Transparent Parallel Image Processing by way of a Familiar Sequential API

F.J. Seinstra and D. Koelma
Intelligent Sensory Information Systems
Faculty of Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
(fjseins, koelma)@science.uva.nl

Abstract

This paper describes an infrastructure that enables transparent development of image processing software for parallel computers. The infrastructure's main component is an image processing library containing operations capable of running on distributed memory MIMD-style parallel hardware. Since the library has a programming interface identical to that of a familiar sequential image processing library, the parallelism is completely hidden from the user. All library functions are based on an abstract parallel image processing machine (APIPM), introduced in this paper. To guide the process of automatic parallelization and optimization, a performance model is defined for operations implemented using APIPM instructions. Experiments show that for realistic image operations performance predictions are highly accurate. These results suggest that the infrastructure's core forms a powerful basis for automatic parallelization and optimization of complete applications.

1. Introduction

For many years it has been recognized that the application of parallelism in low level image processing can be highly beneficial. Consequently, references to optimal parallel algorithms [9] and dedicated parallel architectures [3] abound in the literature. In spite of this, the gap between the areas of image processing and high performance computing has remained large. This is mainly due to the fact that image processing researchers consider most parallel solutions 'too cumbersome' to apply. Rather than blaming the image processing community, researchers in high performance computing should provide tools that allow the development of software in a way that is highly familiar to image processing researchers. In addition, such tools should be applicable to commonly available general purpose parallel computers.

The ideal solution is a fully automatic parallelizing compiler. Unfortunately, the fundamental problem of automatic and optimal *partitioning* remains unsolved. Another possibility is the design of a parallel programming language, either general purpose [14] or aimed at image processing specifically [1]. However, in accordance with the remarks made in [6], we feel that a parallel language is not the preferred solution. Even a few simple language annotations are often considered cumbersome, and thus should be avoided.

In this research we aim at the creation of a *software library* containing a set of abstract data types and associated pixel level operations executing in (data) parallel fashion. All parallelization and optimization is to be hidden from the user by a programming interface identical to that of a familiar *sequential* image processing library. The routines in the library should be capable of running on general purpose distributed memory MIMD-style parallel hardware.

In this paper we describe a software infrastructure that, apart from the library itself, consists of several components dealing with the aspects of automatic parallelization, portability, optimizations across sequences of library calls, etcetera. We focus on the definition of an abstract parallel image processing machine (APIPM), and the modeling of the performance of routines implemented using APIPM instructions. Experiments show that the instruction set, and the performance models based on it, together form a solid basis for automatic parallelization and optimization.

In related work [12] the need for parallel libraries for low level image processing is stressed as well. However, performance models for optimization across library calls are often not incorporated [4]. Also, definition of an abstract parallel image processing machine has been considered rarely. The only references found (see [2] and later work by the same authors) describe the successful implementation of a portable parallel abstract machine. This machine, however, is defined at a lower level of abstraction than our APIPM.

This paper is organized as follows. Section 2 gives an infrastructure overview. Section 3 introduces the APIPM and its instruction set. In Section 4 a performance model is defined for routines implemented using APIPM instructions only. In Section 5 model predictions are compared with results obtained on a real machine that fits within the APIPM model. Concluding remarks are given in Section 6.

2. Infrastructure

The complete infrastructure consists of six logical components (see Figure 1). The first component (C1) is a sequential library that contains several classes of low level image processing routines typically used by image processing researchers. Currently we have adopted an existing library (Horus, see [5]), but in principle the results of this research should be applicable to comparable libraries as well.

The sequential library is extended with several routines that introduce the parallelism into the system (C2). The routines (for the scattering, gathering, broadcasting, and redistribution of image data in a parallel system) are 'kernel' routines, not available in the programming interface. In the resulting extended library all sequential image processing operations are *fully separated* from the additional parallel routines. As separate image data distribution operations are an integrated part of the library, the creation of parallel versions of sequential image processing routines becomes a manageable task. This strategy may result in a loss of efficiency (albeit marginal), but programmability is greatly enhanced.

Each class of operations in the library is annotated with a performance model (C3) for execution time prediction. The models are essential for the process of automatic parallelization and optimization of complete applications. In addition, the models are useful as an indication to users or image library creators as well.

Components C1-C3 constitute a complete library with sequential and parallel capabilities. The components are sufficient for writing parallel image processing applications by hand. To fully hide the parallelism from the image processing application programmer, whilst ensuring efficiency of execution, three additional components are required.

