Parallel and Distributed Algorithms for High Speed Image Processing

Jeffrey M. Squyres* Andrew Lumsdaine* Brian C. McCandless* Robert L. Stevenson†

ABSTRACT

Many image processing tasks exhibit a high degree of data locality and parallelism and map quite readily to specialized massively parallel computing hardware. However, as distributed memory machines are becoming a viable and economical parallel computing resource, it is important to understand how to use these environments for parallel image processing as well. In this paper we discuss our implementation of a parallel image processing software library (the Parallel Image Processing Toolkit). The library is easily extensible and hides the parallelism from the user. Inside the Toolkit, a message-passing model of parallelism is designed around the Message Passing Interface (MPI) standard. Experimental results are presented to demonstrate the parallel speedup obtained with the Parallel Image Processing Toolkit in a typical workstation cluster over a wide variety of image processing tasks. We also discuss load balancing and the potential for parallelizing portions of image processing tasks that seem to be inherently sequential, such as visualization and data I/O.

I. INTRODUCTION

Because of the large data set size of high-resolution image data (typical high-resolution images can be on the order of 10,000×10,000 pixels), most desktop workstations do not have sufficient computing power to perform image processing tasks in a timely fashion. The processing power of the typical desktop workstations can therefore become a severe bottleneck in the process of viewing and enhancing high resolution image data. Many image processing tasks exhibit a high degree of data locality and parallelism and map quite readily to specialized massively parallel computing hardware [1–7]. However, special-purpose machines have not been cost-effective enough (in terms of the necessary hardware and software investment) to gain widespread use. Rather, it seems that the near-term future of parallel computing will be dominated by medium-grain distributed memory machines in which each processing node has the capabilities of a desktop workstation. Indeed, as network technologies continue to mature, clusters of workstations are themselves being increasingly viewed as a parallel computing resource. The advantages of medium-grain parallel computing are low cost and high utility. The disadvantages are relatively high communication costs and irregular load patterns on the computing nodes.

In this paper, we describe the design and implementation of the *Parallel Image Processing Toolkit* (PIPT) library, a scalable, extensible, and portable collection of image processing routines. The PIPT uses a message passing model based on the Message Passing Interface (MPI) standard and is specifically designed to support parallel execution on heterogeneous workstation clusters and parallel machines.

The next section briefly describes cluster-based parallelism, discussing mainly the data distribution and communication issues involved. Section III explains the design of the PIPT, while Section IV provides an example of how a typical image processing function is implemented. Parallel performance results obtained from using the PIPT to perform some common image processing tasks are given in Section V. For the tasks shown, the PIPT obtains a nearly linear speedup as a function of the number of processors used. Finally, Section VI contains our conclusions and suggestions for future work.

II. PARALLEL IMAGE PROCESSING

The structure of the computational tasks in many low-level and mid-level image processing routines readily suggests a natural parallel programming approach.

A. Data Distribution

Many image processing algorithms exhibit natural parallelism in the following sense: the input image data required to compute a given portion of the output is spatially localized. In the simplest case, an output image is computed simply by independently processing single pixels of the input image. More generally, a neighborhood (or window) of pixels from the input image is used to compute an output pixel, as shown in Figure 1. Since, the values of the output pixels do not depend on each other, their values can be computed independently and in parallel.

A fine-grained parallel decomposition of a window operator based image processing algorithm would assign an output pixel per processor and assign the necessary windowed data required for each output pixel to the corre-

*Laboratory for Scientific Computing; Department of Computer Science and Engineering; University of Notre Dame; Notre Dame, IN 46556 {squyres,mccandless,lumsdaine}@lsc.nd.edu

†Laboratory for Image and Signal Analysis; Department of Electrical Engineering; University of Notre Dame; Notre Dame, IN 46556; Stevenson.1@nd.edu

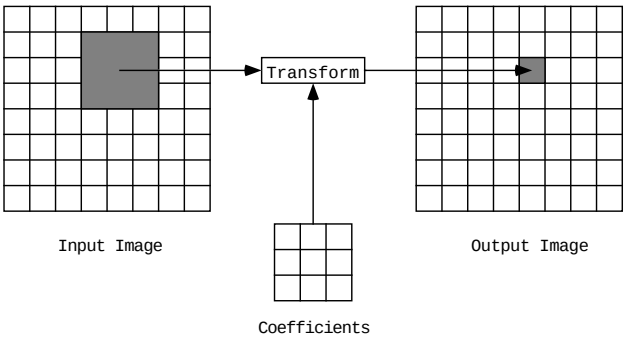

Figure 1: Data dependency for an image processing algorithm using a window operator.

