A toolkit for parallel image processing

J. M. Squyres[†] A. Lumsdaine[†] R. L. Stevenson[‡]

[†]Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

[‡]Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556

ABSTRACT

In this paper, we present the design and implementation of a parallel image processing software library (the Parallel Image Processing Toolkit). The Toolkit not only supplies a rich set of image processing routines, it is designed principally as an extensible framework containing generalized parallel computational kernels to support image processing. Users can easily add their own image processing routines without knowledge or explicit use of the underlying data distribution mechanisms or parallel computing model. Shared memory and multi-level memory hierarchies are exploited to achieve high performance on each node, thereby minimizing overall parallel execution time. Multiple load balancing schemes have been implemented within the parallel framework that transparently distribute the computational load evenly on a distributed memory computing environment. 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 with some common image processing tasks.

Keywords: Parallel, image, toolkit, library, MPI, processing, extensible, load-balancing

1. INTRODUCTION

The typical desktop workstation does not have sufficient computational resources to support large scale image processing. The computational problem is due to the very high resolution of the imagery data (typical images size can be as high as 10,000 by 10,000 pixels). Because of this, the processing power of the typical desktop workstation can become a severe bottleneck in the enhancement of imagery data. Due to the nature of many image processing algorithms, an effective method for alleviating this problem is through parallel computation. Many image processing routines can achieve near linear speed-up with the addition of processing nodes.[1–7]

Although applying parallel computing techniques to image processing seems attractive at first glance, in general, one wants to avoid reinventing the wheel. Thus, a parallel library of common image processing tasks, as well as the capability for end-users to easily extend this library, would meet several needs. However, providing such a library for parallel image processing has certain challenges. In particular, a library must be usable in numerous parallel execution environments and be usable by a wide variety of users. Ideally, the intended audience of such a library is composed of image processing experts, not parallel computing experts. Thus, the library should present a programming interface suitable for that audience.

These issues led to the development of the Parallel Image Processing Toolkit (PIPT). The details of parallelization are hidden from users of the PIPT through a registration/call-back mechanism, which provides an opaque transport mechanism for moving parameterized data between nodes. Specific attention was given to serial performance in the PIPT library; several optimization techniques were applied to exploit data locality and multi-level memory hierarchies present in modern RISC architectures. Additionally, multiple load balancing algorithms were developed and implemented that automatically (and transparently) distribute the computational load over heterogeneous active workstation clusters in an attempt to minimize overall wall clock execution time.

To achieve a high degree of portability, the PIPT uses the Message Passing Interface (MPI) standard to effect parallelism. The leading implementations of MPI are in the public domain so that the PIPT can make use of this important standard while still remaining freely available itself.

The next section briefly describes cluster-based parallelism, discussing mainly the data distribution and communication issues involved. Section 3 explains the design of the PIPT, while section 4 briefly describes the user programming model and discusses multithreading and load balancing issues. Parallel performance results obtained from using the PIPT to perform some common image processing tasks on a cluster of workstations are given in Section 5. For the tasks shown, the PIPT obtains a nearly linear speedup as a function of the number of workstations used. Finally, Section 6 contains our conclusions and suggestions for future work.

Other author information: J. M. Squyres: `squyres@cse.nd.edu`, A. Lumsdaine: `lums@lsc.nd.edu`, R. L. Stevenson: `stevenson.1@nd.edu`.