First, a benchmarking tool (C4) is needed to obtain values for the model parameters. The process of benchmarking is guided by the operations that form the APIPM instruction set.

For a complete program the system needs to know what library routines are being applied, and the order of application. This information is available from the program code. Rather than implementing a complete parser, a simple tool is implemented to allow users to specify their algorithms (C5).

Once the performance models, the benchmarking results, and the algorithm specification are available a scheduler (C6) is required to find an optimal parallel solution for the application at hand. Whether the scheduling results are static or generated and updated dynamically is still an important future research issue.

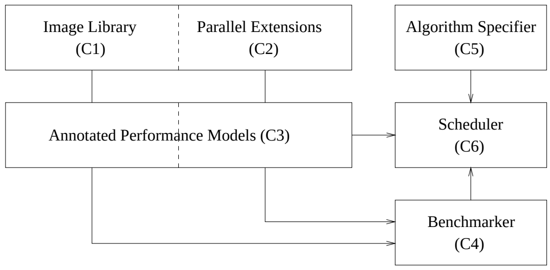

Figure 1. Infrastructure overview.

3. Abstract Parallel Image Processing Machine

In this research, all library implementations and performance models are based on the definition of an abstract parallel image processing machine (or APIPM, see Figure 2). An APIPM consists of one or more abstract sequential image processing machines (ASIPMs), each consisting of four related components: (1) a *sequential image processing unit*, capable of executing APIPM instructions, one at a time, (2) a *memory unit*, capable of storing (image) data, (3) an *I/O unit*, for transporting data from the memory unit to external sensing or storage devices, and (4) *data channels*, the means by which data is transported between the ASIPM units and external devices. In a complete APIPM the memory unit of each ASIPM is connected with those of all other ASIPMs.

Note that this definition of the APIPM reflects a state-of-the-art distributed memory MIMD-style parallel machine. It only differs from a general purpose machine in that each sequential unit is designed for image processing related tasks only. Although a fully connected communication network is often not present, we still have included one in our APIPM. This is because in most multicomputer systems communication is based on circuit-switched message routing, which makes a network *virtually* fully connected.

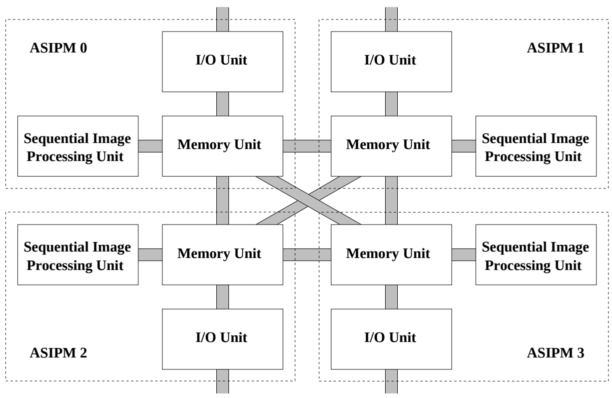

Figure 2. APIPM comprising of 4 ASIPMs.

opcode	memory instructions
CREATE	allocate data block in memory unit
MEMCOPY	copy data in memory unit
DELETE	free up data block in memory unit
opcode	**image processing instructions**
UPOP	unary pixel operation (no argument)
UPOPC	unary pixel operation (constant value as argument)
UPOPI	unary pixel operation (complete image as argument)
BPOPC	binary pixel operation (constant value as argument)
BPOPI	binary pixel operation (complete image as argument)
REDUCOP	global reduce operation
NEIGHOP	generalized neighborhood operation
TEMPLOP	generalized template operation
GEOMAT	geometric transformation (matrix as argument)
GEOROI	geometric transformation (region of interest)
opcode	**I/O instructions**
IMPORT	import data from external device
EXPORT	export data to external device
opcode	**communication instructions**
SEND	send data to other ASIPM
RECV	receive data from other ASIPM

Figure 3. Simplified APIPM instruction set.

The APIPM instruction set (see Figure 3) consists of three classes of sequential operations: (1) memory operations, for example for the creation of scratch images, or the copying of border data, (2) *fundamental* image operations, for the processing of image data without (internal) creation of additional data structures, and (3) I/O routines, for transporting data to external devices, and vice versa. The parallelism is introduced by an additional class of routines containing *communication* operations. These routines enable the exchange of data among ASIPM units. For reasons of simplicity, in the overview of Figure 3 the operands for each opcode have been left out.

