

Figure 1: An overview of the Linear Feature Extraction.

In this paper, we discuss parallelizing a linear feature extraction problem (see Figure 1). This task extracts line segments from input images. The linear feature extraction algorithm consists of two major phases: *contour-pixels detection* and *linear approximation*. A general approach for contour-pixels detection involves detection of edges using convolution operation, removal of false edges using a thinning operation, and connecting similarly oriented edges using a linking operation. Upon detection of the contour pixels, a linear approximation is performed to trace the detected contour pixels and approximate each contour into line segments.

Parallelizing the contour-pixels detection phase on a coarse-grain machine such as SP-2, can be done easily. After exchanging the boundary information, the operations in a processing node can be done independently of operations in other processing nodes [Prasanna and Wang, 1994]. However, in the linear approximation phase, the run-time data dependencies make the design of a parallel algorithm and its implementation to obtain large speed-ups to be a nontrivial task. The irregularity causes some processing nodes to be idle while others are active. In this paper, we only focus on the parallelization of the linear approximation phase. We develop an asynchronous algorithm by allowing each processing node to run independently of other processing nodes without violating any data dependency. Furthermore, we improve the utilization of processing node by maintaining algorithmic threads in each processing node. A preliminary version of this paper appears in [Chung et al., 1995].

Our implementation shows that, the execution of the

Figure 2: Strip-based linear 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.

3 Characteristics of Distributed Memory Machines

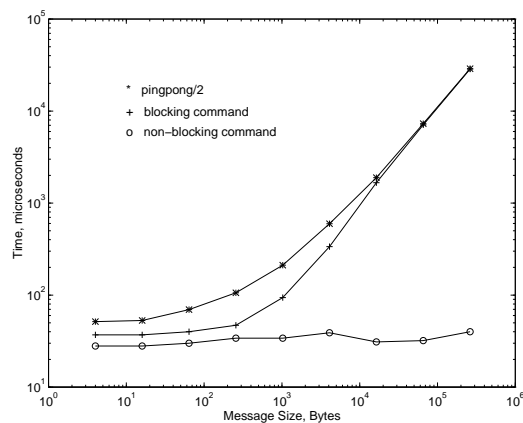


Figure 3: Performance of Point-to-Point Communication Operations on a SP-2

To understand characteristics of current distributed memory machines, we analyze the features of one of the machines, IBM SP-2, in detail. SP-2 is a distributed memory machine employing a network-based computing model. Three types of processing nodes are provided for SP-2: *Wide 66*, *Thin 66*, and *Thin 62*. The POWER2 processor in the *Wide 66* and *Thin 66* processing nodes runs at 66.7 MHz, giving a peak performance of 266 MFLOPS. The POWER processor in

the *Thin 62* processing nodes runs at 62.5 MHz, giving a peak performance of 125 MFLOPS. The processing nodes share data via message passing over the *HPS (High Performance Switch)* multistage interconnection network [Stunkel et al., 1994]. Each 8×8 *switch chip* in a *HPS switch board* has 125 *nsec* channel latency from any input port to any other output port in the absence of contention. The *switch chip* also provides 40 *MBytes/s* peak channel bandwidth and maximum aggregate bandwidth of 320 *MBytes/s* per chip. Due to the low (hardware) latency associated with each stage of the *HPS*, the users may view all the processing nodes as equidistant from each other in the system. The *HPS Adapter-2* switch adapter (interface board between a processing node and the *HPS*) has a separate processor (40 MHz i860 XR) and DMA engines to offload protocol processing from the main processor in the processing node.

Figure 3 shows the performance of point-to-point communication operations. The experiments were conducted at the Maui High Performance Computing Center using the MPL message passing library [IBM, 1994]. We employed 64 *Wide 66* processing nodes to conduct our experiments. In order to reduce other users' interference in the 400-node SP-2 system that was employed to conduct the experiments, each of the 64 processing nodes were set to *dedicated mode*. In the dedicated mode, the code runs in a single user mode in each processing node. However, the interconnection network is shared by other users. All the times reported were measured by employing the *User Space* communication option. Our test code synchronized all the processing nodes before starting the wall clock *gettimeofday*. Then, the time for communication operations were measured. The reported time is the minimum time repeated over 100 executions of the same code. It is reasonable to select the minimum value even in the dedicated-mode because there may be some interference caused by other users' traffic in the network.

