A cluster-based parallel image processing toolkit[*]

Jeffrey M. Squyres Andrew Lumsdaine

Laboratory for Scientific Computing
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

Robert L. Stevenson

Laboratory for Image and Signal Analysis
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556

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 network technologies continue to mature, workstation clusters are becoming a viable and economical parallel computing resource, so 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 Toolkit uses a message-passing model of parallelism 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.

Keywords: cluster based computing, image processing, MPI, parallel processing

1 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$\times$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 wide-spread 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. And, with the advent of such machines as the IBM SP-2, the distinction between workstation clusters and real parallel machines becomes increasingly blurred. The advantages of cluster-based parallel computing are low cost and high utility. The disadvantages are high communication latency and irregular load patterns on the computing nodes.

[*]Presented at the *IS&T Conference on Image and Video Processing, 1995*, San Jose, CA, February 5–10, 1995.

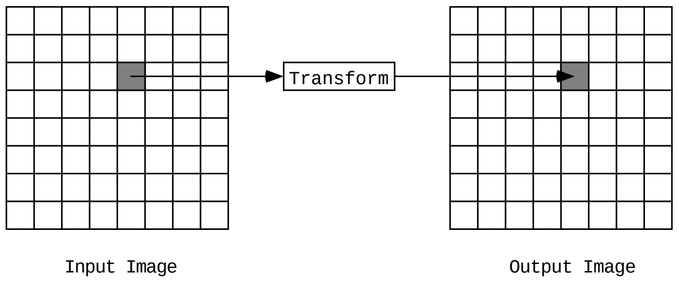

Figure 1: Data dependency for an image processing algorithm using a point operator.

In this paper, we describe the design and implementation of the Parallel Image Processing Toolkit library, a scalable, extensible, and portable collection of image processing routines. The PIP Toolkit 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. The next section briefly describes cluster-based parallelism, discussing mainly the data distribution and communication issues involved. Several important points related to the implementation of the PIP Toolkit are detailed in Section 3, where we also discuss parallelization of data I/O and image visualization. Parallel performance results obtained from using the PIP Toolkit to perform some common image processing tasks are given in Section 4. For the tasks shown, the PIP Toolkit obtains a nearly linear speedup as a function of the number of workstations used. Finally, Section 5 contains our conclusions and suggestions for future work.

2 CLUSTER-BASED PARALLEL IMAGE PROCESSING

The structure of the computational tasks in many low-level and mid-level image processing routines readily suggest a natural parallel programming approach. On either a fine- or coarse-grain architecture the most natural approach is for each computational node to be responsible for computing the output image at a spatially compact set of image locations. This generally will minimize the amount of data which needs to be distributed to the individual nodes and therefore minimize the overall communication cost. Thus, this approach will generally maximize the speedup of the parallel system. This section discusses the approach for data distribution utilized in the toolkit and the message passing system used to implement it.

2.1 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, as shown in Figure 1. More generally, a neighborhood (or window) of pixels from the input image is used to compute an output pixel, as shown in Figure 2. Clearly, the values of the output pixels do not depend on each other. Hence, the output pixels can be computed independently and in parallel. This high degree of natural parallelism exhibited by many image processing algorithms can be easily exploited by using parallel computing and parallel algorithms. In fact, many image processing routines can achieve near linear speedup with the addition of processing nodes (see Section 4).

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 processors. Each processor would perform the necessary computations for their output pixels. An example fine-grained decomposition of an input image is shown in Figure 3. 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 per-

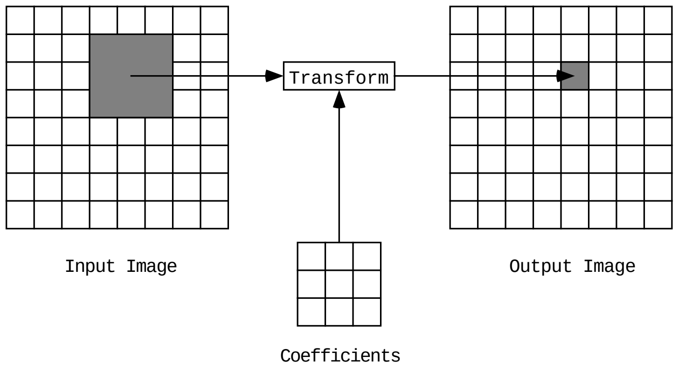

Figure 2: Data dependency for an image processing algorithm using a window operator.