A substantial part of all sequential and parallel low level image processing operations can be implemented using operations from the APIPM instruction set only. These include all image processing operations that can be written as a non-recursive neighborhood operation, or as a geometric transformation (for more information, see [7]). In the (near) future we expect the APIPM instruction set to be extended to incorporate other operations as well. In addition, it is important to note that the programming interface for each parallel image processing operation in our library (that has been implemented using APIPM instructions) has remained identical to its sequential counterpart.

4. Annotated Performance Models

To achieve automatic parallelization and optimization of complete image processing applications, the modeling of the performance of library operations is an essential part of our infrastructure. Based on the APIPM, we have designed a performance model that is *simple*, *accurate*, and *portable* to machines that fit within the APIPM model. In our model we assume that the performance of each library operation can be partitioned into *independent* time intervals, each corresponding to the execution time of a single APIPM instruction. The overall performance of a library operation is obtained by simply adding the execution times of all internal APIPM instructions.

This is formalized as follows. Let $\mathbf{I} = \{I_1, I_2, \cdots, I_n\}$ be the APIPM instruction set. Let $\mathbf{P} = \{P_{I_1}, P_{I_2}, \cdots, P_{I_n}\}$ be the set of performance values for all instructions in $\mathbf{I}$. We assume that, for any system capable of running APIPM instructions, and for each instruction in $\mathbf{I}$, P_{I_i} can be obtained by benchmarking. Also, let $\mathbf{L} = \{\mathbf{L}_1, \mathbf{L}_2, \cdots, \mathbf{L}_m\}$ be the set of all m routines implemented using instructions in $\mathbf{I}$ only. For all library operations $\mathbf{L}_x$ ($x \in \{1, \cdots, m\}$) we define $\mathbf{L}_x = \{I_1, I_2, \cdots, I_n\}$, in combination with the number of occurrences (or *count*) of each APIPM instruction in $\mathbf{L}_x$: $\mathbf{C}_x = \{C_{I_1,x}, C_{I_2,x}, \cdots, C_{I_n,x}\}$. The count of each instruction can have any value in $\mathbb{N}$ (including 0). The expected total execution time of operation $\mathbf{L}_x$ is obtained by

$$T_{\mathbf{L}_x} = \sum_{i=1}^{n} C_{I_i,x} P_{I_i} \tag{1}$$

A similar model was defined in [8], but used for rather different purposes: general machine characterization based on an Abstract Fortran Machine.

A problem with the simplistic model formalized above is that most APIPM instructions (and related performance values) are not single static entities. This is because the execution of an instruction is often dependent on its operands. Therefore, a static entity for each possible operand combination must be incorporated in our model. To avoid an explosion of static entities we allow each instruction I_i and each value P_{I_i} to be *parameterized*. For example, a 'datatype' parameter is incorporated in almost all instructions (e.g. giving $I_i('int')$ and $I_i('float')$). Also, a 'data-input-size' parameter is required for most performance values in $\mathbf{P}$ (e.g. giving $P_{I_i(datatype)}(size)$).

As it is our goal to include no knowledge of underlying hardware, no assumptions can be made about performance growth rates in relation to data input size. To avoid benchmarking for each possible input size, we assume a linear growth rate in the region around each measured size. For each measured input size a 'per data item' value can be obtained by dividing the measured value by the input size. The performance of an operation, for each size s_{in} within the region around the measured size s_m, is then given by

$$P_{I_i(datatype)}(s_{in}) = s_{in} \frac{P_{I_i(datatype)}(s_m)}{s_m}$$

For all communication operations we have created more sophisticated models, based on the 'three paths of commu-

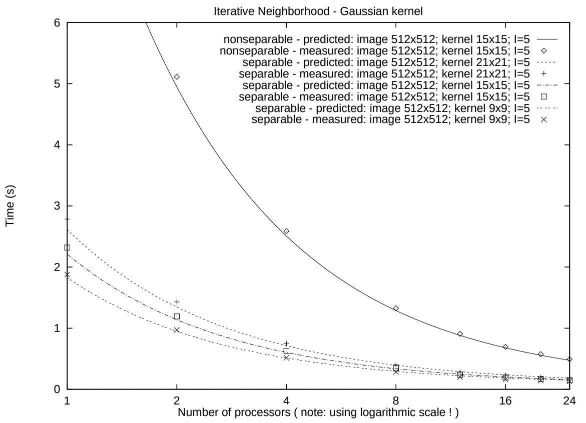

Figure 4. Measured and predicted times for an iterative (5x) Gaussian kernel operation (separable & non-separable) on a 512^2 image.

nication' defined in this project. Discussion, however, is beyond the scope of this paper (see also [10, 11]).