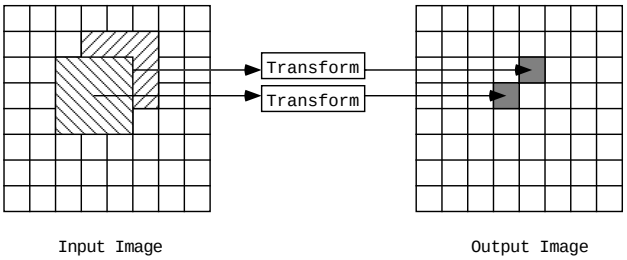

Figure 2: Fine-grained parallel decomposition for an image processing algorithm using a window operator. The parallel computation of two diagonally adjacent pixels is shown.

sponding processors. Each processor would perform the necessary computations for their output pixels. An example fine-grained decomposition of an image is shown in Figure 2. A coarse-grained decomposition (suitable for MIMD or SPMD parallel environments) would assign large contiguous regions of the output image to each of a small number of processors. Each processor would perform the appropriate window based operations to its own region of the image. Appropriate overlapping regions of the image would be assigned to properly accommodate the window operators at the image boundaries. An example coarse-grained decomposition of an input image is shown in Figure 3.

B. Message Passing

One of the challenges in developing a parallel image processing library is making it portable to the various (and diverse) types of parallel hardware that are available (both now and in the future). In order to make parallel code portable, it is important to incorporate a model of parallelism that is used by a large number of potential target architectures. The most widely used and well understood paradigm for the implementation of parallel programs on distributed memory architectures is that of *message passing*. Several message passing libraries are available in the public domain, including p4 [8], Parallel Virtual Machine (PVM) [9], PICL [10], and Zipcode [11]. Recently, a core of library routines (influenced strongly by existing libraries) has been standardized in the Message Passing Interface (MPI) [12,13]. Public domain implementations of MPI are widely available for parallel hardware and clusters of workstations, and most high-performance computing vendors have released native implementations of MPI optimized for their hardware.

C. Goals

The following goals guided the design of the PIPT:

1. The PIPT must be portable to different computing environments. The library must be able to be automatically reconfigured for new environments.
2. The PIPT must be able to execute on homogeneous or heterogeneous clusters of workstations using TCP/IP communication.
3. The PIPT must be able to execute on dedicated parallel hardware using vendor-supplied optimized communication libraries via MPI.
4. The PIPT must hide parallelism completely from users of PIPT image processing routines. That is, users should be able to call parallel image processing routines in a serial fashion.
5. The PIPT should hide parallelism to the greatest extent possible from programmers wishing to add new image processing routines or computational kernels to the PIPT. The PIPT library should provide "wrapper" functions to interface to the parallel transport mechanism.
6. The PIPT must be interoperable with other parallel libraries that may be used in an application using the PIPT. The PIPT must use a statically scoped message space so that PIPT messages do not conflict with other messages from other parallel library routines that may be used in the same application with the PIPT.

III. Design Overview

One method of parallelizing a library such as the PIPT is to parallelize separately each routine contained in it. However, this approach exposes the details of parallelization to anyone wishing to add routines to the PIPT — something that we deliberately want to avoid. Instead of making each routine in the PIPT explicitly parallel, we exploit the fact that a large number of the image processing routines use a relatively small number of shared computational kernels. In addition, since each kernel uses a similar methodology to implement their respective computational engines, an opaque transport mechanism is used for nearly all message passing and parallel aspects of the PIPT, making the parallelism nearly invisible at the kernel level as well.