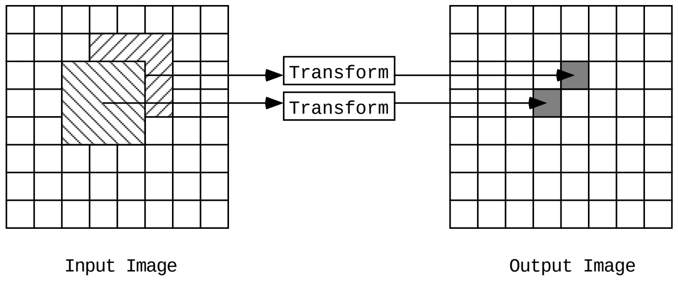

Figure 3: Fine-grained parallel decomposition for an image processing algorithm using a window operator. The parallel computation of two diagonally adjacent pixels is shown.

form 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 4.

2.2 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] Initial public domain implementations of MPI are becoming available now. It is expected that vendors of parallel machines will provide native versions of MPI for their hardware.

3 IMPLEMENTATION

The PIP Toolkit is written in C using a manager/worker scheme in which a manager program reads an image file from disk, partitions it into equally sized pieces, and sends the pieces to worker programs running on machines in the cluster. The worker programs invoke a specified image processing routine to process their sub-images and then sends the processed sub-images back to the manager. The manager re-assembles the processed sub-images to create a final processed output image.

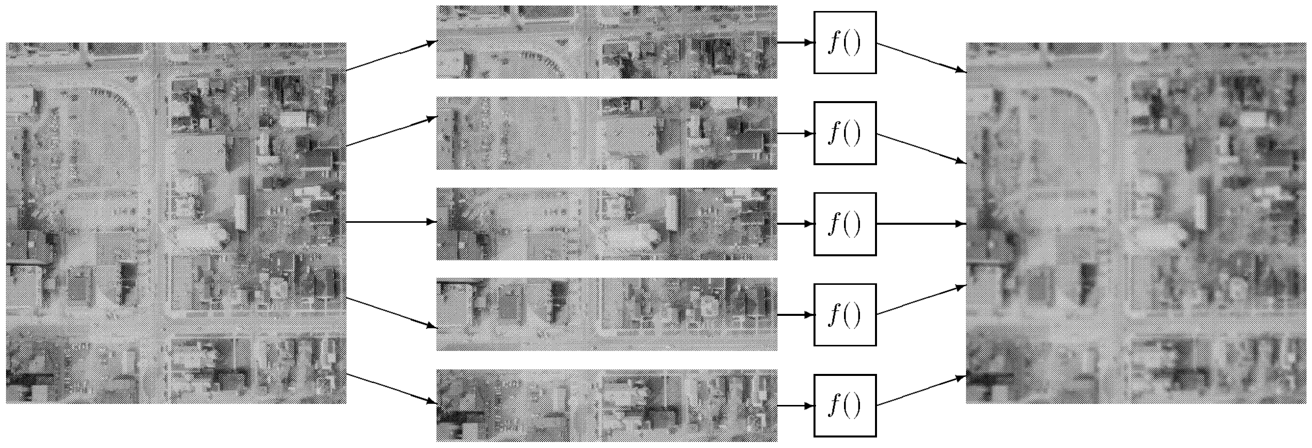

Figure 4: Coarse-grained parallel decomposition for an image processing algorithm using a window operator.