5. Performance Measurements and Validation

To validate the performance models a number of experiments were performed on the 24-node Distributed ASCI Supercomputer (DAS [13]) located at the University of Amsterdam. All nodes contain a 200 Mhz Pentium Pro with 64 Mbyte of DRAM and are connected by a 1.2 Gbit/sec full-duplex Myrinet SAN network. At time of measurement, the clusters ran RedHat Linux 2.0.36.

By running simple benchmarking operations we have obtained values for the performance parameters in our models. Based on these values we have estimated the execution times of many library operations. Unfortunately, space limits the discussion to only one operation: the differential structure of images. In the measurements presented here we have limited ourselves to the first and second order derivatives in the x- and y-direction (five in total).

As is well known, there are many reasons to compute a derivative using convolution with a Gaussian kernel. To avoid round-off errors, the size of the convolution mask should depend on the scale σ and the order of the derivative. A conservative estimate for the average kernel size using only first and second order derivatives is 15x15 pixels. Performance values, obtained by computing the five derivatives in sequence on a 512x512 image, using both separated and nonseparated kernels of various sizes, are depicted in Figure 4. The results are presented only for a row-wise distribution of the image data. As can be seen, on any number of processors, the predicted execution times for all experiments are highly accurate.

6. Conclusions

In this paper we have described an infrastructure that allows an image processing researcher to develop parallel applications transparently. This is made possible by a parallel image processing library, with a programming interface identical to that of a sequential library, that completely hides the parallelism from the user. We have discussed the abstract parallel image processing machine (APIPM), and the performance models based on it. Experiments performed on a machine that fits within the APIPM model show that our performance models are highly accurate. These results strongly suggest that the core of our infrastructure forms a powerful basis for automatic parallelization and optimization of complete image processing applications.

References

[1] J. Brown and D. Crookes. A High Level Language for Parallel Image Processing. *Image and Vision Computing*, 12(2):67–79, Mar. 1994.

[2] D. Crookes and P. Morrow. Design Considerations for a Portable Parallel Abstract Machine for Low Level Image Processing. In *BCS Workshop on Abstract Machine Models for Highly Parallel Computers*, pages 107–110, Mar. 1991.

[3] D. Hammerstrom and D. Lulich. Image Processing Using One-Dimensional Processor Arrays. *Proceedings of IEEE*, 84(7):1005–1018, 1996.

[4] L. Jamieson et al. A Software Environment for Parallel Computer Vision. *IEEE Computer*, 25(2):73–75, Feb. 1992.

[5] D. Koelma, E. Poll, and F. Seinstra. Horus (Release 0.9.2). Technical report, University of Amsterdam, Feb. 2000.

[6] C. Pancake and D. Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers. *IEEE Computer*, 23(12):13–23, Dec. 1990.

[7] G. Ritter and J. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Inc, 1996.

[8] R. Saavedra-Barrera et al. Machine Characterization Based on an Abstract High-Level Language Machine. *IEEE Transactions on Computers*, 38(12):1659–1679, Dec. 1989.

[9] A. Saoudi and M. Nivat. Optimal Parallel Algorithms for Multidimensional Template Matching and Pattern Matching. In *Proceedings ICPIA'92*, pages 240–246, Dec. 1992.

[10] F. Seinstra and D. Koelma. Modeling Performance of Low Level Image Processing Routines on MIMD Computers. In *Proceedings of ASCI'99*, pages 307–314, June 1999.

[11] F. Seinstra and D. Koelma. Accurate Performance Models of Parallel Low Level Image Processing Operations Based On a Simple Abstract Machine. Technical report, University of Amsterdam, Apr. 2000.

[12] R. Taniguchi et al. Software Platform for Parallel Image Processing and Computer Vision. In *Parallel and Distr. Methods for Image Processing*, volume 3166, pages 2–10, 1997.

[13] The Distributed ASCI Supercomputer. Information available at: http://www.cs.vu.nl/das/.

[14] G. Wilson and P. Lu. *Parallel Programming Using C++*. Scientific and Engineering Comp. Series. MIT Press, 1996.