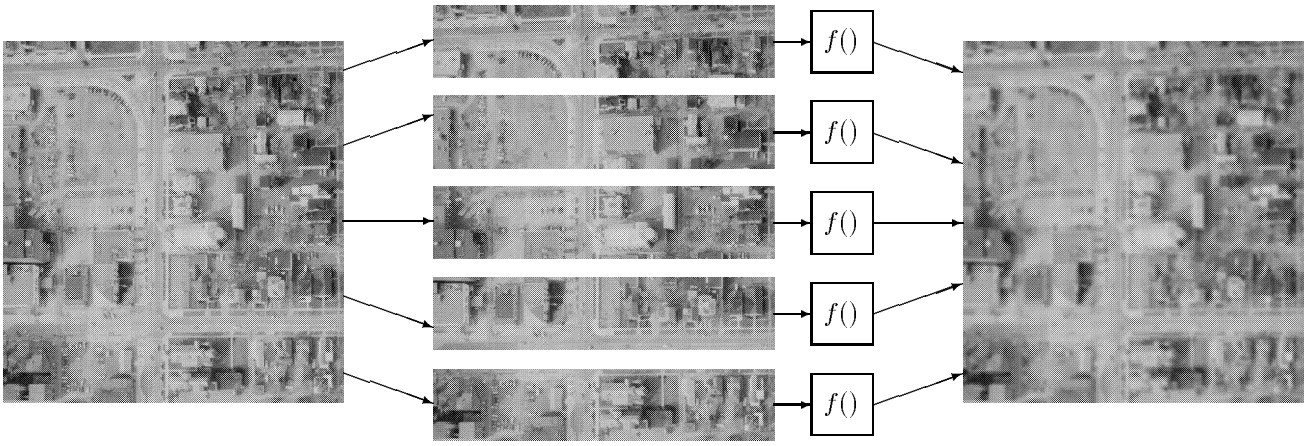

Figure 3: Coarse-grained parallel decomposition for an image processing algorithm using a window operator.

The PIPT uses a manager/worker scheme in which a manager process reads an image file from disk, partitions it into equally sized slices, and sends the pieces to worker processors. The worker processors invoke a specified image processing routine to process their slices and then send the processed slices back to the manager. The manager reassembles the processed slices to create a final processed output image. Since the PIPT's internal image format stores rows of pixels contiguously in memory, slices are likewise composed of rows of pixels from the original image.

A detailed diagram of the interaction between the various components of the PIPT is shown in Figure 4. Applications use the PIPT by calling image processing routines which are in turn built up from abstract computation kernel functions. The kernel functions interface to an opaque transport layer which transparently effects parallel execution. The transport mechanism makes calls to an MPI library for its parallel communication operations. Although the PIPT provides for parallel execution of image processing routines, parallelism is encapsulated at a low level of the system so that users of the PIPT do not need to be concerned with parallel programming. Additionally, MPI functions allow the PIPT to create its own message passing space that is guaranteed not to conflict with any other messages from the user's application.

The user application must provide a raw image as well as function parameters to the PIPT library function. The library function passes the raw image, parameters, and a local processing function to a PIPT computational kernel. It is this local processing function which will be applied by the computational kernel on the worker nodes to actually process the image. In the C programming language, a function can be specified in this way by specifying a pointer to it. Unfortunately, in a distributed memory computing environment, a function pointer is not a meaningful way of specifying functions on remote compute nodes. Thus, each compute node constructs a translation table and stores the local memory addresses of necessary functions. Similarly, translation tables are established for function parameters and local state variables. Global (across all compute nodes) indices are then established for each function, function parameter, and variable that is necessary for the routine. Therefore, functions, parameters, and variables can be specified remotely merely by sending the appropriate index as a message.

IV. Analysis

There are some subtleties in the PIPT implementation which are best illustrated by way of example. Figure 5 shows a listing of the code necessary to implement a parallel average filter that is included in the PIPT library.

A. Necessary Functions for Parallel Image Processing Routines

There are only three functions that are required to implement the average filter:

- `PIPAverage()`: PIPT entry library function for performing average filtering of an image.
- `PIPAverage_Register()`: Function for *registering* the average filter routine and routine-specific parameters that need to be communicated to the workers by the manager (see below).
- `WindowOperator()`: Local function for computing each output pixel — it is called once for each output pixel by `PIPT_Process_window()` (see below).

