Therefore, synchronizing all the nodes at every instruction can result in severe loss of performance. On the contrary, by using the SPMD (Single Program Multiple Data Streams) programming style, the execution time of a node depends on the total number of contour-pixels allocated to it.

The communication overhead caused by the boundary padding can also be reduced by reorganizing the computations to overlap the communication with the computation. This is done by performing the window operations for the boundary pixels of a block (regions A, B, C, D, E, F, G, and H in Figure 2) first, and then sending the results to the neighbors of the node using a non-blocking command. Therefore, the communication for boundary padding can be overlapped with the computations for the interior pixels of a window (region I in Figure 2).

3.2 Linear Approximation

We assume that the input to the linear approximation problem is the output of the contour-pixels detection. The contour-pixel data are stored in a 2-D contour pixel array. In linear approximation, there are two types of contours to be handled: *local* and *global*. Local contour is a contour whose pixels are located in a single node, while global contour is a contour whose pixels belong to more than one node. The local contours can be processed independently of each other as there is no data dependency between the nodes. However, in the case of global contours, the processing can start only after the neighboring node (that contains the contour pixels) completes the approximation on the pixels ahead of the local starting pixel (see Figure 3).

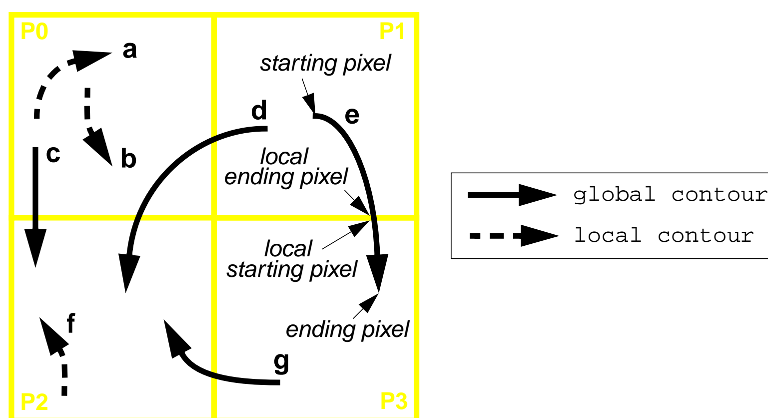


Figure 3: An example illustrating the notation using 4 nodes

Following terminology is used in this section:

- *task*: the computational work to be performed on a local contour or a segment of a global contour located in a node.
- *token*: the data sent from a node to another node to activate the processing of the next segment of a global contour. This data contains the necessary information to continue global contour processing.
- *ready queue*: a queue of tasks for which their tokens from their previous segments have been received.
- *wait queue*: a queue of tasks that are waiting for their tokens to arrive.

```
Procedure: A Loosely Synchronous Algorithm for Linear Approximation
Step 1: Create wait queue.
Step 2: Update wait queue and ready queue.
Step 3: Take a task from ready queue and perform linear approximation.
        Repeat this Step until the ready queue becomes empty.
Step 4: Participate in checking for termination globally. If TRUE, terminate.
Step 5: Send tokens and go to Step 2.
```

Figure 4: Outline of the loosely synchronous algorithm. The code is executed in each node.

To achieve large speed-ups in parallelizing linear approximation, the inter-processor dependencies caused by global contours must be carefully handled. Each node does not know the global information about contours. For example, in Figure 3, node $P0$ can start linearizing local contours a, b, and the first segment of global contour c. However, linearizing the segment of global contour d that belongs to $P0$ can only be initiated after receiving a *token* from node $P1$. The time at which this initiation occurs depends on the computational activities in $P1$. Furthermore, at the beginning of the computation, $P0$ does not know that the segment of contour d that is within $P0$ is the second segment of that contour.

A possible solution is to use a *loosely synchronous* technique. The algorithm using the technique is specified as a sequence of "compute and communicate" phases [9]. In this approach, each node performs operations on its local data, and then checks for a termination condition. If the condition is not satisfied, then all the nodes exchange data and proceed to the next iteration. For the sake of comparison, we outline a loosely synchronous algorithm for linear approximation in Figure 4.

The main disadvantage of this algorithm is that all the nodes synchronize to check for termination (Step 4 in Figure 4) before starting the next iteration. If a node completes Step 3 earlier, then it will be idle until all the other nodes complete Step 3. Since the per-iteration workload in each node depends on the input image data as well as on the computations performed by other nodes, this synchronization is a major source of overhead.