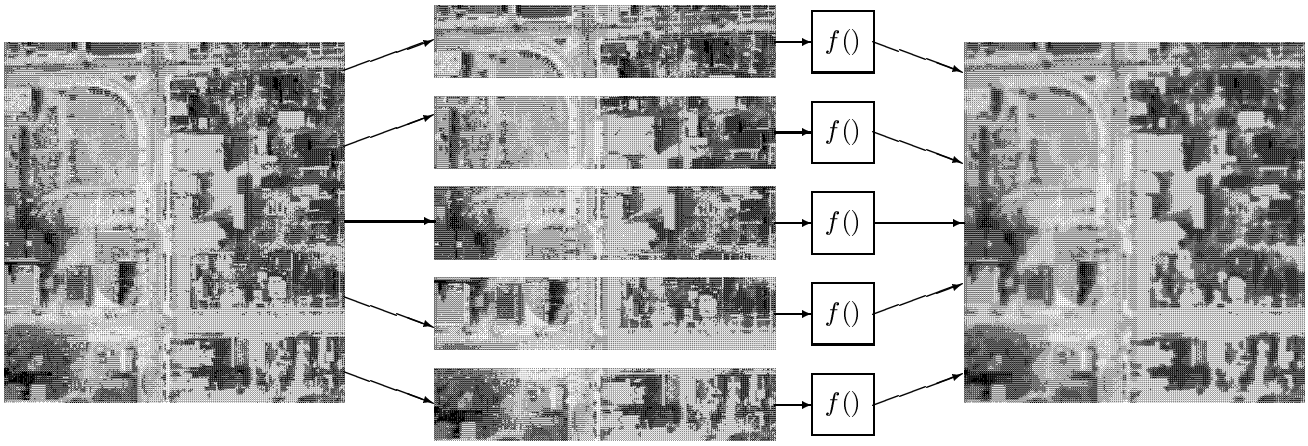

Figure 1. Coarse-grained data decomposition for an image processing algorithm.

2. 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. 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. 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

For large classes of image processing tasks, 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 dependent generally, a neighborhood (or window) of pixels from the input image is used to compute an output pixel. Hence, the output pixels can be computed independently and in parallel. This high degree of natural parallelism can be easily exploited by parallel algorithms. In fact, many image processing routines can achieve near linear speedup with the addition of processing nodes (over a reasonable number of nodes).

A fine-grained parallel decomposition of a window operator based image processing algorithm assigns an output pixel per processor and assign the necessary windowed data required for each output pixel to the corresponding processors. Each processor would perform the necessary computations for their output pixel. 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 performs 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 1.

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] In 1994, a core of library routines (influenced strongly by existing libraries) was standardized in the Message Passing Interface (MPI).[12,13] Public domain implementations of MPI are widely available. More importantly, all vendors of parallel machines and high-end workstations provide native versions of MPI optimized for their hardware.

The PIPT uses MPI for parallel infrastructure and all message passing functionality. As such, there is very little operating system or hardware code in the PIPT; it is highly portable to a wide variety of systems.

3. DESIGN OVERVIEW

One method of parallelizing a library such as the PIPT is to separately parallelize each routine contained in it. However, this approach exposes the details of parallelization to anyone wishing to add new 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 to the kernels as well.

The PIPT includes built-in kernels for both point-wise and windowed operators. Kernels iterate over either an image or a *feature* (an extraction of specific image characteristics), and apply a point-wise or windowed operator to generate output pixels. The windowed kernels include support for window operators in both the input and output images/features.

3.1. Execution Model

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 processes. The worker programs invoke a specified image processing routine to process their slices and then send the processed slices back to the manager. The manager re-assembles 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.

While the manager/worker model is not scalable to large numbers of nodes, it does scale well over machine sizes that most users are likely to have at their disposal (the present model is effective to at least 32 nodes). Even without a designated manager, the disk (and perhaps visualization) I/O of the images already represents a bottleneck which would not be alleviated by using a different model. The workers would still have to read their respective slices from a networked file system, which would effectively be serialized by the file server. Scalability to very large numbers of nodes would thus require a parallel file system as well as a manager-less collection of workers. Although commercial and public domain parallel file systems exist, making a parallel file system a prerequisite for the PIPT introduces complications that, for most users, are unnecessary.

It should be noted that the manager does not actually process any of the image slices; they are all distributed to the workers for computation. Instead, the manager only performs the data distribution and load balancing schemes that are outlined in Section 5.2. As described, the load balancing schemes will necessarily have long periods of inactivity, which results in wasted CPU cycles on the manager node. A simple alternative to these wasted cycles is to spawn a worker thread on the manager that receives and processes slices just like the other workers. This scheme was not implemented into the PIPT, however, due to problems with MPI and multithreading that are discussed in Section 4.1.1.

3.2. User Interface

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.

3.3. Opaque Transport Mechanism

A detailed diagram of the interaction between the various components of the PIPT is shown in Figure 2. 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 providing 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) index values are then established for each function, function parameter, and variable that is necessary for a given routine. Therefore, functions, parameters, variables, and their associated types can be specified remotely merely by sending the appropriate index as a message.

The transport mechanism on the manager uses MPI to pass slices of the image, as well as indices for looking up state information, to the worker nodes. The worker nodes use the indices to find and set values of local variables, process the image

slice, and finally pass the processed image slice back to the manager. The processed image is assembled by the transport mechanism on the manager and passed back to the computational kernel. The kernel passes the processed image back to the PIPT library function, which in turn passes the processed image back to the user application.