`PIPAverage()` is the entry function that is invoked by the user program executing on the manager node. The computational kernel used by `PIPAverage()` is `PIPT_Process_window()`. However, kernels are never directly invoked. Instead, they are called through the indirection function `PIPT_Kernel()` (see Figure 5) for

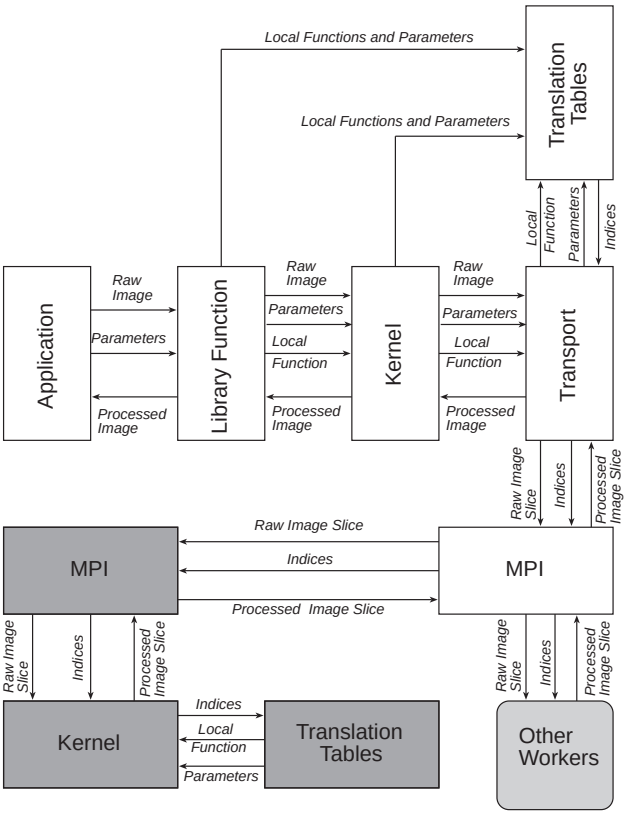

Figure 4: Detailed diagram of the interaction between the various components of the PIPT. Processing starts with the block labeled "Application." Components located on worker processors are shown in grey.

```
#include "pipt.h"
static int WindowSize;

void PIPAverage_Register()
{
  PIPT_Register_routine(PIPAverage,
    WindowOperator, PIPT_Process_window);
  PIPT_Register_parameter(PIPAverage, PIPT_INT,
    PIPT_BROADCAST, &WindowSize);
}

IMAGE *PIPAverage(IMAGE *InputImage, int
  WinHeight, int WinWidth)
{
  int Overlap = WinHeight / 2;
  IMAGE *ret;

  WindowSize = WinHeight * WinWidth;
  PIPT_Kernel_start(PIPT_Process_window);
  PIPT_Kernel_Param(&WinHeight);
  PIPT_Kernel_Param(&WinWidth);
  PIPT_Kernel_Param(&InputImage, &Overlap);
  ret = PIPT_Kernel(WindowOperator);
  PIPT_Kernel_end();

  return ret;
}

static PIXEL WindowOperator(IMAGE *InputWindow)
{
  int i, Val = 0;
  PIXEL *Pixels = InputWindow->data[0][0];
  for (i = 0; i < WindowSize; i++)
    Val += Pixels[i];
  return (PIXEL) (Val / WindowSize);
}
```

Figure 5: Complete code listing for the PIPT implementation of a parallel average filter.

internal setup and bookkeeping reasons. Likewise, parameters are not passed directly to `PIPT_Process_window()` in the conventional sense; pointers to the kernel specific parameters must be passed through `PIPT_Kernel_param()`. `PIPT_Process_window()` requires three such kernel specific parameters: `WinHeight`, `WinWidth`, and `InputImage`. For convenience, user routines can call a wrapper function to the kernel which makes the calls to `PIPT_Kernel_param()` automatically. The wrapper function for `PIPT_Process_window()` is `ProcessWindow()`. Using these two wrapper functions, the body of `PIPAverage()` is reduced to two lines of code.

`PIPT_Process_window()` returns the processed output image to `ProcessWindow()`, which returns it to `PIPAverage()`, which returns it to the calling function.

B. Registration

To manage the communication of function pointers and parameters, the PIPT uses a *registration process* that generates a unique identifier for each routine that is registered. It also associates the parameters of the routine with the unique identifier so that the parameter values can be broadcast to the workers. The function `PIPAverage_register()` registers the average filter routine, its window operator function `WindowOperator()`, and its associated parameter, `WindowSize`. The PIPT library call `PIPT_Register_routine()` creates an entry in the PIPT's internal Routine Table for the average filter. It saves a function pointer to the entry point of the function, a pointer to its window operator function, and a pointer to the computational kernel that it will call. After the pointers have been stored in a new entry in the Routine Table, the internal index for that entry serves as the unique identifier for the registered function.

