

## A Software Architecture for User Transparent Parallel Image Processing

**F.J. Seinstra, D. Koelma and J.M. Geusebroek**

Intelligent Sensory Information Systems  
Department of Computer Science  
University of Amsterdam  
The Netherlands

This report describes a software architecture that enables transparent development of image processing applications for parallel computers. The architecture's main component is an extensive library of low level image processing operations capable of running on distributed memory MIMD-style parallel hardware. Since the library has an application programming interface identical to that of an existing sequential image processing library, the parallelism is completely hidden from the user.

The first part of the report discusses implementation aspects of the parallel library. It is shown how sequential as well as parallel operations are implemented on the basis of so-called *parallelizable patterns*. A library built in this manner is easily maintainable, as code redundancy is avoided as much as possible. The second part of the report describes the application of performance models to ensure efficiency of execution on a range of parallel machines. A high level abstract machine for parallel image processing, that serves as a basis for the performance models, is described as well. Experiments show that for a realistic image processing application performance predictions are highly accurate. These results indicate that the core of the architecture forms a powerful basis for automatic parallelization and optimization of a wide range of image processing software.

**Keywords:** parallel image processing library, parallelizable patterns, performance modeling, abstract parallel image processing machine, general purpose MIMD-style parallel computers.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Architecture for Parallel Low Level Image Processing</b>	<b>3</b>
2.1	Library Requirements . . . . .	3
2.2	Architecture Overview . . . . .	4
2.2.1	Sequential Image Processing Operations . . . . .	4
2.2.2	Parallel Extensions . . . . .	5
2.2.3	Parallel Image Processing Operations . . . . .	6
2.2.4	Single Uniform API . . . . .	6
2.2.5	Annotated Performance Models . . . . .	7
2.2.6	Benchmarking Tool . . . . .	7
2.2.7	Algorithm Specification . . . . .	7
2.2.8	Scheduling Tool . . . . .	7
<b>3</b>	<b>Parallelizable Patterns in Low Level Image Processing</b>	<b>8</b>
3.1	Representation of Digital Images . . . . .	8
3.2	Parallelizable Patterns . . . . .	9
3.2.1	Parallelizable Patterns: The General Case . . . . .	9
3.2.2	Parallelizable Patterns: General Parallelization Strategy . . . . .	11
3.2.3	Example: Parallel Reduction . . . . .	12
3.2.4	Example: Parallel Generalized Convolution . . . . .	13
3.3	Discussion . . . . .	14
<b>4</b>	<b>Performance Models</b>	<b>15</b>
4.1	General Performance Model Requirements . . . . .	15
4.2	Performance Models Based on a High Level Abstract Machine . . . . .	16
4.2.1	Abstract Parallel Image Processing Machine: Components . . . . .	16
4.2.2	Abstract Parallel Image Processing Machine: Instruction Set . . . . .	17
4.3	APIPM-based Performance Models . . . . .	19
4.3.1	Model-Related Benchmarking . . . . .	20
4.4	Extended Model for Point-to-Point Communication . . . . .	20
4.5	Discussion . . . . .	22
<b>5</b>	<b>Performance Measurements and Validation</b>	<b>22</b>
5.1	Detection of Lines in Images . . . . .	23
5.2	Sequential Implementation . . . . .	24
5.3	Parallel Execution . . . . .	25
5.4	Performance Evaluation . . . . .	27

---

<b>6 Conclusions and Future Work</b>	<b>30</b>
--------------------------------------	-----------

---

**Intelligent Sensory Information Systems**

Department of Computer Science  
University of Amsterdam  
Kruislaan 403  
1098 SJ Amsterdam  
The Netherlands

tel: +31 20 525 7463

fax: +31 20 525 7490

<http://www.science.uva.nl/research/isis>

**Corresponding author:**

F.J. Seinstra

tel: +31(20)525 7553

[fjseins@science.uva.nl](mailto:fjseins@science.uva.nl)

<http://www.science.uva.nl/~fjseins>

## 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 [5, 19] and dedicated parallel architectures [10, 15] abound in the literature. In spite of this, the gap between the areas of image processing and high performance computing has remained large. Essentially, this is due to the fact that the image processing community considers most parallel solutions 'too cumbersome' to apply. As it is unrealistic to expect image processing researchers to be experts in parallel computing, tools must be provided to allow them to develop high performance applications in a highly familiar manner.

The ideal solution is to provide a fully automatic parallelizing compiler for a sequential programming language commonly used by image processing researchers (i.e., C or C++). Unfortunately, the fundamental problem of automatic and optimal partitioning remains unsolved. Another possibility is to design a parallel programming language, either general purpose [25] or aimed at image processing specifically [3]. However, in accordance with the remarks made in [16], we feel that a parallel language is not the preferred solution. Even the use of a relatively small number of language annotations is often considered cumbersome, and thus should be avoided.

An interesting and more practical approach is to design a software library containing parallel versions of operations commonly used in image processing research. Due to the relative ease of implementation, many such libraries have been described in the literature (for example, see [11, 12, 23]). An important design goal in much of this research is to provide library operations that have optimal efficiency on a range of parallel machines. In many cases this is achieved by hard-coding a number of different implementations, one for each platform. We feel that this solution to *intra-operation optimization* requires too much implementation effort, is not flexible enough, and is impossible to maintain on the long term. Also, due to the intrinsic difficulty, the important aspect of *inter-operation optimization* (or optimization across library calls) is often not dealt with. For these reasons, we take a different approach.

In our research we aim at creating an architecture for parallel low level image processing that brings the benefits of high-performance computing to the image processing community in a transparent manner (i.e., hidden from the user). The core of the architecture is a library containing a set of abstract data types and associated pixel level operations executing in data parallel fashion. The most distinctive aspect of our work is that in implementing the library we take a minimalistic approach. Essentially, this means that we strive to maximize operation reusability and avoid code redundancy as much as possible. Apart from being relatively simple to implement, a parallel library built in this manner is extensible, easily maintainable, and still high in performance.

In this report we give an overview of the complete software architecture created in this research, and highlight some of the more important aspects. Section 2 shortly introduces all architecture components, and discusses their relationships. Section 3 gives implementation details of the parallel library, and introduces the concept of so-

called *parallelizable patterns*. Section 4 discusses the aspect of obtaining efficiency of execution on a range of parallel machines. It is shown how the problem is attacked by the application of performance models. Also, a description is given of a high-level abstract parallel image processing machine (APIPM), that serves as a basis for all performance modeling. In Section 5 model predictions are compared with results obtained on a real machine from the class of machines under consideration. Concluding remarks are given in Section 6.