As shown in Figure 3, the non-blocking send command in MPL returns to the calling process in constant time (around 30 microseconds). This is because the separate communication processor and DMA engines take care of the user message-buffer management operations. However, the user should be careful to use this user message-buffer again because the command can return to the calling process before the user message-buffer is copied into the system buffer. On the contrary, the blocking send command in MPL returns to the calling process after copying the message-buffer into the system buffer. Thus, the execution time of the blocking send command depends on the message size. Since both these send commands do not guarantee that the message has been delivered to the receiver, we also measured the round-trip communication time by a ping-pong operation. As in [Prasanna and Wang, 1994], the round-trip communication time can be modeled as $T_d + m\tau_d$ time, where T_d denote the startup time for sending a

message and τ_d denote the transmission rate (seconds per bytes) for data communication; $T_d = 46\mu sec$ and $\tau_d = 0.035\mu sec/byte$. Therefore, if the size of message is smaller than 1.3Kbyte, then the startup time is a dominant factor; otherwise, the transmission rate is a dominant one. For T3D, the startup time and transmission rate using MPI were measured as $T_d = 65\mu sec$ and $\tau_d = 0.035\mu sec/byte$ [Bhat, 1995]. Also, we verified that these times were *almost* independent of the physical distance between the sender and the receiver. Note that, as the message size increases, the gap between the time for executing a blocking command and the half ping-pong time reduces.

4 Parallel Algorithms

Let P denote the number of processing nodes. The $n \times n$ image array is divided into P blocks of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$. We assume that the input to the linear approximation phase is the output of the contour-pixels detection. The contour-pixel data are stored in a 2-D contour pixel array. In the linear approximation, there are two types of contour-pixel data to be handled: *local* contour-pixel data and *global* contour-pixel data. Local contour is a contour whose pixels are located in a single processing node, while global contour is a contour whose pixels belong to more than one processing node. The local contours can be processed independently as there is no data dependency between the processing nodes. However, in the case of global contours, the processing can start only after the neighboring processing node completes the approximation on the pixels ahead of the local starting pixel (see Figure 4).

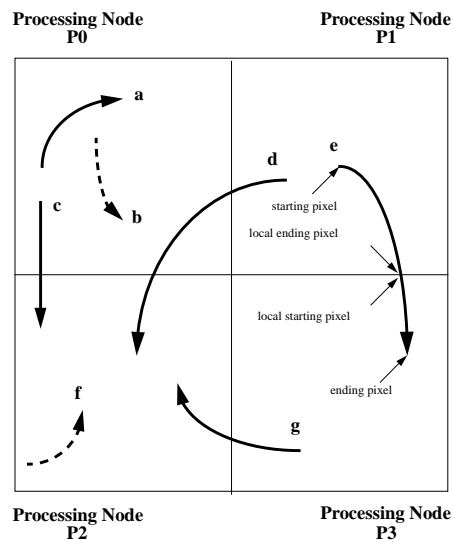


Figure 4: An example illustrating the notations

Following terminology is used in this paper: (1) *task*: the computational work to be performed on a local contour or segment of a global contour located in a processing node, (2) *token*: data sent from a processing node to another processing node to activate the processing of the next segment of a global contour located in the neighboring processing node. This contains information to continue global contour processing, (3) *ready queue*: queue for tasks having a token, (4) *wait queue*: queue for tasks that are waiting for their tokens to arrive.

4.1 A Synchronous Iterative Algorithm

The main difficulty in parallelizing linear approximation is interprocessor dependencies for global contours. Each processing node does not know this global information. For example, processing node P0 can start linearizing local contours a, b, and the first segment of global contour c in Figure 4. However, linearizing the segment of global contour d that belongs to P0 can be initiated after receiving a *token* from processing node P1 and this initiation time depends on the computational activities in the processing node P1. Furthermore, the processing node P0 does not know that its segment of the global contour d is the second segment of that contour.

A possible solution is to use a *synchronous iterative* technique widely used in scientific computations [Lester, 1993]. In this approach, each processing node performs operations on its local data, and then checks for a termination condition. If the condition is not satisfied, then all the processing nodes exchange data and proceed to the next iteration. For the purpose of comparison, we outline the synchronous linear approximation algorithm in Figure 5. This algorithm is executed in each processing node.