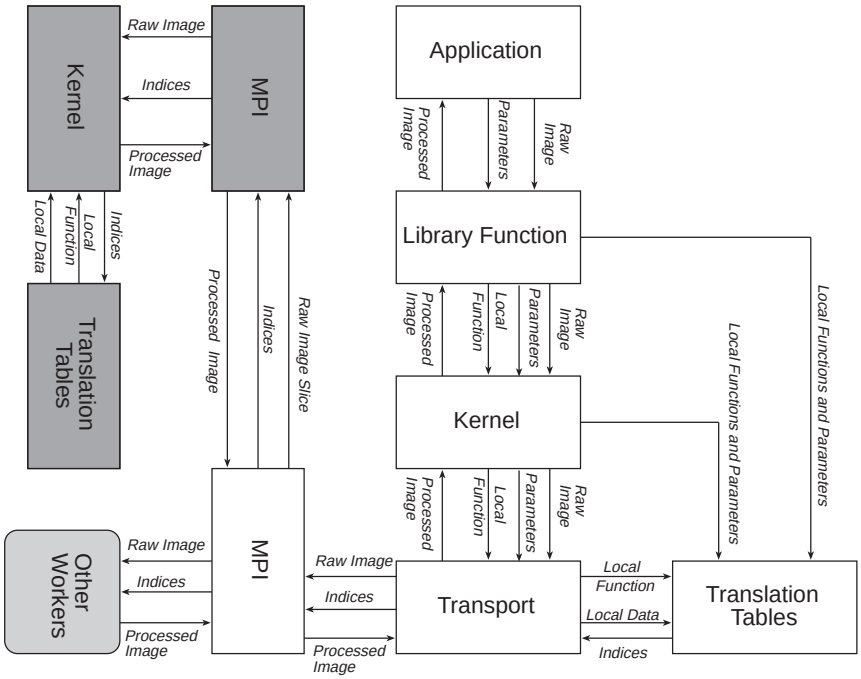

Figure 2. Detailed diagram of the interaction between the various components of the PIPT. Processing starts with the block labeled "Application." Components located on worker processors are shown in grey.

3.4. Shared Memory

On a single SMP workstation, a master CPU will still allocate work to individual processors, but the message that is passed between processors is a small control message and not a segment of the image data. The PIPT thus incorporates a hierarchical model of parallelism that allows it to function on a single workstation, a single SMP, or a cluster of workstations and/or SMPs. As mentioned above, the PIPT relies on a relatively small number of computational kernels. By multithreading these kernels, hierarchical parallelism was provided to the entire toolkit with only a modest effort and a consistent programming interface was maintained.

3.5. Living with RISC

One important design feature of modern RISC microprocessor-based workstations is a hierarchical memory system. That is, the microprocessor can operate at maximum efficiency with data in registers and in cache, but at a much lower level of efficiency for data in main memory, or even worse, for data swapped out to disk. Image processing requires not only a large number of computations but a large amount of data movement as well. If a program is not careful, the costs of data movement can easily overwhelm the costs of computation.

Computationally intensive problems must both take the hierarchical nature of the memory system into account and exploit multi-stage execution pipelines present in RISC architectures. Regarding memory usage, the fundamental task is to minimize data movement to and from main memory — once moved to cache, data should be kept there and used for as long as possible. Typical strategies for increasing memory system and pipeline performance include unrolling and blocking loops, restructuring algorithmic memory access patterns, removal of unnecessary dependencies in code blocks, and so forth.[14,15]

3.6. Load Balancing

Maximum performance and efficiency from a parallel computing environment are obtained when the processing load is properly distributed among the processing nodes. Since the entire parallel computation cannot be completed until all nodes have completed their computations, the parallel computation will be limited by the slowest processor. A single processor taking longer to complete its task than the other processors will hold up the entire parallel computation. In an ideally distributed parallel computing task, all the processing nodes will complete their tasks at precisely the same time. The goal of load balancing is to partition the parallel task in such a way that this will occur. This becomes particularly difficult in an environment such as an active heterogeneous workstation cluster because in such an environment, different processing nodes may both have different raw computing power and may be loaded by running other tasks.

There are several approaches that can be taken to load balancing, including the following:

1. Partition the parallel task into a large number of relatively small tasks such that there are many more tasks than processors. Allow the worker processors to request processing tasks on a first-come first-serve basis.
2. Actively monitor the available processing power of the processing nodes being used and adjust the partitioning of tasks to achieve optimal balance.

The first approach was incorporated into early implementations of the PIPT with encouraging results.[16,17] This approach has the disadvantage that it increases the number of communication operations that must be performed, thus increasing the total communication cost. In some situations, this might be outweighed by the savings gained by good load balancing, but in other situations it might not. Further refinements, such as adjusting the granularity of the task division, or combining this approach with the second, are discussed in Section 4.2.