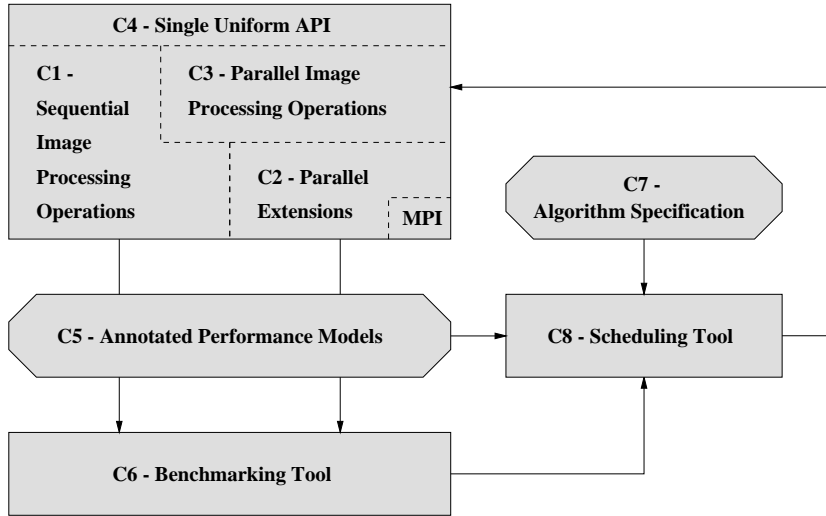
## 2 Architecture for Parallel Low Level Image Processing

In this section we introduce the requirements put forward in this research for the development of the parallel image processing library that forms the core of our architecture. This discussion is followed by an overview of the complete set of architecture components.

### 2.1 Library Requirements

In our research the intention is to build a parallel image processing library that adheres to the following requirements:

1. *Low threshold*. To ensure that the library is of great value to the image processing community, care should be taken to ensure that its application fits the user's frame of reference. For this reason, the library must be implemented such that no need arises for the image processing researcher to obtain additional skills related to the parallelism.
2. *Maintainability*. To ensure longevity, the library must be extensible and easily maintainable. Therefore, care must be taken in the implementation of the library to avoid unnecessary code redundancy, and to enhance operation reusability.
3. *Availability*. For practical and economic reasons, the library must be applicable to commonly available parallel computers. Essentially, this restricts the class of target platforms to that of general purpose distributed memory MIMD-style parallel machines. Although a higher efficiency is often obtained on dedicated hardware, the relatively low cost and high flexibility has caused general purpose machines to be more generally available, and are therefore preferred.
4. *Portability*. Implementation in C/C++ in combination with MPI [8] is most appropriate to ensure portability to a wide range of machines. In the implementations no assumptions should be made about a specific interconnection network topology. All processing units within the system can be assumed identical, however, and each communication line can be assumed to be as fast as any other.
5. *Efficiency*. Despite the requirement of library maintainability, efficiency of execution must be ensured on all machines in the class of target platforms.



**Figure 1:** *Architecture overview.*

Efficiency, in this respect, refers to each separate library operation, as well as to multiple operations applied in sequence.

## 2.2 Architecture Overview

The complete architecture created in this research consists of eight logical components (see Figure 1). In this section each component is described in short, and design choices are related to the aforementioned requirements.

### 2.2.1 Sequential Image Processing Operations

The first component (C1) contains a large set of sequential operations typically used by image processing researchers. As recognized in, for example, Image Algebra [17], a small set of *operation classes* can be identified that covers the bulk of all commonly applied image processing operations. Each operation class gives a generic description of a large set of operations with comparable behavior. The implementation of all sequential image processing operations in C1 is based on the operation classes to enhance library maintainability, and to improve flexibility.

Each operation class is implemented as a *generic algorithm*, using the C++ *function template mechanism* [22]. Each image operation that maps onto the functionality as provided by a generic algorithm is implemented by instantiating the generic algorithm with the proper parameters, including the function to be applied to the individual data elements. As an example, a generic algorithm may produce a result image by applying a unary function to each pixel in a given input image. By instantiating the algorithm with, for example, the absolute value operation on a single pixel, the produced output will constitute the input image with the absolute value taken for each pixel.

In the current version of our library the following set of generic algorithms has been implemented:

- *Unary pixel operation.* Operation in which a unary function is applied to each pixel in a given input image. Examples: negation, absolute value, square root.
- *Binary pixel operation.* Operation in which a binary function is applied to each pixel in a given input image. Examples: addition, multiplication, threshold.
- *Global reduction.* Operation in which all pixels in a given input image are combined to obtain a single result value. Examples: sum, product, maximum.
- *Neighborhood operation.* Operation in which several pixels in the neighborhood of each pixel in a given input image are combined. Examples: percentile, median.
- *Generalized convolution.* Special case of neighborhood operation. The combination of pixels in the neighborhood of each pixel is expressed in terms of two binary functions. Examples: convolution, gauss, dilation.
- *Geometric (domain) operation.* Operation in which a given input image's domain is transformed. Examples: translation, rotation, scaling.

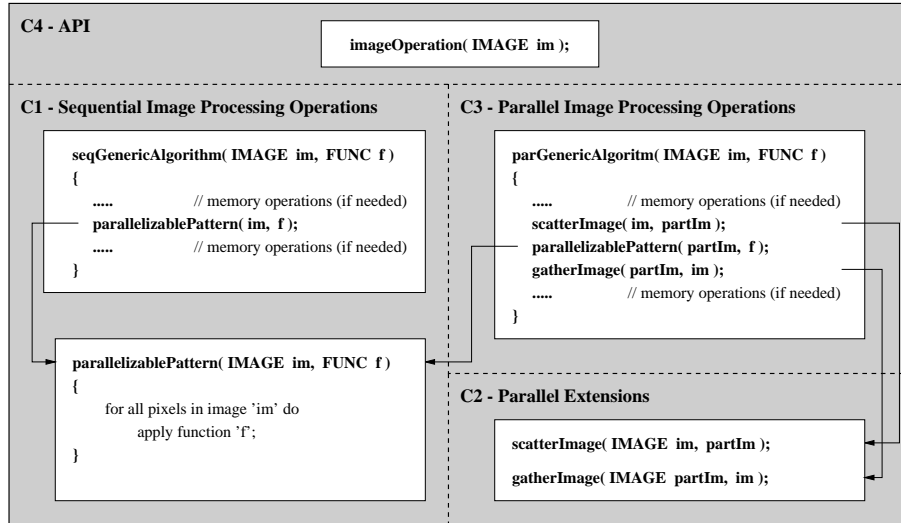
In the future additional generic algorithms will be added to this list, e.g. iterative and recursive neighborhood operations, and queue based algorithms. Apart from the geometric operations the set of generic algorithms is translation invariant. Translation variant versions of the operations will be incorporated in the future as well.

### 2.2.2 Parallel Extensions

The set of sequential operations is extended with several routines that introduce the parallelism into the library (C2 in Figure 1). The routines, implemented using MPI 1.1 [8], are separated into three classes:

1. Routines for image data *partitioning*. Used to specify the image data responsibilities for each processing unit, i.e. to indicate which data parts should be processed by each unit. In practice, a data structure is mapped onto a logical grid of processing units of up to 3 dimensions. The mapping is performed such that the number of elements each unit is responsible for is well-balanced.
2. Routines for image data *distribution* and *redistribution*. Used to scatter, gather, broadcast, and redistribute data structures. Essentially, the MPI 1.1 implementation available to us provides most of this functionality. For reasons beyond the scope of this report, we also have made additional implementations ourselves, using the standard (blocking) MPI 'send' and 'receive' calls.
3. Routines for *overlap communication*. Used to exchange *shadow regions*, such as image borders in neighborhood operations.