The `PIPT_Register_parameter()` library call adds a pointer to a list of parameters that is associated with a particular entry in the Routine Table. Its arguments are a pointer to the entry point of the associated function, the data type of the parameter, what communication pattern

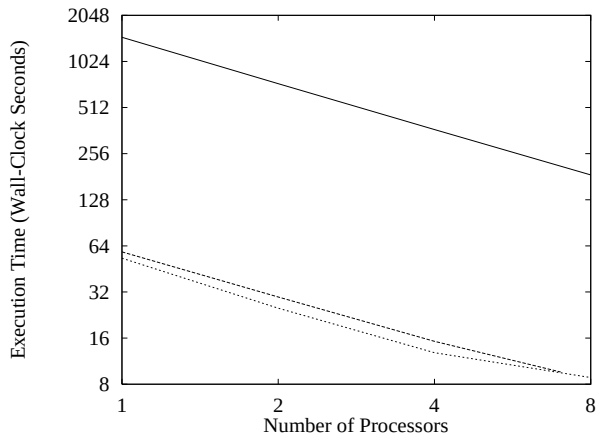

Figure 6: Execution times (in wall-clock seconds) for a parallel square median filter on (from top to bottom) a cluster of Sparc 5's, an IBM SP-2, and an SGI PowerChallenge.

should be used with this parameter (broadcast, scatter, gather, or reduce), and a pointer to the parameter itself.

V. Experimental Results

In this section we present experimental results using the PIPT to perform some common image processing tasks. In order to check the correctness of the parallel implementation, a large number of images were processed in parallel and compared to images processed with the same parameters using a sequential version of the PIPT. In all cases, the output images produced by the parallel routines exactly matched the output images produced by the serial routines.

The input image used for the parallel image processing experiments discussed below was the 1455×1540 TIFF image file previously shown in Figure 3. The experiments were conducted on a network of Sun SPARCStation 5 workstations connected with standard (10 Mbit/s) Ethernet, an IBM SP-2, and an SGI PowerChallenge. The `mpich` and `LAM` implementations of MPI (from Argonne National Labs and Mississippi State University, and Ohio Supercomputing Center, respectively) were used [14,15] for the LAN-based experiements.

A. Parallel Performance

Figure 6 shows a plot of execution time (in wall-clock seconds) as a function of the number of processors used for a parallel square median filter on three different types of parallel architectures. Each run used a window size of 17×17 pixels.

To quantify parallel performance, we define parallel *speedup* as:

$$\text{speedup} = \frac{T_{\text{par}}}{T_{\text{seq}}}$$

where T_{par} and T_{seq} are measured parallel and sequential wall-clock execution times, respectively. The observed speedup for the parallel square median filter scaled nearly linearly with the number of processors, achieving a speedup of a factor of close to 8.0 on eight processors on all architectures.

B. Load Balancing

A simple dynamic load balancing algorithm has been incorporated into the PIPT. Instead of dividing up the input image such that each worker has only one image slice to process, the dynamic load balancing algorithm divides the image into many small slices and puts them into a pool. Each worker is given a slice to process from the pool on a first-come, first-serve basis. Faster (or less loaded) workstations will automatically receive more work than slower (or more loaded) workstations. This approach attempts to keep all workers busy until the whole image has been processed, but not necessarily with the same amount of work. However, there is a tradeoff between granularity and communication overhead. Ideally, the slices can be arbitrarily small so that each processor, no matter how loaded, will never have to spend a significant amount of time processing it. Unfortunately, the communication cost for sending a large number of small packets over a LAN quickly becomes too expensive as the slice size is decreased.

VI. Conclusions

The PIPT demonstrated a significant speedup in processing large images compared to sequential functions. For the examples shown in this paper, the PIPT obtained a nearly linear speedup as a function of the number of workstations used. The PIPT's simple interface allows users to easily develop new parallel image processing routines. Since the parallel aspects of the PIPT are contained in a low level opaque transport mechanism of the library, the user can design and implement image processing algorithms and kernels without having to consider the complexities of parallel programming.

Clusters of workstations are already widely available, so the cluster-based approach will continue to be an effective, practical, and economical mode of distributed memory parallel computing. As more vendors introduce native implementations of MPI (tuned for their particular platforms), software that is based on MPI (such as the PIPT) will immediately be able to take advantage of the increased efficiency.

A. Further Extensions to PIPT

There are still several operations in the PIPT which contribute significantly to the overall processing time. Disk I/O is inherently sequential, as are scattering and gathering the image data between the manager and workers. Intelligent

algorithms can be devised to take advantage of distributed I/O mechanisms.