Note that, in order to maximize efficiency, the machine on which the manager program is running may also have a worker program running (or, equivalently, the manager program itself may process one of the image pieces).

3.1 Parallelizing image processing functions

One method of parallelizing a library such as the PIP Toolkit is to separately parallelize each routine contained in it. However, this approach exposes the details of parallelization to anyone wishing to add routines to the toolkit — something that we deliberately want to avoid. Instead of making each routine in the PIP toolkit explicitly parallel, we exploit the fact that a large number of the image processing routines use a relatively small number of shared computational kernels. By focusing our energies on parallelizing the computational kernels, we are able to maximize our parallel programming effort, maximize the parallel efficiency, and hide the details of parallelization from users of the Toolkit.

This approach introduces some subtleties into the implementation; these are best illustrated by way of an example. Figure 5 shows a listing of all the code necessary to implement the parallel averaging filter that is included in the PIP Toolkit library. There are only three functions that are required to implement the average filter:

`WindowOperator()`: Local function for computing each output pixel — it is called once for each output pixel by `PIPT_Process_Window()`.

`PIPAverage()`: PIP Toolkit 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.

`PIPAverage()` is the top-level function that is invoked by the user program executing on the manager node. The computational kernel used by `PIPAverage()` is `PIPT_Process_Window()`, which is declared as

```
IMAGE *PIPT_Process_Window(IMAGE *input, int height, int width, PIXEL (*Op)());
```

The arguments to `PIPT_Process_Window()` are the input image, the window height and width, and a pointer to the function used to calculate the values of the output pixels. `PIPT_Process_Window()` returns the processed output image to `PIPAverage()`, who returns it to the calling function.

```
#include <PIPTdatatype.h>
#include <PIPInternal.h>
#include <PIPRegister.h>
#include <Filter.h>

static int WindowSize;

static PIXEL WindowOperator(IMAGE *InputWindow)
{
  int i, Val = WindowSize / 2;
  PIXEL *Pixels = InputWindow->data[0][0];

  for (i = 0; i < WindowSize; i++)
    Val += Pixels[i];
  return (PIXEL) (Val / WindowSize);
}

IMAGE *PIPAverage(IMAGE *InputImage, int WinHeight, int WinWidth)
{
  WindowSize = WinHeight * WinWidth;

  return PIPT_Process_Window(InputImage, WinHeight, WinWidth, WindowOperator);
}

void PIPAverage_Register()
{
  PIPT_Register_Routine(PIPAverage, WindowOperator, NULL, NULL);
  PIPT_Register_Parameter(PIPAverage, PIPT_INT, &WindowSize);
}
```

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

The `PIPT_Process_Window()` function is one of the computational kernels in the PIP toolkit that is actually parallelized. This function divides the input image into sub-images, scatters the sub-images to the worker processors for processing, gathers the results, and returns the processed output image. The actual processing done on the workers is accomplished by repeatedly applying the function pointed to by the input parameter `Op()` to windows centered around each pixel of the input image. In our example, `Op()` would point to `WindowOperator()`.

The difficulty in parallelizing `PIPT_Process_Window()` is that although the manager program can use the function pointer to call the required window processing operator, the value of the function pointer on the manager would, in general, not be meaningful (as a function entry point) to the workers. In addition, there are often local parameters that are used by the window processing operator (i.e., `WindowSize` in our example). The value of this parameter is also only contained on the manager and must be communicated to the workers. Since different filters have different local functions and different local data, a general means of communicating the required information was incorporated into the PIP Toolkit.