All classes of operations in C2 contain 'kernel' routines, and are not available to the user.



**Figure 2:** Relationships between library components C1, C2, C3, and C4.

### 2.2.3 Parallel Image Processing Operations

For each sequential generic algorithm it is possible to implement a separate, optimized parallel counterpart. However, this strategy requires that for each change in the implementation of a sequential operation the related parallel operation is updated as well. As a result, library maintainability is reduced.

This problem is avoided if the source code for the sequential generic algorithms is reused in the implementation of their respective parallel counterparts. To that end, for each generic algorithm we have defined a so-called *parallelizable pattern*. Each pattern constitutes the maximum amount of work in a generic algorithm that can be performed both sequentially and in parallel - in the latter case without having to communicate to obtain non-local data. An extensive discussion of parallelizable patterns is given in Section 3.

As shown in Figure 2, implementation of a sequential generic algorithm is obtained by concatenating basic memory operations (for the allocation and copying of data) and a single parallelizable pattern. Parallel implementations of generic algorithms are obtained by inserting communication operations in the concatenation of sequential library routines.

### 2.2.4 Single Uniform API

The image processing library is provided with an application programming interface (C4 in Figures 1 and 2) identical to that of an existing sequential library (Horus [13]). As such, the threshold for applying the library is low, as all parallelism is fully transparent to the user.

### 2.2.5 Annotated Performance Models

In our library we provide only one parallel implementation of each generic algorithm. To ensure efficiency of execution on all target platforms, the parallel generic algorithms are implemented such that they are capable of adapting to the performance characteristics of a specific parallel machine. To make these characteristics explicit, each library operation is annotated with a performance model for run-time cost estimation (C5 in Figure 1). As our library is intended to be portable to all computers in the class of distributed memory MIMD-style parallel machines, the performance models must be applicable to all these machines as well. An overview of the models applied in this research is given in Section 4.

### 2.2.6 Benchmarking Tool

For a specific machine, actual performance values for the model parameters are obtained by running a set of *benchmarking* operations (C6 in Figure 1). Based on the performance models and the benchmarking results, intra-operation optimization choices can be made automatically, fully transparent to the user. Apart from being an essential part of the optimization process, the obtained performance knowledge is useful as an indication to users and image library creators as well.

### 2.2.7 Algorithm Specification

Besides intra-operation optimization, optimization across library calls can be performed as well if information is available on the order in which library operations are applied in a given application. Essentially, this information is obtainable from the original program code. As implementation of a complete parser is not an essential part of this research, we currently assume that a complete algorithm specification is provided in addition to the program itself (C7 in Figure 1).

### 2.2.8 Scheduling Tool

Once the performance models, the benchmarking results, and the algorithm specification are available, a scheduling component (C8 in Figure 1) is applied to find an optimal solution for the application at hand. Note that this scheduling component is responsible for both intra- and inter-operation optimization. In the implementation of each parallel generic algorithm, requests for scheduling results are performed to determine which parallelization strategy is required. Whether scheduling results are static only, or should be generated and updated dynamically is still an important future research issue.

In the remainder of this report we focus on the two most important aspects of the software architecture. In the next section, the concept of parallelizable patterns is treated in detail. This discussion is followed by an overview of the performance models applied in the architecture.

### 3 Parallelizable Patterns in Low Level Image Processing

In this section we introduce the representation of images as applied in our library. Based on this representation we give a generalized description of parallelizable patterns. In addition, we show how such patterns are applied in the implementation of parallel generic algorithms.

#### 3.1 Representation of Digital Images

A digital image consists of a set of pixels. Associated with each pixel is a location (point) and a (pixel) value. Here, we denote an image by a lower case bold character from the beginning of the alphabet (i.e., **a**, **b**, or **c**). Locations are denoted by lower case bold characters from the end of the alphabet (i.e., **x**, **y**, or **z**). The pixel value of an image **a** at location **x** is represented by **a(x)**.

The set of all locations is referred to as the *domain* of the image, and is denoted by a capital bold character (i.e., **X**, **Y**, or **Z**). Usually, the point set is a discrete  $n$ -dimensional lattice  $\mathbb{Z}^n$ , with  $n = 1, 2$ , or  $3$ . Also, the point set is bounded in each dimension resulting in a rectangular shape for  $n = 2$  and a block shape for  $n = 3$ . That is, for an  $n$ -dimensional image

$$\mathbf{X} = \{(x_1, x_2, \dots, x_n) \in \mathbb{Z}^n : o_i \leq x_i \leq o_i + k_i - 1, i \in \{1, 2, \dots, n\}\},$$

where  $\mathbf{o} = (o_1, o_2, \dots, o_n)$  represents the *origin* of the image, and  $k_i$  represents the extent of the domain in the  $i$ -th dimension.

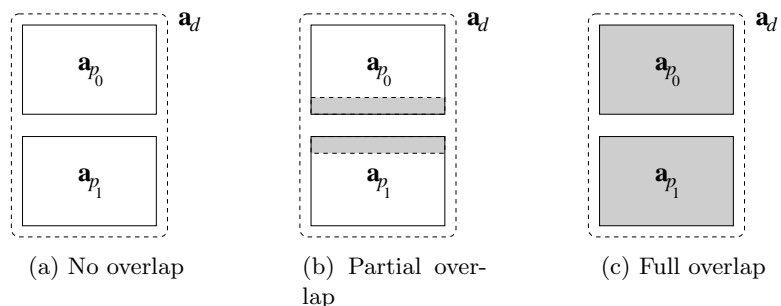
The set of all pixel values **a(x)** is referred to as the *range* of the image, and is denoted by  $\mathbb{F}$ . A pixel value is a vector of  $m$  scalar values, with  $m = 1, 2$  or  $3$ . A scalar value is represented by one of the common data types, such as byte, int, or float. The set of all images having range  $\mathbb{F}$  and domain  $\mathbf{X}$  is denoted by  $\mathbb{F}^{\mathbf{X}}$ . In summary,  $\mathbf{a} \in \mathbb{F}^{\mathbf{X}}$  (i.e.,  $\mathbf{a} : \mathbf{X} \rightarrow \mathbb{F}$ ) is a shorthand notation for

$$\{(\mathbf{x}, \mathbf{a}(\mathbf{x})) : \mathbf{x} \in \mathbf{X} \subset \mathbb{Z}^n (n = 1, 2, 3), \mathbf{a}(\mathbf{x}) \in \mathbb{F} \subset \{\mathbb{Z}^m, \mathbb{R}^m, \mathbb{C}\} (m = 1, 2, 3)\}.$$