Image visualization also presents an interesting opportunity for parallelization. Presently, the manager gathers the processed image from the workers and then post-processes it in order to render it on the display device. By distributing visualization routines in the same manner as the processing routines, the PIPT can gain added overall parallel efficiency [17].

Data caching can also improve performance. If a node caches the last several image slices that it processed, it is possible to apply a new image processing algorithm to these slices without having to re-send them across the network.

Much of the development effort in this version of the PIPT focused on proper encapsulation of the data structures and in providing a consistent programming interface to the library. A natural extension of this effort would be to implement the PIPT in a true object-oriented fashion using C++.

Many high-end workstations contain multiple processors that operate in a symmetric multiprocessor mode. Many of the issues and design decisions made for the PIPT will be different in a symmetric multiprocessing environment. Thus, it is particularly appropriate to study an implementation of PIPT that is aimed particularly at symmetric multiprocessing workstations, and/or to a networked cluster of such machines.

For image exploitation, it is often economical to maintain a central database of imagery data that are served to image processing clients. There are several emerging technologies that provide Gbit/s data transfer rates — ATM being the premiere among these. A study of Gbit/s network technology for image exploitation is a natural follow on to the PIPT.

B. Availability

A public domain implementation of the PIPT can be obtained via the World Wide Web at URL

```
http://www.cse.nd.edu/
  ~lsc/research/PIPT/
```

VII. ACKNOWLEDGMENTS

Effort sponsored by Rome Laboratory, Air Force Materiel Command, USAF under grant number F30602-94-1-0016. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Rome Laboratory or the U.S. Government.

VIII. REFERENCES

[1] C. M. Chen, S.-Y. Lee, and Z. H. Cho, "A parallel implementation of 3D CT image reconstruction on HyperCube multiprocessor," *IEEE Transactions on Nuclear Science*, vol. 37, no. 3, pp. 1333–1346, 1990.

[2] R. L. Stevenson, G. B. Adams, L. H. Jamieson, and E. J. Delp, "Parallel implementation for iterative image restoration algorithms on a parallel DSP machine," *Journal of VLSI Signal Processing*, vol. 5, pp. 261–272, April 1993.

[3] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems," *IEEE Computer*, vol. 25, pp. 54–63, February 1992.

[4] C. H. Chu, E. J. Delp, L. H. Jamieson, H. J. Siegel, F. J. Weil, and A. B. Whinston, "A model for an intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 598–662, June 1989.

[5] L. H. Jamieson, E. J. Delp, C. C. Wang, J. Li, and F. J. Weil, "A software environment for parallel computer vision," *IEEE Computer*, vol. 25, pp. 73–77, February 1992.

[6] K. Kim and V. K. P. Kumar, "Parallel memory systems for image processing," in *Proceedings of the 1989 Conference on Computer Vision and Pattern Recognition*, pp. 654–659, 1989.

[7] D. Lee, *A Multiple-processor Architecture for Image Processing*. PhD thesis, University of Alberta, 1987.

[8] R. Butler and E. Lusk, "User's guide to the p4 programming system," TM-ANL 92/17, Argonne National Laboratory, Argonne, IL, 1992.

[9] A. Beguilin *et al.*, "A users' guide to PVM parallel virtual machine," ORNL/TM 11826, Oak Ridge National Laboratories, Oak Ridge, TN, 1992.

[10] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "A user's guide to PICL: A portable instrumented communication library," ORNL/TM 11616, Oak Ridge National Laboratories, Oak Ridge, TN, October 1990.

[11] A. Skjellum and A. Leung, "Zipcode: A portable multicomputer communication library atop the reactive kernel," in *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pp. 767–776, IEEE Press, 1990.

[12] M. P. I. Forum, "Document for a standard message-passing interface," Tech. Rep. Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[13] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[14] N. E. Doss, W. Gropp, E. Lusk, and A. Skjellum, "An initial implementation of MPI," Tech. Rep. MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[15] G. Burns, R. Daoud, and J. Vaigl, "Lam: An open cluster environment for mpi," in *Proceedings of Supercomputing Symposium '94*.

[16] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *Euro-Par'96, LIP*, (ENS Lyon, France), 1996.

[17] LAM Project, "XMTV - A Graphics Server for LAM," October 1994. http://www.osc.edu/Lam/lam/xmtv.html.