In particular, the PIP Toolkit uses a *registration process* that generates a unique identifier for each routine that is registered. It also associates routine-specific parameters with the unique identifier so that the parameter values can be broadcast to the workers. The function `PIPAverage_Register()` registers both the average filter routine and its associated parameter, `WindowSize`. The PIP Toolkit library call `PIPT_Register_Routine()` creates an entry in the Toolkit's internal Routine Table for the average filter. It is declared as

```
void PIPT_Register_Routine(IMAGE *(*Top)(), PIXEL (*Op)(),
                           void (*Init)(), void (*Term)());
```

Its arguments are function pointers to the entry point of the function, its window operator function, and its initialization and termination functions, respectively. After the pointers have been stored in a new entry in the Routine Table, the 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.

```
void PIPT_Register_Parameter(IMAGE *(*Top)(), PIPT_DATATYPE PType, void *Param);
```

The arguments are a pointer to the entry point of the associated function, the data type of the parameter, and a pointer to the parameter itself. `PIPT_Register_Parameter()` associates `PType` and `Param` with the function specified by `Top` in the Routine Table. Thus, using the Routine Table, `PIPT_Process_Window()` can look up the index of the `Op()` function that is passed to it and broadcast this index to the workers. In addition to the index, `PIPT_Process_Window()` also broadcasts any routine-specific parameters that are associated with the entry in the Routine Table.

Before a user program can use any of the PIP Toolkit functions, it must call `PIPT_Init()`. The `PIPT_Init()` function initializes the Toolkit and prepares for parallel execution. One of its initialization steps is to call `PIPT_Register_All()`. `PIPT_Register_All()` registers all of the image processing routines in the PIP Toolkit by invoking their registration functions:

```
void PIPT_Register_All()
{
  PIPAverage_Register();
  PIPSquareMedian_Register();

   ··· Registration calls for all other library functions ···

  PIPT_Register_User_Routines();
}
```

Both the manager and the workers call `PIPT_Register_All()`, so the indexing for the functions on the manager and workers will be consistent. `PIPT_Register_User_Routines()`, invoked at the end of `PIPT_Register_All()`, is a call-back function that can be supplied by the user to call registration functions of new image processing routines. In this way, users can utilize the parallel aspects of the Toolkit without being required to actually add their functions into the Toolkit library itself.

3.2 Load balancing

A dynamic load balancing algorithm has been proposed which will soon be incorporated into the Parallel IP Toolkit. Instead of dividing up the input image such that each worker has only one sub-image to process, the dynamic load balancing algorithm divides the image into many small sub-images and puts them into a pool. Each worker is given a sub-image 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 will keep all workers busy until the whole image has been processed, but not necessarily with the same amount of work.

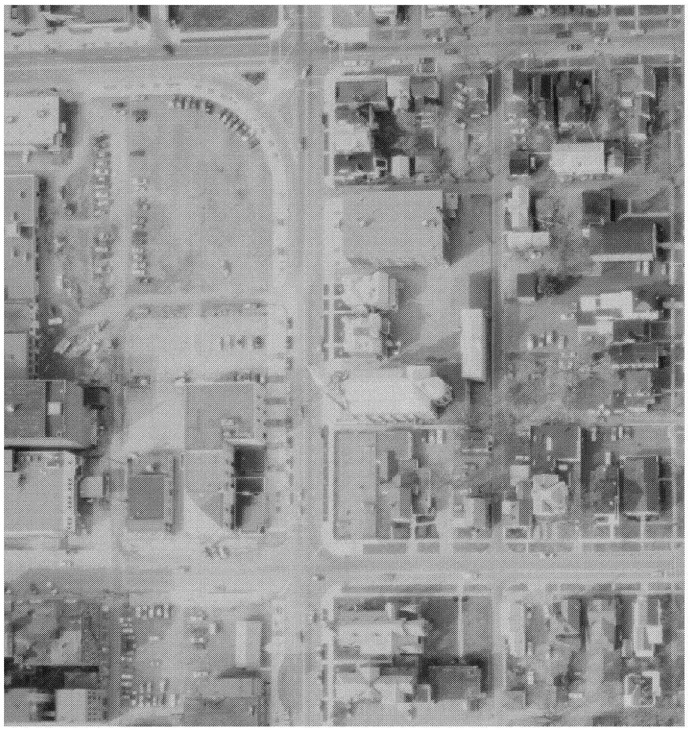

Figure 6: Input image for preliminary experiments.