When image data is spread throughout a parallel system, multiple data structures residing on different locations form a single logical entity. In our library, each image data structure obtained in a scatter or broadcast operation is considered a *partial image*. For such special type of image additional partitioning and distribution information is available. This information includes, but is not restricted to, (1) the processor grid used to map the original image data onto, (2) origin and size of the domain of the original image, and (3) the type of data distribution applied (e.g., scatter or broadcast). Partial image **a** residing on processing unit  $i$  is denoted by  $\mathbf{a}_{p_i}$ ; its domain is denoted by  $\mathbf{X}_{p_i}$ . As data spreading can not result in a loss of data, for each image  $\mathbf{a} \in \mathbb{F}^{\mathbf{X}}$  distributed over  $n$  processing units:

$$\bigcup_{i=0}^{n-1} \mathbf{X}_{p_i} = \mathbf{X}.$$

The  $n$  partial images related to **a** together form one logical structure, referred to as a *distributed image*. A distributed image is denoted by  $\mathbf{a}_d$ , and differs from a partial image in that it does not reside as a physical structure in the memory of



**Figure 3:** Three examples of a distributed image  $\mathbf{a}_d$  comprising of two partial images,  $\mathbf{a}_{p_0}$  and  $\mathbf{a}_{p_1}$ . The gray areas represent domain overlap; the white areas represent the unique domain parts.

one processing unit (unless it is formed by one partial image only). A distributed image's domain  $\mathbf{X}_d$  is given by the union of the domains of its related partial images. The domains of the partial images that constitute a distributed image may be either *non-overlapping*, *partially overlapping*, or *fully overlapping* (see Figure 3).

Essentially, it is possible for each processing unit to perform operations on each partial image *independently*. In the library, however, we make sure that each operation (logically) is performed on distributed image data only. In all cases this results in the processing of all partial images that constitute the distributed image. This strategy is of great importance to avoid inconsistencies in distributed image data.

## 3.2 Parallelizable Patterns

As stated in Section 2.2.3, we try to enhance library maintainability by reusing as much sequential code as possible in the implementations of the parallel generic algorithms. To that end, for each generic image processing algorithm we have defined a so-called parallelizable pattern. Each such pattern represents the maximum amount of work in a generic algorithm that - when applied to partial image data - can be performed without the need for communication. In other words, in a parallelizable pattern all internal data accesses must refer to data *local* to the processing unit executing the operation. In the following we give a generalized description of parallelizable patterns, and show their application in parallel implementations.

### 3.2.1 Parallelizable Patterns: The General Case

A parallelizable pattern is a sequential generic operation that takes zero or more source structures as input, and produces one destination structure as output. Each pattern consists of  $n$  independent tasks, where a task specifies what data in any of the structures involved in the operation must be acquired (read), in order to update (write) the value of a *single* data point in the destination structure. In each task read access to the source structures is unrestricted, as long as no accesses are performed outside any of the structures' domains. In contrast, read access to the destination structure is limited to the single data point to be updated.

All  $n$  tasks are tied to a different *task location*  $\mathbf{x}_i$ , with  $i \in \{1, 2, \dots, n\}$ . The set  $TL$  of all task locations constitutes a subset of the positions inside the domain of one of the data structures involved in the operation (either source or destination). As a simple example,  $TL$  may refer to all  $n$  pixels in an image data structure, all of which are processed in a loop of  $n$  iterations.

Each task location  $\mathbf{x}_i$  has a relation to the positions accessed in all data structures involved in the operation. As such, for the parallelizable patterns relevant in image processing we define four *data access pattern types*:

- *One-to-one*. For a given data structure, in each task  $T_i$  (with  $i \in \{1, 2, \dots, n\}$ ) no data point is accessed other than  $\mathbf{x}_i$ .
- *One-to-one-unknown*. For a given data structure, in each task  $T_i$  (with  $i \in \{1, 2, \dots, n\}$ ) not more than one data point is accessed. In general, this point is not equal to  $\mathbf{x}_i$ .
- *One-to-M*. For a given data structure, in each task  $T_i$  (with  $i \in \{1, 2, \dots, n\}$ ) no data points are accessed other than those within the *neighborhood* of  $\mathbf{x}_i$ . As an example, the  $5 \times 3$  neighborhood of a point  $\mathbf{x} = (x_1, x_2) \in \mathbf{X}$  is given by

$$N(\mathbf{x}) = \{\mathbf{y} \in \mathbf{Y} : \mathbf{y} = (x_1 \pm j, x_2 \pm k), j \in \{0, 1, 2\}, k \in \{0, 1\}\},$$

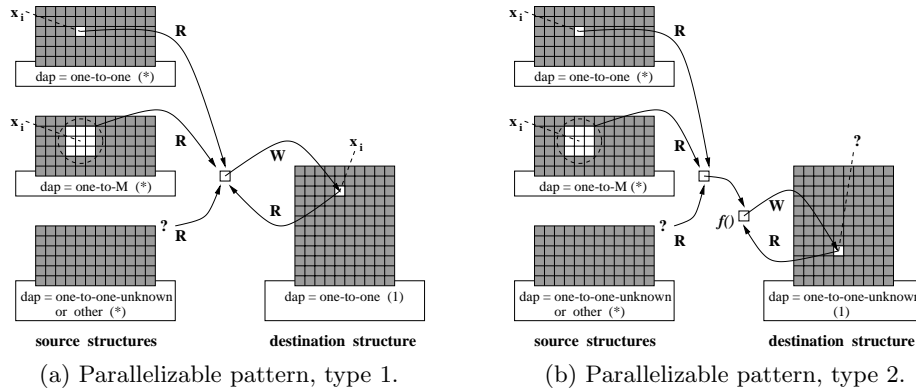
where  $\mathbf{X} \subset \mathbf{Y}$ .

- *Other*. For a given structure, in each task either all elements are accessed, or the accesses are irregular or unknown.

A parallelizable pattern requires that for all data structures the access pattern type is given. Essentially, all four access pattern types are applicable to source structures. In contrast, the single destination structure can only have a 'one-to-one' or a 'one-to-one-unknown' access pattern type. This is because - by definition - in each task only one data point is accessed in the destination structure.

Figure 4 shows the two parallelizable pattern types that we discern. In a type 1 parallelizable pattern the set of task locations has a 'one-to-one' relation to the destination structure. In a type 2 parallelizable pattern, on the other hand, the access pattern type related to the destination structure is of type 'one-to-one-unknown'. Furthermore, the two parallelizable patterns differ in the type of *combination operation* that is permitted. In a parallelizable pattern of type 1 no restrictions are defined for the combination operation. In a type 2 pattern, on the other hand, the final combination of the intermediate result of all values read from the source structures with the value of the data point to be updated, must be performed by a function  $f()$  that is associative and commutative. Also, prior to execution of a type 2 pattern, all elements in the destination structure must have a value that is 'neutral' for operation  $f()$ . As an example, the neutral value for addition is 0, while for multiplication it is 1.