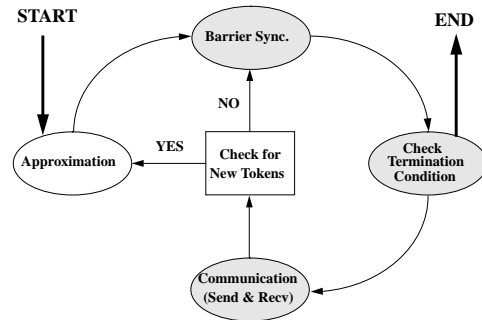
Method 1 : A Synchronous Iterative Algorithm

- 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 is empty.
 - Step 4:** Synchronize all processing nodes.
 - Step 5:** Participate in checking for *global* termination. If TRUE, terminate.
 - Step 6:** Exchange tokens and go to Step 2.
- end

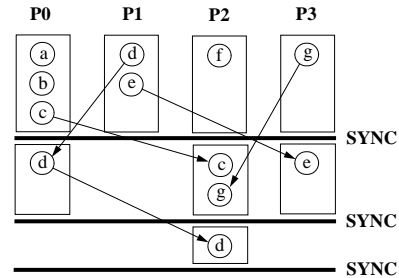
Figure 5: An outline of Synchronous Iterative Algorithm for linear approximation.

In Step 1, we identify the starting and local-starting pixels from the contour pixel array and store them in a wait queue. Initially, the contours having the starting pixel are considered as the tasks having a token. Then, we check the wait queue in Step 2. If any task has a token, it is extracted from the wait queue and inserted into the ready queue. Step 3 performs the linear approximation operations on all tasks in ready queue. Each pro-

cessing node has eight outgoing buffers corresponding to its eight neighboring nodes in 2 dimensions. If the task corresponds to a global contour, then the token is stored in the corresponding outgoing buffer. After completing all the ready tasks, the processing nodes participate in a barrier synchronization in Step 4. In Step 5, each PE checks the status of its wait queue and participates in *global* check for termination. If any of the processing nodes has a non-empty wait queue, then the tokens are exchanged in Step 6 and the algorithm proceeds to the next iteration. The main steps of the algorithm can be represented by four major states (see Figure 6 (a)).



(a) State Diagram of the Synchronous Iterative Algorithm



(b) Illustration of the Synchronous Iterative Algorithm

Figure 6: State Diagram and an illustration of the Synchronous Iterative Algorithm

The main disadvantage of this algorithm is all processing nodes synchronize (Step 4 in Figure 5) before starting the next iteration. If a processing node completes Step 3 earlier, then it will be idle until all the other processing nodes complete Step 3. Because per-iteration work load in each processing node depends on the input image data, this barrier synchronization in each iteration is a possible source of parallel overhead. Furthermore, SP-2 does not provide hardware support for fast barrier synchronization. It takes approximately 100 microseconds to synchronize 2 processing nodes and it increases to 500 microseconds for synchronizing 64 processing nodes, provided that all processing nodes are ready before the MPL barrier synchronization command is executed.

4.2 An Asynchronous Algorithm

In this algorithm, the processing nodes iterate without synchronizing between iterations. Local approximations are initiated by arrival of tokens. To reduce the overhead in detecting the termination status, each node computes the number of outgoing and incoming global contours. Because each processing node can terminate itself and this *local* termination does not affect the computational activities in other processing nodes, the asynchronous algorithm does not need barrier synchronization at all. (In the synchronous iterative algorithm, each processing node should participate in every barrier synchronization to avoid deadlock even though it has completed all its tasks.) This elimination of synchronization overhead can improve the utilization of processing nodes as each processing node can start its next task independently of all the other processing nodes.

In order to further improve the execution time, the local contour processing and the global contour processing in each processing node are interleaved. The processing in each processing node can be divided into two categories. Processing of local contours and segments of the global contours having the starting pixel in the processing node can be done independently of other processing nodes. However, processing of segment of the global contour not having the starting pixel can only be done after approximating the previous segment of that global contour. A *priority-based scheduling* heuristic can contribute to improved performance. The tasks for contour pixels in each processing 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 tasks in Priority Class 2.

Method 2 : An Asynchronous Algorithm

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 the approximation.
If needed, send token. Go to Step 6.

Step 5: Take a task for local contour from ready queue and perform approximation.