Data re-mapping algorithms [10, 12] can be used to balance the workload on the nodes. In these methods, the contours are redistributed initially to balance the workload of the nodes. However, the cost of calculating the workload of the nodes and the cost of data re-mapping make these algorithms attractive only if the computational cost associated with processing the contour data is high. In linear approximation, the amount of computation

```
Procedure: An Asynchronous Algorithm for Linear Approximation
Step 1: Count the number of sends and receives. Post non-blocking receives.
Step 2: Create wait queue.
Step 3: Check for new tokens. Check wait queue.
        If no token for global contours, go to Step 5.
Step 4: Take a task for global contour from ready queue and perform approximation.
        If needed, send token. Go to Step 6.
Step 5: Take a task for local contour from ready queue and perform approximation.
Step 6: Check termination condition locally. If TRUE, terminate. Else, go to Step 3.
```

Figure 5: Outline of the asynchronous algorithm. The code is executed in each node.

to be performed on each segment is very small. For instance, on a single node of SP2, the execution time to approximate a contour in a typical image is less than 20 $\mu secs$/pixel.

To obtain large speed-ups, we propose an *asynchronous* algorithm where the nodes iterate without synchronizing between iterations. Approximating a local contour is initiated by the arrival of tokens. To reduce the overhead in detecting the global termination status, each node computes the number of outgoing and incoming global contours. As we mentioned, all the starting pixels and the local-starting pixels are identified initially and stored (as tasks) in a wait queue. Therefore, each node can terminate itself when its wait queue becomes empty. Because this *local* termination does not affect the computational activities in other nodes, the asynchronous algorithm does not need barrier synchronization.

In order to further improve the execution time, the local contour processing and the global contour processing in each node are interleaved. The processing in each node can be divided into two categories. Processing of local contours and segments of the global contours having the starting pixel in the node can be performed independently of other nodes. However, processing a segment of a global contour whose starting pixel is outside the node can only be performed after approximating all the previous segments of that contour. A *priority-based scheduling* heuristic is used. The tasks for contour-pixels in each node are grouped into two priority classes: (1) *Priority Class 1*: tasks for global contours (2) *Priority Class 2*: tasks for local contours. Tasks in Priority Class 1 have higher priority compared with the tasks in Priority Class 2.

The above task scheduling can be regarded as an emulation of multi-threading at the algorithm level. In our algorithm, each node performs global contour tasks first, since the result of global contour processing is needed in the subsequent nodes. The arrival of new

tokens at a node depends on the input image. Therefore, by maintaining two *threads*, whenever tokens for the global contour *thread* are exhausted, instead of idling, the node switches to the local contour *thread*. Once a new token arrives, the node switches to the global contour *thread*. Note that processing of local contour *threads* voluntarily yields the CPU after approximating a fixed number of local contours to check for the arrival of new tokens. Details are shown in Figure 5. Even though the polling to check for tokens (Step 3 in Figure 5) is an additional overhead in this algorithm, it is negligible compared with the synchronization overhead for global termination checking in the loosely synchronous algorithm. Note that, this overhead does not depend on the number of nodes in the system.