The two parallelizable pattern types give a generalization of a large set of sequential image processing operations, e.g. covering all operations of Section 2.2.1. It should be noted that the two types do not capture the complete set of all possible



**Figure 4:** *Two parallelizable pattern types.  $R$  = read access;  $W$  = write access;  $dap$  = data access pattern; (1) = exactly one data structure of this type; (\*) = zero or more data structures of this type.*

operations. For example, operations in which the values of data points in the source structures are updated do not fall in the category of operations currently under consideration. The same holds for operations in which the value of each data point in the destination structure depends on values of other data points in the same destination structure as well. Still, all generic algorithms that do map onto the given generalization are applicable in the process of 'parallelization by concatenation of library operations', as introduced in Section 2.2.3

### 3.2.2 Parallelizable Patterns: General Parallelization Strategy

The number of elements in  $TL$  determines the number of steps executed by a parallelizable pattern. By providing each processing unit in a parallel system with a set  $X \subset TL$ , the total amount of work is distributed. In addition, based on the access pattern type defined for each structure involved in the operation, non-local data accesses can be avoided with minimal communication overhead.

Before executing a type 1 parallelizable pattern each processing unit must be provided with a non-overlapping partial destination structure that matches the elements in  $X$ . If the destination structure is updated but never read, the partial structure can be created locally. Otherwise, it is obtained by scattering the destination structure such that no overlap in the domains of the local partial structures is introduced. Before executing a type 2 pattern, each processing unit must create a fully overlapping destination structure locally. This is always possible, as the value of all data points must be given a 'neutral value', defined by the operation.

Source data structures are obtained by executing (1) a non-overlapping scatter for each structure having a one-to-one access pattern, (2) a partially overlapping scatter for each structure having a one-to-M access pattern type (such that in each dimension the size of each shadow region equals half the size of the neighborhood in that dimension), and (3) a broadcast for all other structures. Alternatively, if the values of a source structure can be calculated locally, one may decide to do so.

When a type 1 parallelizable pattern has finished, the complete destination structure is obtained by executing a gather operation. For a type 2 parallelizable pattern this is achieved by executing a reduce operation across all processing units. Here, the elements that have not been updated in each local destination structure have kept a neutral value, and assure the correctness of the final reduction. In both cases, the result structure may be returned either to one node, or to all.

In the following, we shortly discuss parallelization of two example generic algorithms, i.e. global reduction and generalized convolution. We will investigate the access pattern types for the data structures involved in the operations. Also, for both generic algorithms a related parallelizable pattern will be given.

### 3.2.3 Example: Parallel Reduction

Referring to the image representations introduced earlier, a sequential generic reduction operation performed on input image  $\mathbf{a}$ , producing a single scalar or vector value  $k$ , is defined as follows:

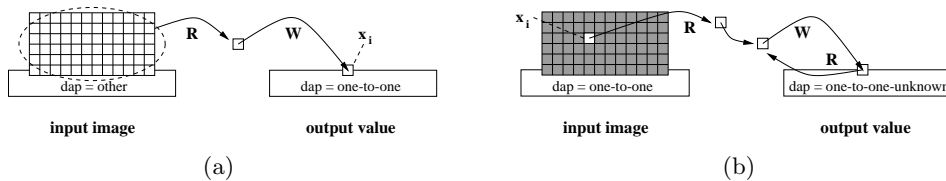
Let  $\mathbf{a} \in \mathbb{F}^{\mathbf{X}}$  and  $k \in \mathbb{F}$ , then

$$k = \Gamma \mathbf{a} = \Gamma_{\mathbf{x} \in \mathbf{X}} \mathbf{a}(\mathbf{x}) = \Gamma_{i=1}^n \mathbf{a}(x_i) = \mathbf{a}(x_1) \gamma \mathbf{a}(x_2) \gamma \cdots \gamma \mathbf{a}(x_n),$$

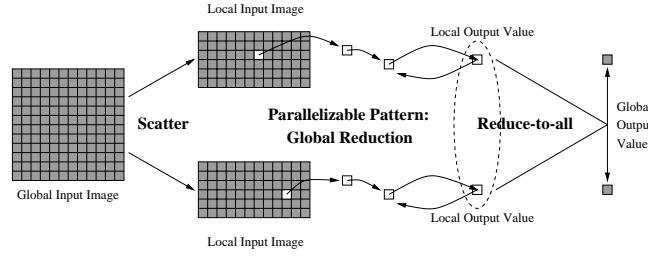
with  $\gamma$  an associative and commutative binary operation on  $\mathbb{F}$ .

As shown in Figure 5, at least two possible sequential implementations exist for this operation. In the first implementation, the operation is performed in one step. All data points in  $\mathbf{a}$  are obtained and combined to a single value, which is written out to  $k$ . In the second implementation, the operation is performed in  $n$  steps. In each step, one data point in  $\mathbf{a}$  is read and combined with the current value of  $k$ .

The two implementations both constitute a specialization of one of the generalized parallelizable patterns described in Section 3.2.1 (i.e., a type 1 pattern and a type 2 pattern respectively). The first implementation is not preferred, however, as its execution is limited to a single processing unit. This is because the (implicit) set of task locations  $TL$  consists of one element only, i.e. the location of the single output value  $k$ . The second implementation, on the other hand, is easily run in parallel, as  $TL$  contains all locations in input image  $\mathbf{a}$ . For this implementation the input image's access pattern type is 'one-to-one'; for the single result value it is 'one-to-one-unknown'.



**Figure 5:** *Sequential reduction - two implementations.*



**Figure 6:** Example reduce-to-all operation executed on 2 processing units.

tion operation follows directly from the generalization of Section 3.2.2. A pictorial view of the operation executed in parallel is given in Figure 6.

### 3.2.4 Example: Parallel Generalized Convolution

A generalized convolution performed on input image  $\mathbf{a}$ , producing output image  $\mathbf{c}$ , given a kernel  $\mathbf{t}$ , is defined as follows:

Let  $\mathbf{a}, \mathbf{c} \in \mathbb{F}^{\mathbf{X}}$ ,  $\mathbf{t} \in \mathbb{F}^{\mathbf{Y}}$  with  $\mathbf{Y} = \{(y_1, y_2, \dots, y_n) : |y_i| \leq k_i \in \mathbb{Z}\}$ , and with  $\mathbf{X}$  having dimensionality  $n$ , then

$$\mathbf{c} = \mathbf{a} \odot \mathbf{t} = \{ (\mathbf{x}, \mathbf{c}(\mathbf{x})) : \mathbf{c}(\mathbf{x}) = \Gamma_{\mathbf{y} \in \mathbf{Y}} \mathbf{a}(\mathbf{x} + \mathbf{y}) \circ \mathbf{t}(\mathbf{y}), \mathbf{x} \in \mathbf{X} \},$$

where  $\circ$  and  $\gamma$  are binary operations on  $\mathbb{F}$ , and  $\gamma$  is associative and commutative. The extent of the domain in the  $i$ -th dimension of kernel  $\mathbf{t}$  is given by  $2k_i + 1$ . Several common generalized convolution instantiations are shown in Table 1.