4. ANALYSIS

The PIPT was designed to be easily extensible by end users; it is not necessary to modify code inside the PIPT to add image processing routines that use the parallel framework provided by the library. New functionality can be added by *registering* three routines with the PIPT framework. Registering functions with the PIPT creates internal accounting tables that enable the parallel framework to marshal arguments, distribute parameters, and gather return values during a parallel run.

The image processing routines supplied with the PIPT follow the same registration process as user-provided functions; they are essentially user-level functions that are registered during the last step of PIPT initialization. As an example of how the PIPT is extensible by the user, the implementation of the PIPT-supplied averaging filter, `IPAverage()`, is discussed below.

There are three principle functions required to implement a PIPT routine: a function entry point, a generic function for computing output data, and a registration function. The `IPAverage()` function (or, in general, any image processing routine in the PIPT). The `IPAverage()` routine uses a windowed kernel named `PIPT_Process_window()`.

1. `IPAverage()`: Entry point for performing the routine. User code calls this function to calculate an average filter on an input image; the user code has no knowledge that this function will execute in parallel. This function performs any initialization necessary and then invokes its kernel, `PIPT_Process_window()` (one of the built-in PIPT kernels).

   `PIPT_Process_window()` invokes the opaque transport mechanism which distributes `IPAverage()`'s parameters and input image slices to the workers. Each of the workers invoke `PIPT_Process_window()`, which loops over the input pixels in their respective slices, applying a user-specified processing function for each output pixel.
2. `IPAverage_Window_operator()`: Local function that, given an input window of pixels, computes an output pixel. It is repeatedly called by `PIPT_Process_window()` on each of the workers to calculate the output pixels in their respective input slices.
3. `IPAverage_Register()`: Function for registering the routine and routine-specific parameters that need to be communicated to the workers by the manager. This function is responsible for identifying the `IPAverage()` function to the PIPT, associating `IPAverage()` with the `PIPT_Process_window()` kernel and the window operator function (see #2, below), and registering both `IPAverage()`'s arguments and return value(s).

Once the workers have finished their computations, they return their output slices to the manager. The PIPT transport mechanism gathers the slices into a single image, and returns it to the invoking kernel (`PIPT_Process_window()`), who, in turn, returns it to `IPAverage()`. `IPAverage()` performs any cleanup necessary and returns the output image to the user code.

4.1. Data Handling & Shared Memory

In the PIPT, there is not only a large amount of computation related to image processing, but also a large amount of data movement both between the CPU and main memory, and between main memory and the underlying communication network. As described in Section 3.5, it is critical for the PIPT to properly manage its use of the hierarchical memory system by keeping data in the cache as long as possible.

Techniques such as loop interchange ensure that memory is accessed in a linear manner rather than in non-unit strides (which would cause cache thrashing). Loop fusion is also used; multiple loops are merged that access the same indices of the same arrays so that a given section of an array can be pulled into cache, operated on multiple times, and then discarded from the cache as opposed to loading the same section into cache multiple times.

The CPU pipeline is kept full by manually unrolling loops in PIPT routines and kernels to assist the compiler's optimizer. While compiler technology is getting better at identifying loop unrolling opportunities, only the programmer can understand the larger picture of the program and accurately select which loops can (and cannot) be unrolled. Finally, wherever possible, dependencies were eliminated between loop iterations to further allow the optimizer to unroll loops and perform other, lower-level optimizations.[14,15]

4.1.1. Multithreading Strategies

The current PIPT implementation includes threading on SMP workers. When an SMP worker receives a slice, it divides it into n sub-slices, where n is the number of CPUs in the machine (unless overridden by the user). n threads are created to process the sub-slices. The threads operate on their sub-slice independently of the other threads, and die when they are finished.

Thread Safety Thread safety usually entails ensuring that a particular instance of a function either only uses local resources or has exclusive use of global resources. Local resources generally mean automatic variables that are created for each instance of the function's invocation. If multiple threads share read and write access to a common resource, inconsistencies can occur. For example, if thread θ uses the value of a shared state variable α to choose a course of action that ultimately ends up changing the value of α, θ must have exclusive access to α for the both the read and the writeback. Without exclusive access, θ can neither be sure that it is the only thread that will choose that particular course of action, nor know that the value of α that it read was valid (if some other thread was just about to update α after θ read it).[18,19]

Shared resources (whether local or global) can be used, provided that "critical" sections of the code are protected (serialized) with mutual exclusion mechanisms. The PIPT provides wrappers to the operating system mutual exclusion locking system, but care must be taken in designing and using mutual exclusion algorithms. That is, one must protect those sections of the code that need to be protected, but at the same time, use of mutual exclusion serializes protected sections of code, so such protection should be kept to an absolute minimum.