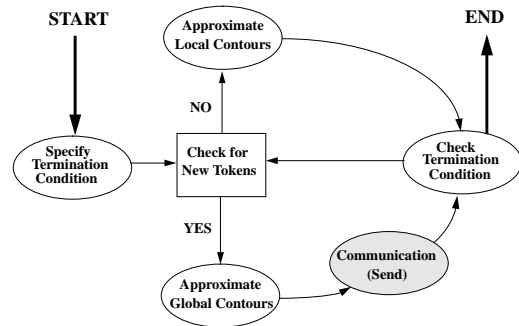
Step 6: Check termination condition *locally*.
If FALSE, go to Step 3.

end

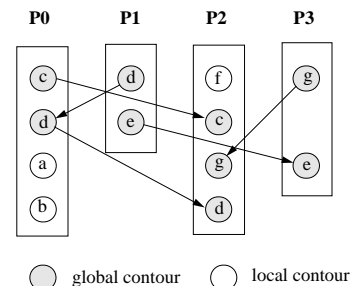
Figure 7: An outline of Asynchronous Algorithm for linear approximation.

The above task scheduling can be regarded as an emulation of *multithreading* [Agarwal, 1992] at an algorithmic level. Early work in multithreading focused on operating systems and shared-memory machines to hide

I/O latency and cache miss latency, respectively. Recently, a *message-driven* technique has been used in designing a parallel compiler [Holm et al., 1994], a parallel programming language [Kale, 1993], a fast communication mechanism [Eicken et al., 1992], as well as in designing parallel algorithms [Crockett and Orloff, 1994, Felten and McNamee, 1992]. In our algorithm, each processing node performs global contour tasks first, since the result of global contour processing is needed in the subsequent processing nodes. The arrival of new tokens depends on input image. Therefore, by maintaining two *threads*, whenever tokens for the global contour *thread* are exhausted, instead of idling, the processing node switches to the local contour *thread*. Once new tokens arrive, each processing node switches to the global contour *thread* again. Note that processing of local contour *thread* voluntarily yields the CPU after approximating a given number of local contours to check for arrival of new tokens, instead of running to completion. Details are shown in Figure 7. Even though the polling to check for tokens is an additional overhead in this algorithm, it is negligible compared with the synchronization overhead in the synchronous iterative algorithm. This overhead can be reduced by using *active messages* [Eicken et al., 1992].



(a) State Diagram of the Asynchronous Algorithm



(b) Illustration of the Asynchronous Algorithm

Figure 8: State Diagram and an illustration the Asynchronous Algorithm

The main steps of this algorithm can be represented by

five major states (see Figure 8 (a)). Note that the *Communication* state is only for sending new tokens. There is no explicit state for receiving tokens. An illustration of the execution of the asynchronous algorithm for the example in Figure 4 is shown in Figure 8 (b).

5 Implementation Details and Experimental Results

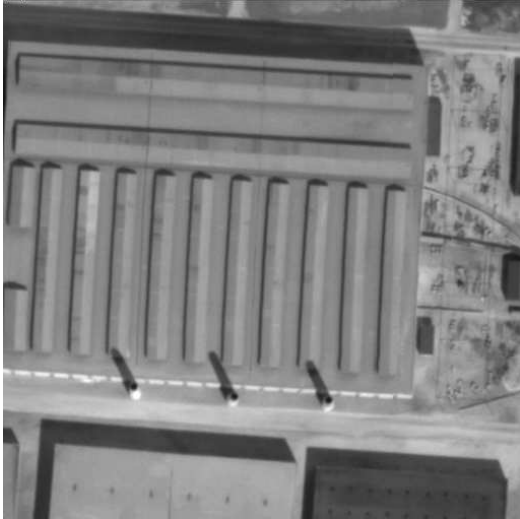


Figure 9: A 512×512 Building Image

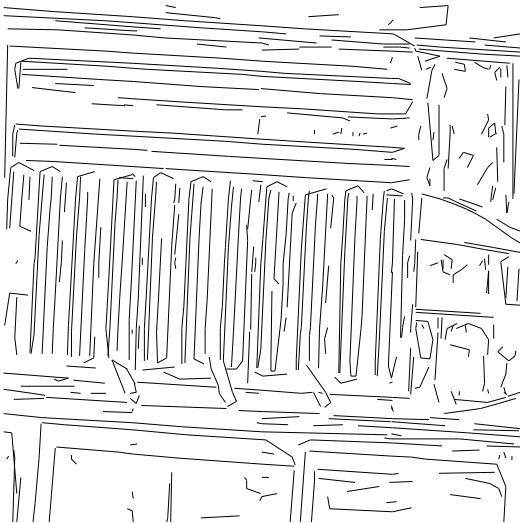


Figure 10: Extracted Line Segments

The algorithm was implemented on a SP-2 at the Maui High Performance Computing Center and a T3D at the Pittsburgh Supercomputing Center. We used 1, 4, 8, 16, 32, and 64 processing nodes in each machine. A 512×512 shopping mall image (Figure 9) was used and the output of our algorithm is shown in Figure 10. The

code was written using C and MPI and MPL message passing libraries. The total length of the code is around 3000 lines. The code has two parts: *manager* part and *worker* part. The manager part reads the image data to processing node 0 and then evenly distributes it to all the processing nodes including processing node 0. The worker part is executed by the processing nodes which perform the contour-pixels detection and linear approximation. In this paper, we only report the experimental results of the linear approximation.