The definition states that each pixel value in the output image depends on the pixel values in the *neighborhood* of the pixel at the same position in the input image, as well as on the values in the related kernel structure. A sequential implementation of the operation is presented in Figure 7. Again, set  $TL$  is implicit, and contains all pixel positions in either the input image or the output image.

When comparing Figure 4(a) to 7(a) it may seem that the operation directly constitutes a parallelizable pattern. Figure 7(b) shows that this is not the case, however, as accesses to pixels outside the input image's domain are possible. In sequential implementations of this operation it is common practice to redirect such accesses according to a predefined *border handling strategy* (e.g., mirroring or tiling). A better approach for sequential implementation, however, is to separate the border handling from the actual convolution operation. This makes implementations more

Kernel Operation	$\circ$	$\gamma$
Convolution	multiplication	addition
Dilation	addition	maximum
Erosion	addition	minimum

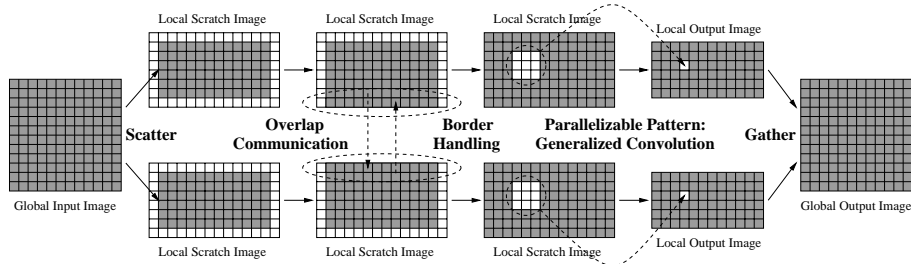
**Table 1:** Example generalized convolution instantiations.



**Figure 7:** *Sequential generalized convolution. Does not represent a parallelizable pattern, as read accesses outside the domain of the input image are possible (see (b)).*

robust and, in general, also faster. For parallel implementation this strategy has the additional advantage that the generic algorithm for generalized convolution can be implemented such that it constitutes a parallelizable pattern.

Implementation in this manner can be performed in many different ways. In our library a so-called *scratch border* is placed around the original input image. The border is filled with pixel values according to the required border handling strategy. The newly created *scratch image* is used as input to the parallelizable pattern. Figure 8 depicts the operation executed in parallel. As each local scratch image has a one-to-M access pattern, an overlapping scatter of the global input image is required. In Figure 8 this is implemented by a non-overlapping scatter followed by overlap communication. Remaining scratch border data is obtained by local copying. Finally, the parallelizable pattern is executed, producing local result images that are gathered to obtain the complete output image. Note that Figure 8 gives a simplified view, as some steps of the operation are not shown. For example, depending on the type of operation, the kernel is either broadcast or calculated locally.



**Figure 8:** *Example kernel operation performed using 2 processing units (simplified).*

### 3.3 Discussion

The generalized description of parallelizable patterns is important as it states the requirements for *sequential* implementation of a large set of generic low level image processing operations. In addition, for each specialized parallelizable pattern implemented on the basis of the generalized description, a parallelization strategy directly follows. As such, code reusability is maximized, and library maintainability

and flexibility is enhanced.

It should be noted that if a sequential operation does not map onto the generalized description of a parallelizable pattern, we currently take no special action to obtain good performance. In such situations, the operation is always executed using one processing unit only. In the future we will investigate whether parallelization of such operations can be generalized as well. Additional formulations may be integrated in the current generalization, or may exist independently.

## 4 Performance Models

As shown in the previous section, in all parallel implementations both the parallelization granularity as well as the data dependencies have been fixed. In each individual operation, however, additional optimization decisions still can be made at run time. Such decisions relate to the number of processing units to be used, the preferred logical processor grids, the routing strategies for communication, etcetera. As was indicated in Section 2.2, the decisions are made on the basis of performance models attached to each library operation.

In this section an overview is given of the performance models. First, the requirements for such models are investigated. Second, a short description is given of the abstract parallel image processing machine (APIPM) that is used as a basis for all performance models. Finally, APIPM-based performance models are introduced that capture the relevant behavior of all library operations.

### 4.1 General Performance Model Requirements

Naturally, a performance model designed for our purposes should incorporate all relevant tasks typically performed by image processing operations executing in data parallel fashion. As was indicated in the previous sections, in our library such tasks relate to either *computation* or *communication*. Computational tasks include parallelizable patterns as well as basic memory operations. Communicating tasks are formed by (the bulk of) routines from the parallel extensions described in Section 2.2.2.

Apart from having to reflect the typical behavior of parallel low level image processing routines, the performance models should also conform to the following (more general) requirements:

1. *Simplicity*. The more detailed a model, the less manageable it is and the more expense will go into obtaining its performance measures. To reduce the costs of static or run-time model evaluation the number of parameters in the model should be kept to a minimum.
2. *Accuracy*. To make sure the library routines can make 'clever' decisions regarding issues of parallel execution, performance estimates obtained from the model should be sufficiently accurate. The degree of accuracy is considered sufficient if good decisions are made in most situations (preferably in at least 95 percent of all cases), and poor decisions are generally avoided.

3. *Applicability*. As it is our intention to make the library available for a range of parallel machines, it must be portable to different types of architectures. Naturally, the related performance models must be applicable to all such platforms as well.

Note that the requirement of simplicity enhances applicability, but reduces accuracy. As a result, care must be taken in the design of the models to ensure that they produce good estimates with relative ease.

## 4.2 Performance Models Based on a High Level Abstract Machine

In the creation of a performance model we must rely heavily on intuition instead of precise characterizations. This is particularly true when deciding which system properties to include in a model, and which to ignore. However, by introducing well-defined abstractions for both the operations as well as the machines to be applied, our intuition can be guided somewhat.

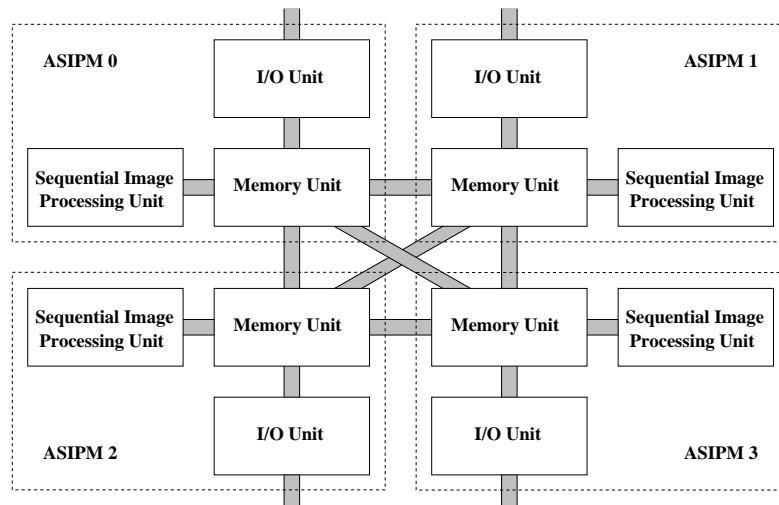
In our research we have introduced such abstractions by defining a high level abstract machine for image processing: the APIPM (or Abstract Parallel Image Processing Machine). In the APIPM common hardware characteristics of the target machines are reflected by the definition of *abstract hardware components*. In addition, the typical behavior of the routines to be run are reflected in a related *instruction set*.