PIPT is designed and implemented to be thread safe. However, users adding their own routines into the PIPT must (at times) ensure the thread safety of their own routines. Since a PIPT worker may be running as a multi-threaded process on an SMP, a PIPT computational kernel may spawn multiple threads to call a user-defined operator for processing a single input slice. The user-defined operator routine must be thread safe to ensure deterministic results of the output slice. For example, if a global (rather than local) floating point variable `total` were used by `IPAverage_Window_operator()` to maintain the running total of pixels, several problems could occur. First, each thread would attempt to add its own input pixel values to `total`, irrespective of what the other threads were contributing. Second, each thread θ would add its current pixel value to the last value that it read from `total`, which may not be the current value if another thread wrote a new value between the time that θ read `total` and wrote back its new value.

Problems With Multithreading Most current implementations of MPI are not thread safe; they tend to fail or act nondeterministicly when used with multithreaded message passing code. Having the PIPT depend on multithreaded message passing code in the PIPT would severly limit its portability. As such, two schemes were not implemented into the PIPT at this time: spawning a worker thread on the manager and sending asynchronous signals to the workers.

Spawning a worker thread on the manager (as discussed in Section 3.1) would prevent idle cycles on the manager while it blocks waiting for a worker to return a slice. This scheme requires a thread safe MPI because the manager and workers rely on MPI for all message passing functionality. While the PIPT message passing functions could be rewritten to use different methods of sending to workers (depending on whether the worker is on the same node or on a remote node), this would involve a significant redesign of the PIPT internal communication routines. A workaround for this problem that will be used until more

thread safe MPI implementations become available is to use the MPI framework to launch a separate worker process on the manager node. The underlying MPI will presumably use shared memory to communicate on the same node, and not costly loopback socket calls.

Asynchronous signals are included in the Redundant First-Finish, First-Serve (RFFFS) load balancing algorithm that is described in Section 4.2.3. This scheme requires multithreaded message passing code because one thread needs to block on an incoming message while another thread processes the slice. Unless the MPI library is reentrant, when the processing thread tries to send its slice back to the manager with an MPI send call, internal accounting information becomes corrupted and the program fails.

4.2. Load Balancing

The goal of load balancing is to partition the parallel task in such a way that each processor will work for approximately the same amount of time, but not necessarily perform the same amount of work. The PIPT's approach to load balancing on an active heterogeneous workstation cluster is a compromise between dividing up the work into many small pieces and actively monitoring the available processing power. The work is divided into more pieces than nodes are available, and available processing power is passively monitored by distributing (possibly redundantly) slices of work on a first-come, first-serve basis.

The PIPT provides three different load balancing algorithms: no load balancing, simple first-finish, first-serve (FFFS), and redundant first-finish, first-serve (RFFFS).

4.2.1. No load balancing

With no load balancing, the PIPT gives equally sized slices to each of the workers. This approach is suitable for dedicated parallel machines or unloaded homogeneous workstation clusters, since every node will process the same amount of work in the same amount of time. This approach also has the least communication overhead since only one slice of work is sent to each worker.

4.2.2. First-Finish, First-Serve

The FFFS load balancing algorithm divides the input image into more slices than there are nodes available. Each node is initially given a single slice of the image to process. The faster (or less loaded) nodes will naturally finish their section quickly, return it to the manager, and request more work. In this manner, a fast worker may process multiple slices in the time that it takes a slow (or loaded) worker to process one slice. Since the run is limited by the slowest node, the FFFS algorithm gives more work to faster nodes and less work to slower nodes.

4.2.3. Redundant First-Finish, First-Serve

FFFS can still end up waiting for a slow node at the end of the computation — in some cases, a slow node can be assigned another section of work just before all the other sections are completed. The entire job must then wait for the slow node to finish its slice, even though the other nodes are idle.

Redundant FFFS is an extension of the original FFFS algorithm. It exploits a special case in the FFFS algorithm that occurs when a worker node returns a slice and all remaining work sections have already been distributed to other workers. In this situation, FFFS lets the worker remain idle until the end of the run. This is wasteful if fast nodes return their slices while a slow nodes are still processing their slices; the fast nodes sit idle waiting for the slow nodes to finish.