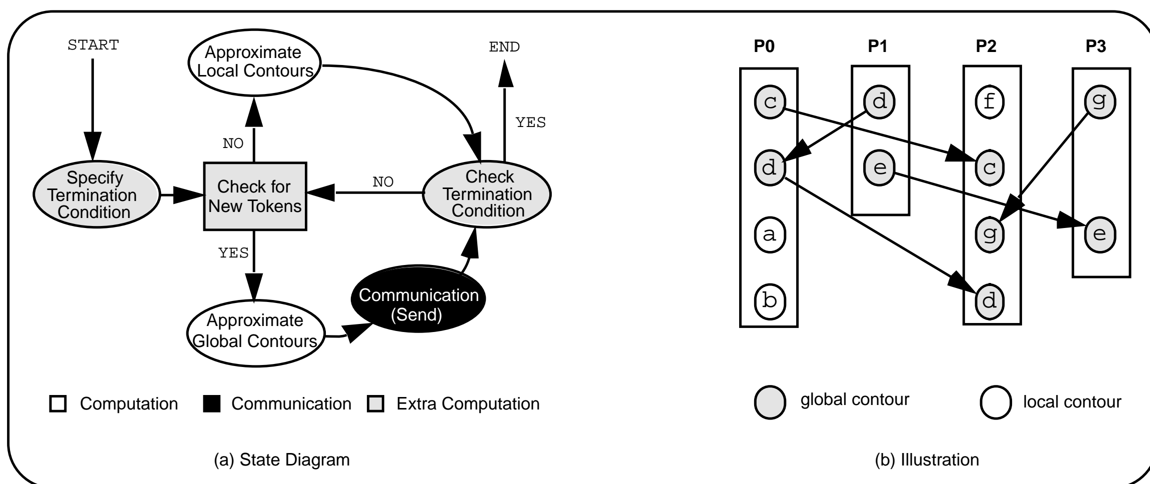


Figure 6: State diagram and an illustration of the asynchronous algorithm

The main steps of this algorithm are represented by five major states (see Figure 6 (a)). Note that the *Communication* state refers to sending new tokens only. There is no explicit state for receiving tokens. An illustration of the execution of the asynchronous algorithm for the example in Figure 3 is shown in Figure 6 (b).

4 Implementation Details and Experimental Results

Our algorithms were implemented on the SP2 (employing Wide nodes) at the Maui High Performance Computing Center (MHPCC) and on the T3D at the Pittsburgh Supercomputing Center (PSC). We used 1, 4, 8, 16, 32, and 64 nodes in the dedicated mode. On the SP2 at MHPCC, the maximum number of Wide nodes available in the dedicated mode was 32. A

512×512 Modelboard image, a 512×512 Landsat image, a 1024×1024 Modelboard image, and a 1024×1024 Landsat image were used in our experiments. Due to space limitations, the raw images and the extracted line segments are not shown.

The code was written using C and the MPI library. In our implementations, parallel I/O was not employed. In all the reported times, the initial I/O time for loading the image data is not considered. This convention is consistent with the previous implementations reported in [2, 4, 8]. Parallelization of the subsequent task (i.e., *Perceptual Grouping* in the building detection system [11]) using the image features distributed among the nodes is discussed in [6].

The speed-ups achieved for the contour-pixels detection are shown in Figure 7. Given a Modelboard image of size 1024×1024, contour-pixels can be detected in 1.047 seconds on a 32-node SP2 and in 0.344 seconds on a 64-node T3D. A serial implementation takes 29.490 seconds and 20.990 seconds on a single node of SP2 and T3D, respectively.