3.3 Further opportunities for parallelization

There are still several operations in the PIP Toolkit 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. To parallelize these operations in a workstation environment, however, would require distributing the images *a priori* across the network. This would unfortunately conflict with the load balancing scheme described above. Study of the tradeoffs involved between automatic load balancing and parallelized I/O will be part of the future work for this project.

Finally, image visualization presents an interesting opportunity for parallelization. Presently, the manager gathers the processed image from the workers and then processes it some more in order to render it on the display device. By distributing visualization routines in the same manner as the processing routines, the PIP Toolkit can gain added overall parallel efficiency.

4 EXPERIMENTAL RESULTS

In this section we present experimental results using the Parallel IP Toolkit to perform a variety of 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 PIP Toolkit. 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 was the 1455×1540 TIFF image file shown in Figure 6. The experiments were conducted using a network of Sun SPARCStation 5 workstations connected with standard (10 Mbit/s) Ethernet. The `mpich` implementation of MPI (from Argonne National Labs and Mississippi State University) was used.[14]

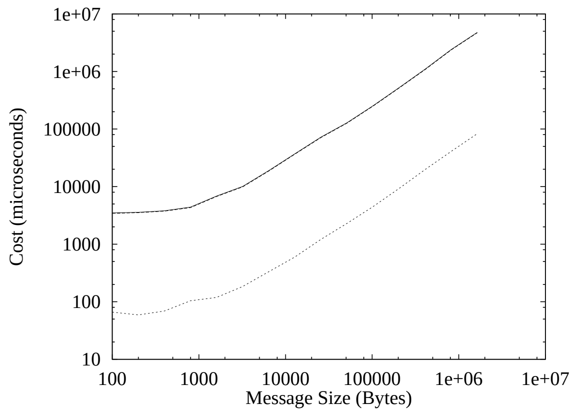

Figure 7: Graph of time required to assemble and send messages of different sizes in a workstation cluster. The time required to pack the messages is shown with the dotted line, the time required to send the message is shown with the dashed line, and the total time is shown with the solid line.

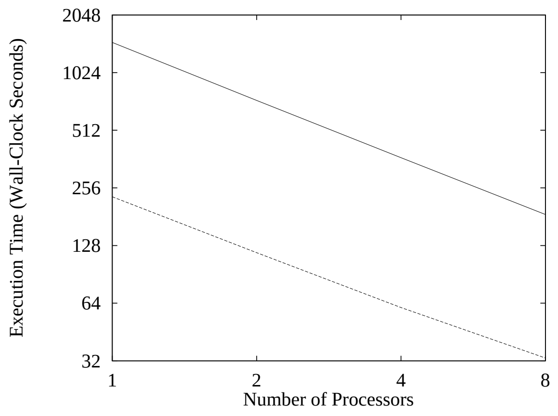

Figure 8: Execution times (in wall-clock seconds) for parallel parallel squared median filter (solid) and average filter (dashed).

4.1 Communication Costs

Figure 7 illustrates the costs of sending messages using MPI in a workstation cluster environment. Messages of sizes from 100 bytes to 1,000,000 bytes (1 Mbyte) were sent from manager to worker in the cluster used for the experiments in this section. The time required to assemble (pack) and send the messages was recorded and plotted as a function of the message size. Note that the cost for the smaller messages is dominated by latency time (approximately 3.5 milliseconds), but that for larger messages, the cost becomes linearly proportional to the message size. One conclusion that can be drawn is that, in workstation cluster-based computation, one should strive to minimize the number of communication operations and at the same time, one should maximize the size of the messages that are sent.

4.2 Parallel performance

Figure 8 shows a plot of execution time (in wall-clock seconds) as a function of the number of processors used for the parallel average filter and the parallel square median filter. Both filters used a window size of 17×17.