In the linear approximation, we exploited the non-blocking command. The multithreading strategy lets some tasks to be processed after issuing a non-blocking command. In Step 2 of Figure 7, a non-blocking receive command was used. After posting the commands for the global contour thread, the main processor can start the next computational work for the local contour thread while the actual receive operation is performed by a communication processor. Also, the main processor initiates the non-blocking send command in Step 4 of Figure 7. Then, the next computational work of the local thread can overlap with the actual send operation being performed by the communication processor.

The implementation of the message packing technique can lead to problems. The communication pattern is irregular and is not known at compile time. If we determine the degree of the packing using a fixed-size outgoing buffer and require that each processing node sends the packed message when the outgoing buffer is full, then deadlock can occur. We avoid this scenario by sending the outgoing buffer after some fixed amount of time.

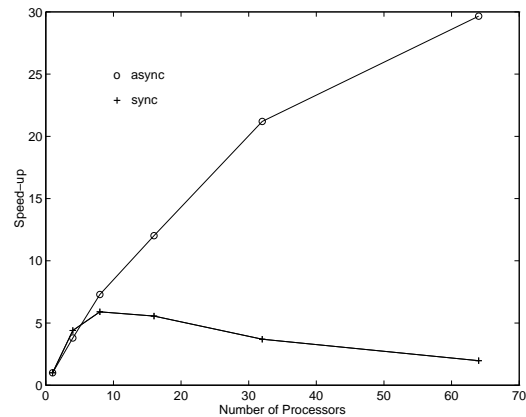


Figure 11: Speed-up of the Linear Approximation for a 512×512 image on a SP-2

The speed-ups of both synchronous and asynchronous algorithms are calculated using the elapsed time. The elapsed time was measured using the wall clock in the dedicated-mode. These speed-ups are shown in Figure 11 and Figure 12. The speed-ups of the synchronous algorithm become saturated soon due to the idling of the processing nodes caused by data dependencies and syn-

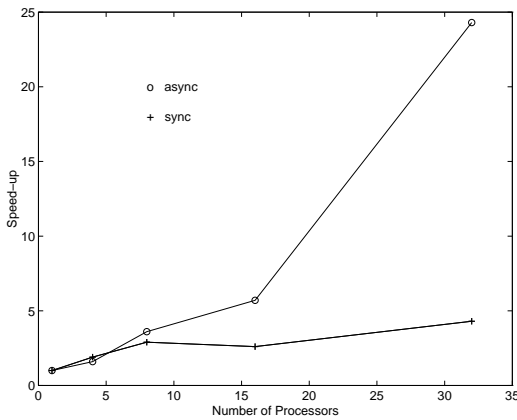


Figure 12: Speed-up of the Linear Approximation for a 512×512 image on a T3D

chronization overheads. The asynchronous implementation on an image of size 512×512 completes in 0.015 seconds on a 64-node SP-2 and 0.032 seconds on a 32-node T3D. The execution time of the synchronous algorithm is 0.225 seconds on a 64-node SP-2 and 0.182 seconds on a 32-node T3D. A serial implementation of the same linear approximation task takes 0.445 seconds on a single-node of SP-2 and 0.779 seconds on a single-node of T3D.

6 Conclusion

We have presented parallel algorithms for linear approximation and shown implementations of them on SP-2 and T3D. Our implementation exploited the features of current distributed memory machines after characterizing the communication performance of the machine. Our asynchronous algorithm enhanced processor utilization by eliminating the barrier synchronizations in the synchronous iterative algorithm. Furthermore, we applied a multithreading technique at an algorithmic level to overlap communication with computation. For implementing the overlapping of computation and communication, we used the non-blocking command to exploit the feature of SP-2 and T3D. In addition, the effect of the relatively large start-up time of the non-blocking command was reduced by packing large number of short messages into a long message. Given a 512×512 image, we obtained a speed-up of 29.7 and 24.3 using our asynchronous algorithm on a 64-node SP-2 and a 32-node T3D, respectively; while the speedup was 1.98 and 4.28 using the synchronous algorithm, respectively. Although we have shown the experimental results on IBM SP-2 and Cray T3D, our code using MPI standard library can be run on other parallel machines (Intel Paragon and Meiko CS-2 among others).