On the one hand, we have made the level of abstraction in the APIPM as high as possible to hide irrelevant details of the underlying system. Referring to the list of requirements of Section 4.1, a high level of abstraction allows for the creation of performance models that are relatively *simple*. On the other hand, the abstract machine is made low-level enough to capture the specific behavior of a realistic parallel image processing machine. This enables localization of the relevant costs involved in a running system and thus allows for the creation of performance models that are sufficiently *accurate*. Finally, we claim that an APIPM-based performance model is *applicable* to all systems that have a behavior similar to that of the abstract machine. Such systems, of course, are the ones we are aiming at in this project: general purpose distributed memory MIMD-style parallel machines.

### 4.2.1 Abstract Parallel Image Processing Machine: Components

An APIPM consists of one or more abstract sequential image processing machines (ASIPMs), each consisting of four closely related components (see Figure 9):

1. a *sequential image processing unit*, capable of executing APIPM instructions, one at a time,
2. a *memory unit*, capable of storing (image) data structures,
3. an *I/O unit*, for transporting data between the memory unit and external sensing or storage devices,
4. *data channels*, the means by which data is transported between the ASIPM units and external devices.



**Figure 9:** *Abstract Parallel Image Processing Machine (APIPM) comprising of four ASIPMs.*

Although the memory unit of each ASIPM is connected with those of all other ASIPMs, no ASIPM has direct access to data maintained by any other ASIPM.

Note that the 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. Each ASIPM is capable of running code individually, independent of all other ASIPMs. The programs executed by each ASIPM need not be identical. Exchange of data between processing units is possible by communication over the interconnecting data channels.

Although most realistic distributed memory MIMD-style parallel machines do not have a fully connected communication network, we have decided to include one in the APIPM. This is because in most multicomputer systems communication is currently based on circuit-switched message routing, which makes a network *virtually* fully connected. Essentially, this means that data sent to a node that is not directly connected to the sending node does not need to interrupt the processing units of the intermediate nodes on the complete send path. In other words, the time needed to send a message from one node to another is not significantly influenced by the distance between the nodes.

#### 4.2.2 Abstract Parallel Image Processing Machine: Instruction Set

The APIPM instruction set (see Figure 10) consists of four classes of operations:

1. *Generic image processing instructions*, i.e. the specialized parallelizable patterns described in Section 3.
2. *Memory instructions*, for allocation and copying of (image) data.

3. *I/O instructions*, for transporting data between the memory unit and external devices.
4. *Communication instructions*, for exchanging data among ASPIM units.

For reasons of simplicity, in the overview of Figure 10 the operands (i.e., arguments) for each opcode have been left out. A complete overview is given in [20].

In the instruction set we have included only two communication instructions (i.e., `SEND` and `RECV`). Collective communication operations are not included, as these can be created using the two point-to-point operations only. The definition of the `SEND` and `RECV` instructions is identical to the standard blocking communication operations available in MPI [8] (i.e., `MPI_Send()` and `MPI_Recv()`).

In the abstract machine multiple real-world objects must be represented, which should be passed as parameters to the APIPM instructions. The most prominent objects are images, but kernels, matrices, and the likes, are essential as well. In the instruction set we do not introduce a special data representation for each of these objects. Instead, we make use of *memory references*. Such references contain information about the internal data representation, but lack any form of semantic information. The semantics are determined by the APIPM instruction the memory reference is passed to as a parameter.

It is important to note that for several generic image processing operations in the instruction set *data element homogeneity* is required. This means that the scalar type and the dimensionality of the elements in multiple data structures passed as parameters to a single instruction must be identical. The restriction of data element homogeneity is enforced to acknowledge the differences between operations on homogeneous and heterogeneous types. If homogeneity would not be required

<b>opcode</b>	generic image processing instructions
UPOP	unary pixel operation
BPOPC	binary pixel operation (constant value as argument)
BPOPI	binary pixel operation (complete image as argument)
REducOP	global reduction operation
NEIGHOP	neighborhood operation
GCONVOP	generalized convolution
GEOMAT	geometric transformation (matrix as argument)
GEOROI	geometric transformation (region of interest)
<b>opcode</b>	memory instructions
CREATE	allocate data block in memory unit
MEMCOPY	copy data in memory unit
DELETE	free up data block in memory unit
<b>opcode</b>	I/O instructions
IMPORT	import data from external device
EXPORT	export data to external device
<b>opcode</b>	communication instructions
SEND	send data to other ASIPM
RECV	receive data from other ASIPM

**Figure 10:** Simplified APIPM instruction set.

additional casting or copying of data would be hidden inside the APIPM. For many instructions such additional tasks constitute a significant overhead, which must be made explicit.

As a final note we should state that the above description of the abstract parallel image processing machine and its related instruction set is not complete. By this we mean that it can not be used to serve as a basis for an actual implementation. This, however, was also not the fundamental reason for introducing the abstract machine. We stress that the sole purpose of the definition of the APIPM is for it to serve as a basis for platform independent performance models. Many components deliberately have been left out of the specification, to keep the APIPM-based performance models as simple as possible.

### 4.3 APIPM-based Performance Models

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

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

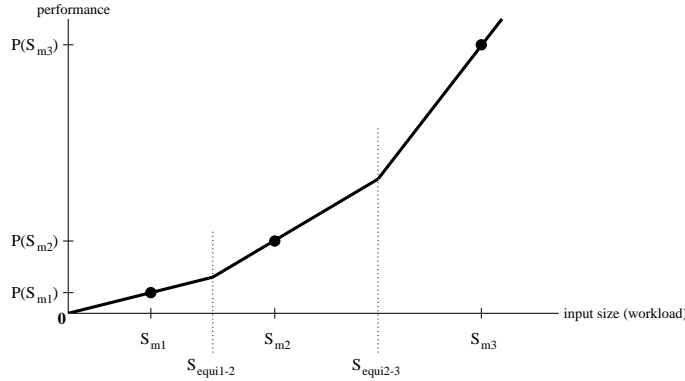
A problem with the simplistic model formalized here is that most APIPM instructions are not single static entities. This is because the execution of an instruction is often dependent on the values of its operands. Therefore, a static entity for each possible operand combination must be incorporated in our model. To avoid an explosion of the number of static entities we allow each instruction  $I_i$  and each value  $P_{I_i}$  to be *parameterized*. As we have not discussed the operands of the APIPM instructions we will not give a detailed overview of the model parameterization. To give a straightforward example, however, in almost all APIPM instructions a 'datatype' parameter is incorporated (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  $P_{I_i(datatype)(size)}$ ). Note that the choice of model parameters is dependent on the actual library implementation of each APIPM instruction. Again, for a complete overview we refer to [20].