Instead of letting the fast worker node sit idle, RFFFS gives the workers another slice of work. The section of work is already being processed by some other worker node (since all remaining work sections have already been distributed). That is, multiple worker nodes are processing the same slice. In this way, fast worker nodes can be assigned the same work that a slow worker is processing. The fast workers will finish the section first and return it to the manager, thereby ending the computation (without having to wait for the slow node to finish). Pseudocode for the RFFFS algorithm is shown in Figure 3.

The receipt of the abort message requires asynchronous message passing and requires multithreading message passing code in the workers. That is, the worker must simultaneously block on the potential receipt of an abort message (via MPI) and process its slice at the same time. Unfortunately, there are no MPI implementations for workstation clusters that are thread safe, so blocking on the receipt of a message while processing a slice was not possible. As such, the abort signal was not implemented into the current version of the PIPT.

Instead of aborting the worker nodes, the manager places the redundant workers in a “working” state. When the entire input image has been processed, the manager returns the output to the calling function, regardless of whether the redundant nodes

```
Divide input image into num_slices slices
Send each of the N workers a slice
Mark each worker as "working"
Mark first N slices as "sent"; mark remaining slices as "not done"

While there are more slices to process
    Receive "done" message from worker x
    If worker marked as "working"
        Receive output slice from worker x
        Mark slice as "done"
        Mark worker as "idle"
        If other workers redundantly processing the same slice
            Send "abort" messages

    If there are more slices that are not being processed
        Send next slice
        Mark worker x as "working"
        Mark slice as "sent"
    Else if there are more slices that have not been returned yet
        Find a "sent" slice, send to worker x
        Mark worker x as "working"

Send abort messages to workers marked as "working"
```

Figure 3. Redundant First-Finish, First-Serve load balancing algorithm

have completed their sections or not. The redundant nodes are reclaimed later; when a redundant node finally attempts to return its processed (but now outdated) slice, the manager ignores the slice and invites the worker to join the current computation. These complexities only required because MPI implementations are not thread safe, and are not shown in Figure 3.

4.2.4. Communication Costs

Sending and receiving the image slices across an interconnection network incurs some delay which contributes to the overall wall-clock execution time. Two factors which contribute to network overhead are latency and bandwidth. While many workstation networks operate at 10Mbps, faster speeds, such as 100Mbps and 155Mbps, are becoming more common. Naturally, since the PIPT sends large messages across the network, a fast underlying transport will reduce the amount of overhead incurred while processing images in parallel.

Latency Figure 4 illustrates the costs of sending messages using MPI in a workstation cluster environment. Messages of sizes from 1 to 1,048,576 bytes (1 Mbyte) were sent from manager to worker on 10Mbps and 100Mbps switched ethernet networks, as well as a 155Mbps ATM network. Note that the cost for the smaller messages is dominated by latency time (approximately 0.5 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.

Network Speed The speed of the underlying communications network can also significantly affect the wall clock execution time. Scattering and gathering an 8MB image file to the workers across a 10Mbps ethernet (assuming ideal conditions) takes over 13 seconds. That is, regardless of how fast the nodes can process the image, the lower bound for the wall clock running time is roughly 13 seconds. In contrast, the combined scatter and gather time for 100Mbps and 155Mbps networks (assuming ideal conditions) is 1.3 second and 0.86 seconds, respectively.

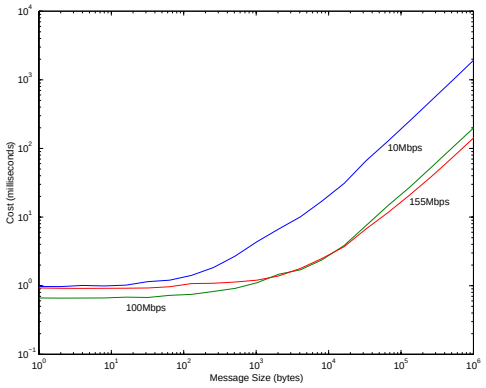

Figure 4. Time required to send messages across networks with different bandwidths.

5. 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 a single plane 1455×1540 TIFF image. The experiments were conducted using a network of dedicated dual processor Sun UltraSPARC/2 200 workstations connected with switched 100 Mbit/s Ethernet. Each workstation ran one worker which spawned two threads to perform the work (except the single processor case); no load balancing was used. The LAM implementation of MPI from the Ohio Supercomputing Center was used for all results.[20]

5.1. Parallel Performance

Figure 5 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 pixels. To quantify parallel

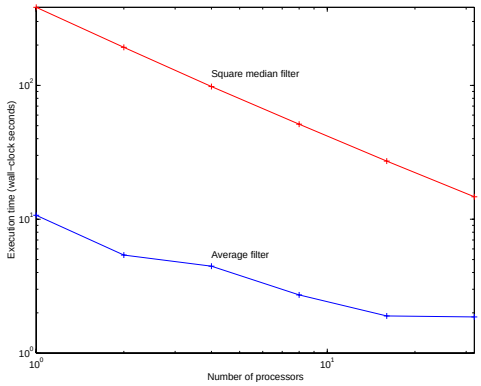

Figure 5. Execution times (in wall-clock seconds) for parallel squared median filter and average filter.