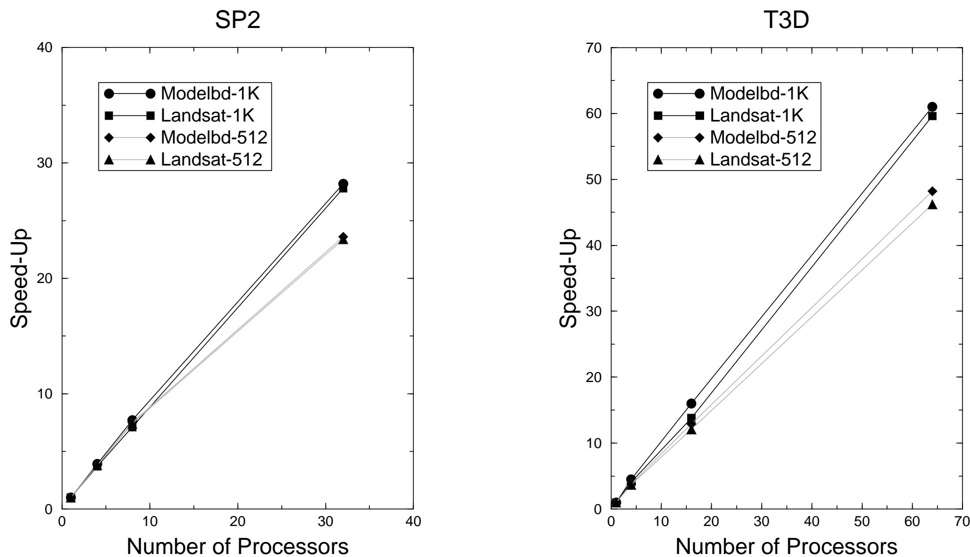
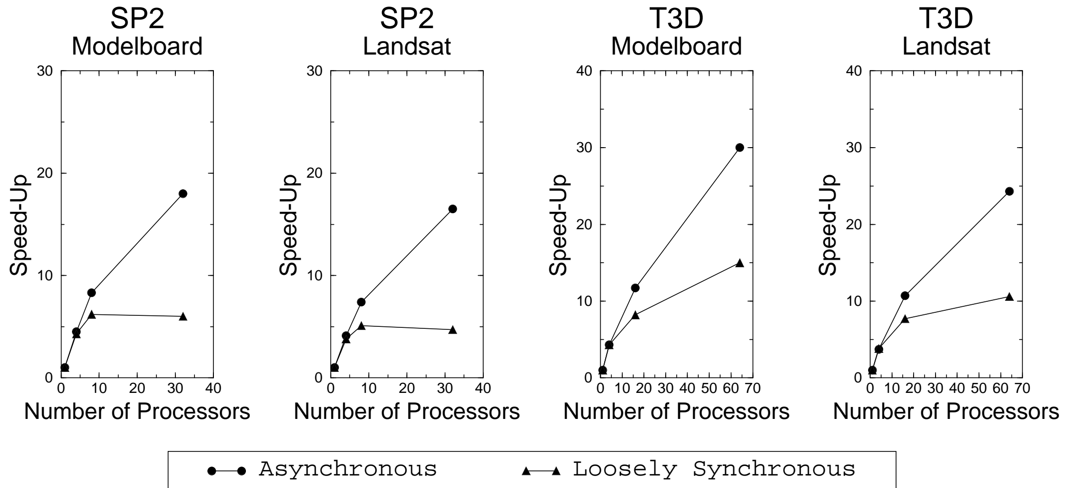
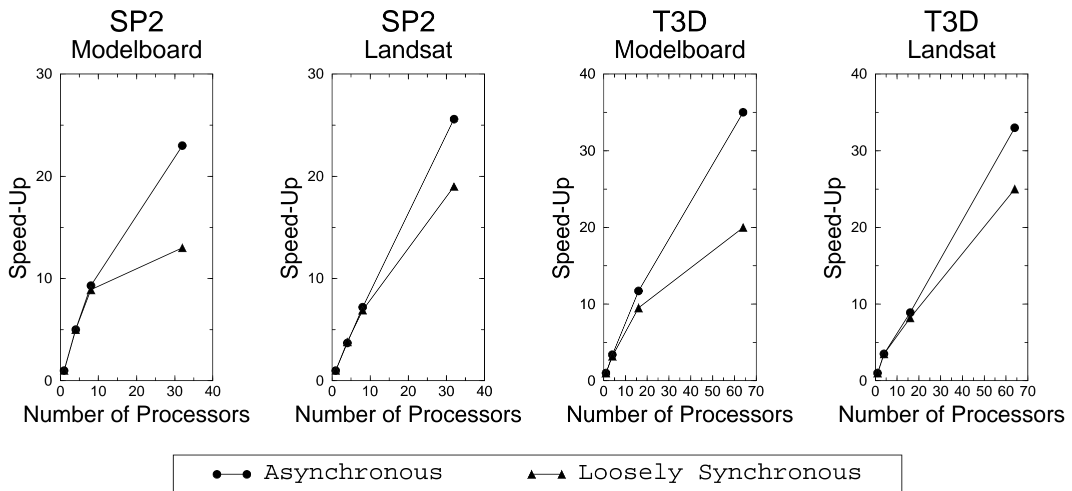


Figure 7: Observed speed-ups for the contour-pixels detection step

For the linear approximation step, we also implemented the *loosely synchronous* algorithm discussed in Section 3 to evaluate the effectiveness of our algorithm. The observed speed-ups of the loosely synchronous algorithm and the asynchronous algorithm are shown in Figure 8. The speed-ups of the loosely synchronous algorithm become saturated soon due to the idling of the nodes. On a Modelboard image of size 1024×1024, our asynchronous algorithm completes in 0.125 seconds on a 32-node SP2 and in 0.079 seconds on a 64-node T3D. For



(a) Speed-ups for 512 x 512 images



(b) Speed-ups for 1K x 1K images

Figure 8: Observed speed-ups for the linear approximation step

the same image, the execution times of the loosely synchronous algorithm is 0.212 seconds on a 32-node SP2 and 0.124 seconds on a 64-node T3D. A serial implementation of the same linear approximation problem takes 2.860 seconds and 2.684 seconds on a single node of SP2 and T3D, respectively.