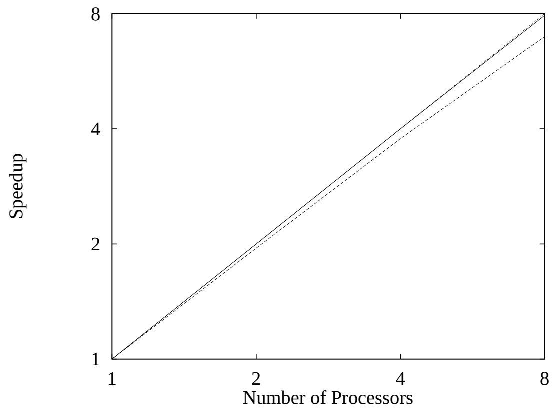

Figure 9: Comparison of ideal speedup (dotted) with observed speedup for parallel square median filter (solid) and parallel average filter (dashed).

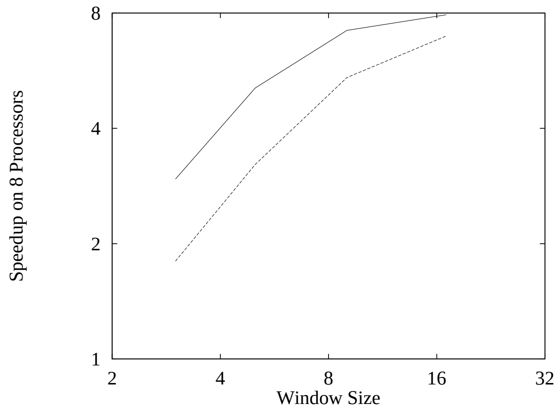

Figure 10: Observed parallel speedup as a function of window size for parallel square median filter (solid) and parallel average filter (dashed).

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. Figure 9 shows a comparison of observed speedup with ideal speedup for the parallel square median filter and the parallel average filter. The observed speedup for the parallel square median filter scaled nearly linearly with the number of processors, achieving a speedup of a factor of 7.92 on eight processors. The observed speedup for the parallel average filter was 6.97 on eight processors. The difference between the parallel square median filter and the parallel average filter was due to the different amount of computation required by each (since they both processed the same input image, the amount of communication was same). The parallel square median filter required substantially more computation than did the parallel average filter, so the communication cost did not as noticeably impact the speedup.

The same effect can be seen in Figure 10, where we show observed speedup on 8 processors as a function of the window size of the filter (an indication of the amount of computation). For the larger window sizes, and hence, for the larger amounts of computation, larger speedup is observed.

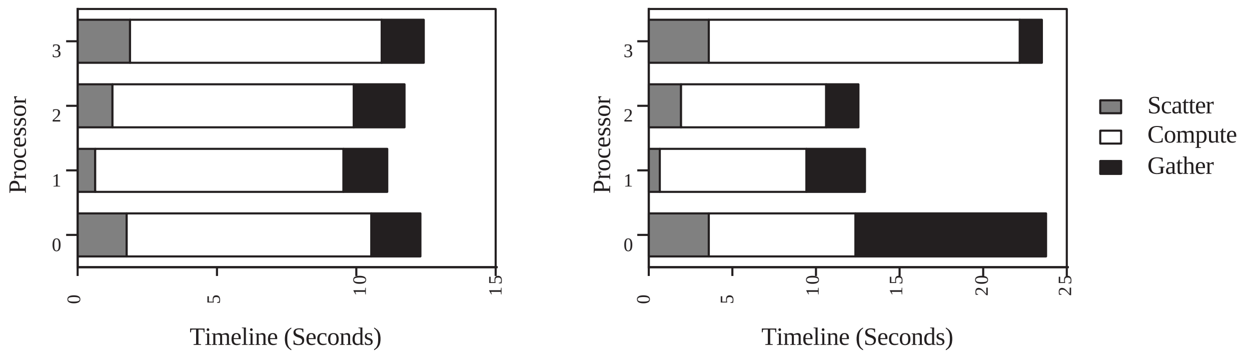

Figure 11: Communication patterns for scattering and gathering image data for four processor parallel average filter with 7×7 window. The first graph is for a cluster of four unloaded machines. The second graph is for a cluster of four machines in which processor 4 is loaded with another computationally intensive task and so takes twice as long to complete its image processing task as the other processors.