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. Table 1 lists the speedups of both filters for each number of processors. The speedup for the parallel average filter does not scale linearly with the number of processors after 2 nodes, indicating that the communication costs, conspiring with Amdahl's Law, quickly dwarf the computation costs. The parallel square median filter requires substantially more computation relative to communication than

Number of processors	2	4	8	16	32
Average filter	1.983	2.400	3.936	5.647	5.743
Square median filter	1.990	3.907	7.468	14.081	26.018

Table 1. Speedups.

the parallel average filter. Its speedups vary almost linearly with the number of processors until 32 processors are used, when the both the communication costs and synchronization costs of the manager/worker paradigm outweigh the computation time.

5.2. Load Balancing

The PIPT works well without load balancing on dedicated parallel hardware; there is no need for the additional overhead of load balancing when all the compute nodes are equivalent in compute power and dedicated to the PIPT job. But in an active heterogeneous workstation cluster, this algorithm performs poorly. Figure 6 shows a timing plot of a run with no load balancing on an active workstation cluster. The horizontal lines represent the workers; the top line is the manager, the bottom four lines are the workers. Vertical lines represent messages between processors. Light areas indicate computations contributing to that worker's slice; dark areas indicate that the node is idle.

Figure 6. A PIPT run with no load balancing on an active workstation cluster. Workers 2 and 3 are heavily loaded, and delay the completion of the computation.

It is clear from Figure 6 that in an active workstation cluster, run times can be very large without load balancing. Figure 7 shows the PIPT using FFFS on the same workstation cluster; the overall run time is 86% shorter than without load balancing.

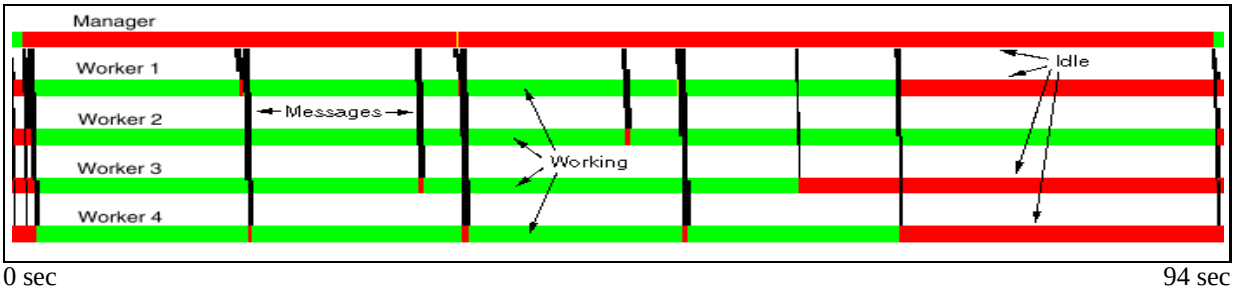

Figure 7. A PIPT run using FFFS load balancing on an active workstation cluster. Workers 2 and 3 are heavily loaded, but workers 1 and 4 compensate by processing more slices than workers 2 and 3. But Workers 1, 2, and 3 sit idle while waiting for Worker 2's final slice.

The FFFS algorithm shows a marked improvement in performance over no load balancing, but worker 2 still delays the end of the computation. Figure 8 shows the PIPT using the RFFFS algorithm on the same workstation cluster. Workers 1 and 4

both perform more work than workers 2 and 3, but also redundantly process the same slices as workers 2 and 3, resulting in a shorter run time for the overall computation.