Although we have shown the experimental results on the SP2 and the T3D, our code written using the MPI library can be easily ported to other parallel machines (such as Intel Paragon and Meiko CS2, among others).

5 Conclusion

We have presented a fast parallel algorithm for image feature extraction and shown implementations of it on the SP2 and the T3D. Our asynchronous algorithm enhances node utilization. Furthermore, we developed a multi-threading technique at an algorithmic level to overlap communication with computation.

The experimental results are encouraging. For example, for a 1024×1024 image, we obtained speed-ups of 27.6 and 56.0 using our algorithm on a 32-node SP2 and a 64-node T3D, respectively. Our code using the MPI standard library can be easily ported to other parallel machines.

We are currently extending the message-based task scheduling technique to parallelize symbolic grouping operations [6]. Intermediate and high-level vision problems are characterized by uneven distribution of symbolic features among the nodes and irregular interprocessor data dependencies caused by the input image. In parallelizing these problems, the overhead caused by synchronization increases with the number of nodes. We believe that a sound basis to develop parallel algorithms for irregular vision problems is to let each node run asynchronously and perform task scheduling based on message arrival.

Acknowledgments

We would like to thank Dr. David A. Bader at the University of New Mexico for his assistance in using the Landsat images. We also thank Dr. Cho-Li Wang at the University of Hong Kong for helpful discussions.

References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[2] D. Bader and J. Já Já, "Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study," *Technical Report, University of Maryland*, 1994.

[3] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 8, No. 6, 1986.

[4] A. Choudhary and R. Thakur, "Connected Component Labeling on Coarse Grain Parallel Computers: An Experimental Study," *Journal of Parallel and Distributed Computing,* Vol. 20, pp. 78-83, 1994.

[5] Y. Chung, V. Prasanna, and C. Wang, "A Fast Asynchronous Algorithm for Linear Feature Extraction on IBM SP-2," *Proc. of Conference on Computer Architecture for Machine Perception*, pp. 294-301, 1995.

[6] Y. Chung, C. Wang, and V. Prasanna, "Parallel Algorithms for Perceptual Grouping on Distributed-Memory Machine," Submitted to *Journal of Parallel and Distributed Computing,* 1997.

[7] I. Foster, "Designing and building parallel programs," Addison Wesley, 1995, http://www.mcs.anl.gov/dbpp.

[8] N. Copty, S. Ranka, G. Fox, and R. Shankar, "A Data Parallel Algorithm for Solving the Region Growing Problem on the Connection Machine," *Journal of Parallel and Distributed Computing,* Vol. 21, pp. 160-167, 1994.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Slamon, and D. Walker, *Solving Problems on Concurrent Processors: General Techniques and Regular Problems*, Prentice-Hall, Inc., 1988.

[10] D. Gerogiannis and S. Orphanoudakis, "Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 9, pp. 994-1013, 1993.

[11] A. Huertas, C. Lin, and R. Nevatia, "Detection of Buildings from Monocular Views of Aerial Scenes using Perceptual Grouping and Shadows," *Proc. of the Image Understanding Workshop*, pp. 253-260, 1993.

[12] C. Lin, V. Prasanna, and Y. Chung, "Data Remapping for Intermediate Level Analysis in Image Understanding on Distributed-Memory Machines," *Proc. of Workshop on Solving Irregular Problems on Distributed Memory Machines, IPPS'95*, pp. 35-42, 1995.

[13] R. Nevatia and K. Babu, "Linear Feature Extraction and Description," *Computer Graphics and Image processing*, Vol. 13, pp. 257-269, 1980.

[14] V. Prasanna, *Parallel Architectures and Algorithms for Image Understanding,* Academic Press, 1991.

[15] J. Roberge, "A Data Reduction Algorithm for Planar Curves," *Computer Vision, Graphics, and Image Processing*, Vol. 29, pp. 168-195, 1985.

1994.

[16] T. Yang and A. Gerasoulis, "List Scheduling with and without Communication," *Parallel Computing*, Vol. 19, pp. 1321-1344, 1993.