Acknowledgment

The implementations were performed on the SP-2 at the Maui High Performance Computing Center (MHPCC) and the T3D at the Pittsburgh Supercomputing Center. Research at MHPCC is sponsored in part by the Phillips Laboratory, Air Force Material Command, USAF, under cooperative agreement number F29601-93-2-0001. Pittsburgh Supercomputing Center is supported by NSF under Grant CCR-940010P.

References

- [Agarwal, 1992] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, pp. 525-539, 1992.
- [Bhat, 1995] P. Bhat, University of Southern California, *Personal Communication*.
- [Chung et al., 1995] Y. Chung, V. Prasanna, and C. Wang, "A Fast Asynchronous Algorithm for Linear Feature Extraction on IBM SP-2," *Proc. of the Computer Architectures for Machine Perception*, pp. 294-301, 1995.
- [Crockett and Orloff, 1994] T. Crockett and T. Orloff, "Parallel Polygon Rendering for Message Passing Architectures," *IEEE Parallel and Distributed Technology*, pp. 17-28, Summer 1994.
- [Dunham, 1986] J. Dunham, "Optimum Uniform Piecewise Linear Approximation of Planar Curves," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 8, No. 1, pp. 67-75, 1986.
- [Eicken et al., 1992] T. Eicken, D. Culler, S. Goldstein, and K. Schauer, "Active Messages : A Mechanism for Integrated Communication and Computation ," *Proc. of International Symposium on Computer Architecture*, pp. 252-266, 1992.
- [Felten and McNamee, 1992] E. Felten and D. McNamee, "Improving the Performance of Message Passing Applications by Multithreading," *Proc. of the Scalable High Performance Computing Conference*, pp. 84-89, 1992.
- [IBM, 1994] IBM, *Parallel Programming Subroutine Reference Release 2.0*, 1994.
- [Kale, 1993] L. Kale, "Message Driven Execution and Scalability," *Workshop on Analyzing Scalability of Parallel Algorithms and Architectures, IPPS'93*, pp. 16-20, April 1993.

- [Holm et al., 1994] J. Holm, A. Lain, and P. Banerjee, "Compilation of Scientific Programs into Multithreaded and Message Driven Computation," *Proc. of the Scalable High Performance Computing Conference*, pp. 518-525, 1994.
- [Lester, 1993] B. Lester, *The Art of Parallel Programming*, Prentice-Hall, Inc., 1993.
- [Lin et al., 1995] C. Lin, V. Prasanna, and Y. Chung, "Data Remapping for Intermediate Level Analysis in Image Understanding on Distributed-Memory Machines," *Workshop on Solving Irregular Problems on Distributed Memory Machines, IPPS'95*, pp. 35-42, April 1995.
- [Lin and Prasanna, 1995] C. Lin and V. Prasanna, "Analysis of Cost of Performing Communications Using Various Communication Mechanisms," *Symposium on the Frontiers of Massively Parallel Computation*, pp. 290-297, February 1995.
- [MPI, 1994] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, May 1994.
- [Nevatia and Babu, 1980] R. Nevatia and K. Babu, "Linear Feature Extraction and Description," *Computer Graphics and Image processing*, Vol. 13, pp. 257-269, 1980.
- [Prasanna and Wang, 1994] V. Prasanna and C. Wang, "Image Feature Extraction on Connection Machine CM-5," *Image Understanding Workshop*, pp. 595-602, 1994.
- [Roberge, 1985] J. Roberge, "A Data Reduction Algorithm for Planar Curves," *Computer Vision, Graphics, and Image Processing*, Vol. 29, pp. 168-195, 1985.
- [Stunkel et al., 1994] C. Stunkel, D. Shea, B. Abali, M. Atkins, C. Bender, D. Grice, P. Hochschild, D. Joseph, B. Nathanson, R. Swetz, R. Stucke, M. Tsao, and P. Varker, "The SP2 Communication Subsystem," Technical Report, IBM T.J. Watson Research Center, August 1994.
- [Wang et al., 1994] C. Wang, V. Prasanna, H. Kim, and A. Khokhar, "Scalable Data Parallel Implementations of Object Recognition using Geometric Hashing," *Journal of Parallel and Distributed Computing*, pp. 96-109, March 1994.