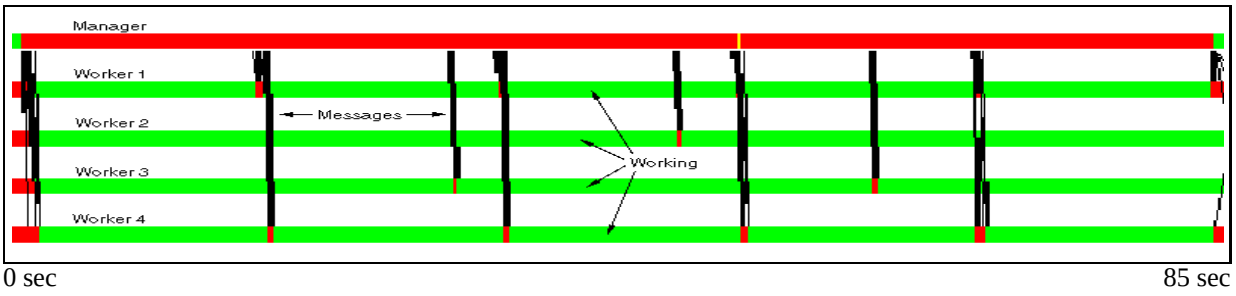

Figure 8. A PIPT run using RFFFS load balancing on an active workstation cluster. Workers 2 and 3 are heavily loaded, but workers 1 and 4 compensate by performing more work than workers 2 and 3, and then redundantly processing the final slices of workers 2 and 3. Notice that workers 2 and 3 keep computing even after the manager finishes the computation, since asynchronous abort messages were not possible without thread safe MPI implementations.

6. 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 on functions having sufficient opportunity for parallel speedup as a function of the number of processors 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.

Cluster-based computing is one important, but under appreciated, form of parallel computing. 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. Through its load balancing schemes, the PIPT adapts to run efficiently on active heterogeneous clusters.

6.1. Further Extensions to PIPT

Continuing research in image and video processing will require changes in several of the internal PIPT models to afford optimizations that are not possible with the current implementation. One example is changing the internal framework to allow workers to receive multiple slices simultaneously, based upon processing power, availability, etc. These changes naturally fit within an object-oriented design, and would most easily be implemented by converting the PIPT to C++. The C++ language has significantly more expressive power than C, offers better data encapsulation, and allows for compile-time binding of image processing functions (vs. run-time binding with function pointers) through inlining and templates.

Better data and functionality encapsulation will enable natural associations between data objects and the methods that operate on them; image processing routines and kernels will likely become objects with associated data members containing all necessary internal state, for example. Stronger type checking in C++ also allows for registration of parameters without additional type identifies that are necessary in C. Compile-time binding of functions will allow for much greater compiler optimization of image processing loops; short, inlined window operator functions can be highly optimized (unrolled, blocked, etc.) inside kernel loops, whereas compilers cannot optimize a window operator function call as the inner statement of a loop.

Parallel input/output systems will be investigated with the recent implementation of the MPI-IO library from the Argonne National Laboratory. Thread safety issues will also continue to be explored; in particular cases, it is possible to utilize non-thread-safe libraries in a multithreaded environment. Finally, the possibility for asynchronous one-sided operations will be researched as implementations of the one-sided MPI functions become available.

6.2. Availability

A public domain implementation of the PIPT is available on the World Wide Web. Version 2.1 can be found at:

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

7. ACKNOWLEDGMENTS

This effort was sponsored by Rome Laboratory, Air Force Materiel Command, USAF under grant number AF-F30602-96-C-0235. 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.

The current release of PIPT has benefited from the contributions of many people. Brian McCandless contributed heavily to the initial development of the PIPT, and prototyped the multithreaded workers. John Tran assisted in the development of the PIPT and in testing the software. M. D. McNally spent countless hours developing a comprehensive test suite for the final product, as well as assisting in debugging, packaging, and documenting the current version, 2.1.

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* **37**(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* **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* **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* **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* **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. J. Bilmes, K. Asanovic, J. Demmel, and C.-W. Chin, "Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology," in *International Conference on Supercomputing*, (Vienna, Austria), July 1997.
15. J. L. Hennessy and D. A. Patterson, eds., *Computer Architecture - A Quatitative Approach*, Morgan Kaufmann Publishers, Inc, 1996.
16. B. C. McCandless, J. J. Tran, J. M. Squyres, A. Lumsdaine, and R. L. Stevenson, *Software Documentation Parallel and Distributed Algorithms for High-Speed Image Processing*. Laboratory for Scientific Computing, University of Notre Dame, Notre Dame, IN, 1995.
17. J. M. Squyres, "MPI: Extensions and applications," Master's thesis, University of Notre Dame, Notre Dame, IN, 1996.
18. A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, Inc., Upper Saddle River, NJ, 1995.
19. A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, Inc., Upper Saddle River, NJ, 1992.
20. G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Proceedings of Supercomputing Symposium '94*, J. W. Ross, ed., pp. 379–386, University of Toronto, 1994.
21. 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.