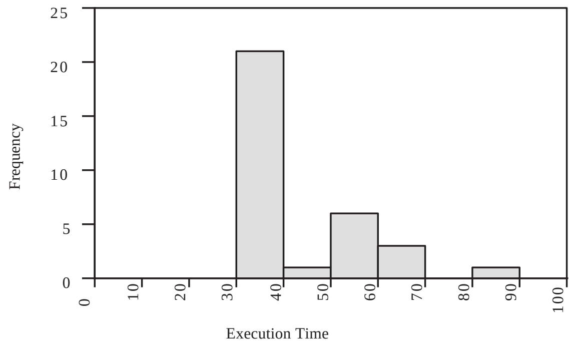

Figure 12: Histogram of execution times for 32 runs of parallel average filter on four processors.

4.3 Communication and computation patterns

Figure 11 illustrates the communication and computation patterns for execution of a parallel average filter (with a 7×7 window) on a four processor cluster. The plot data were generated using the Message Passing Extensions (MPE) to instrument MPI. The plots show the manager (processor 0) scattering the image data to the workers, then all processors computing, and then the manager gathering the processed image data back from the workers. In the first plot, the four machines are unloaded except for the image processing task. In the second plot, processor 4 is loaded with another computationally intensive task. Presently, the PIP Toolkit does not attempt to do any load balancing. As a result, parallel performance showed marked deterioration if one of the processing nodes was significantly loaded. In Figure 11, the manager must wait during the gather stage until processor 4 finishes its computations and sends its processed sub-image. Thus, the execution time of the entire parallel job is increased by nearly a factor of two because of the load imbalance. The effect of unequally loaded nodes is also illustrated in Figure 12, which shows a histogram of execution times for 32 runs of the parallel average filter (with 17×17 window) on four processors. For about a third of the runs, one or more processors was loaded with another computationally intensive task, causing the entire parallel computation to take twice as long (or longer) to complete.

The total communication time required to scatter and gather the data for the example in Figure 11 was about 3.5 seconds. Communication time will increase somewhat as the number of processors increases. With two processors (one manager and one worker), only half of the image needs to be sent to the worker. With four processors (one manager and three workers), three-quarters of the image needs be sent out, etc. However, as processors are added, the amount of work required of each worker decreases, so the ratio of communication to computation will increase so that parallel efficiency will decrease.

5 CONCLUSIONS

The PIP Toolkit demonstrated a significant speedup in processing large images compared to sequential functions. For the examples shown in this paper, the PIP Toolkit obtained a nearly linear speedup as a function of the number of workstations used. The Toolkit's simple interface allows users to easily develop new parallel image processing routines. Since the parallel aspects of the PIP Toolkit are contained in lower level computational kernels of the library, the user can design and implement image processing algorithms 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 parallel computing. As vendors introduce native implementations of MPI (tuned for their particular platforms), software that is based on MPI (such as the PIP Toolkit) will immediately be able to take advantage of the increased efficiency.

Present work on the Toolkit focuses on the implementation and testing of a dynamic load balancing algorithm and the incorporation of additional image processing algorithms into the PIP Toolkit library. Once the PIP Toolkit is reasonably solid, we hope to be able to release it for public use and to obtain feedback from user experiences that will allow us to enhance its capabilities.

6 ACKNOWLEDGMENTS

Brian McCandless and John J. Tran contributed to the development and testing of the Parallel IP Toolkit. 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.

7 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.