### 4.3.1 Model-Related Benchmarking

As it is our goal to include no knowledge of underlying hardware, it is not advisable to make strict assumptions about performance growth rates in relation to data input size. However, to avoid benchmarking for each possible input size, we still assume a linear growth rate in the region around each measured size (see Figure 11). In the general case, the performance of an instruction  $I$ , for each input size  $S_{in}$  within the region around the measured size  $S_m$  (bounded at the bottom end by data input size  $S_{equi}$ , having performance value  $P_I(S_{equi})$ ) is obtained by

$$P_I(S_{in}) = (S_{in} - S_{equi}) \left( \frac{P_I(S_m) - P_I(S_{equi})}{S_m - S_{equi}} \right) + P_I(S_{equi})$$

The region around the smallest sized measurement is bounded by a workload of 0 at the bottom end. As the important topic of benchmarking is outside the scope of this report, no additional details will be given here.



**Figure 11:** Assumed performance growth in relation to workload on the basis of three measured performance values.

## 4.4 Extended Model for Point-to-Point Communication

Whereas the model described in Section 4.3 is sufficient for all sequential APIPM instructions, for the two communication instructions an extension is required. This is because a prediction of the time a processor is busy executing a **SEND** or a **RECV** instruction is not an accurate measure for the time involved in a complete message transfer. In other words, the end-to-end communication time is not incorporated in the current model.

For this reason we have designed an extended model for point-to-point communication. The model, called P-3PC (or the *Parameterized model based on the Three Paths of Communication*), closely resembles other models described in the literature (e.g., the Postal Model [2, 4], LogP [6], and LogGP [1]), but is more accurate. The model acknowledges the difference in the time both communicating nodes are occupied in a message transfer, and the complete end-to-end delivery time. In addition,

the model closely matches the MPI abstractions in that it incorporates a possible difference in the communication of data stored either contiguously or noncontiguously in memory.

In the model we introduce the notion of the *three paths of communication*, and assume that the cost of transferring a message from a sender to a receiver is captured in three independent values:

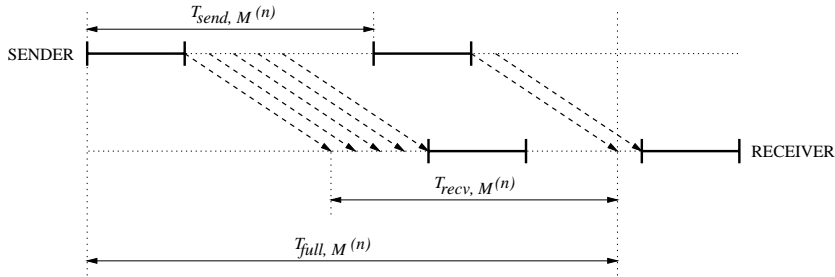
- $T_{send}$ : the cost related to the communication path at the sending node. This value represents the time from the moment the sender initiates a transmission until it has finished communicating and is ready to continue (i.e., the time required for executing the `SEND` instruction).
- $T_{recv}$ : the cost related to the communication path at the receiving node. This value represents the time from the moment the receiver is ready to accept a message until it has safely stored all data and is ready to continue (i.e., the time required for executing the `RECV` instruction).
- $T_{full}$ : the cost related to the full communication path. This value represents the end-to-end delivery time, or the time from the moment the sender initiates a transmission until the moment the receiver has safely stored all data and is ready to continue.

For each path we assume that the transmission of a message involves a constant amount of time, which is captured by the mutually independent parameters  $t_{cs}$ ,  $t_{cr}$ , and  $t_{cf}$  (for the sender, receiver, and full path respectively). In addition, for each communication path we assume an 'additional time' ( $t_{as}$ ,  $t_{ar}$ , and  $t_{af}$  respectively) which is a function of the number of bytes transmitted. To capture differences in the sending of contiguous and noncontiguous data, the model is parameterized with a cost indicator  $M$ , which represents the memory layout at the two communicating nodes. The three communication times (see also Figure 12) involved in the transmission of a message containing  $n$  bytes are then given by:

$$\begin{aligned} T_{send,M}(n) &= t_{cs} + t_{as,M}(n), \\ T_{recv,M}(n) &= t_{cr} + t_{ar,M}(n), \\ T_{full,M}(n) &= t_{cf} + t_{af,M}(n), \end{aligned}$$

where  $M \in \{cc, cn, nc, nn\}$ . These four layout descriptors indicate the four possible memory layout combinations at the sender and the receiver combined. For example,  $cn$  means that a contiguous block of data is transmitted by the sender, which is accepted noncontiguously by the receiver.

A complete description of the P-3PC model is given in [21]. In this report we have made a comparison with well-known models described in the literature, and argue why P-3PC is more suitable for our needs. In addition we have shown that P-3PC is highly accurate in modeling MPI's standard blocking point-to-point communication operations, as well as collective operations implemented using point-to-point operations exclusively.



**Figure 12:** *Communication according to the P-3PC Model.*

## 4.5 Discussion

The most important advantage of the APIPM-based performance models is that predictions are based on very *high level* instructions. Modeling on the basis of much lower level instructions is possible as well, but execution times of such instructions tend to be less independent than those of higher level instructions. This is mainly caused by possible optimizations performed by the compiler used. Also, obtaining accurate values for lower level instructions is much more difficult. This is due to the inherent intrusiveness of the benchmarking process.

Our models resemble the model described in [18], which was used for general machine characterization based on an Abstract Fortran Machine (AFM). The instruction set used in this model incorporates the primitive operations available in Fortran. As performance evaluation on the basis of the AFM proved to be highly accurate, we expect results of a similar accuracy for the higher level APIPM-based performance models.

A possible drawback of our models is that the instructions and related performance values are parameterized with quite a large number of instruction behavior and workload indicators. Obtaining accurate performance values for all possible combinations of parameters is both costly and difficult. However, it is possible to combine several parameters to obtain a more general indicator. As an example, promising candidates for parameter merging are those that relate to data structure sizes (e.g., width, height, depth, etc.). In addition, a benchmarking tool should allow a user to set regions of interest, to restrict the set of all possible measurements. For this reason we feel that the performance models are both powerful and useful for our purposes.

## 5 Performance Measurements and Validation

In this section we show how a realistic image processing application [9], implemented using our library, is executed in parallel. First, a detailed description is given of the underlying algorithm. Next, both a straightforward sequential implementation as well as the related parallel implementation are discussed. Finally, performance measurements are compared with predictions obtained from the annotated performance models.

## 5.1 Detection of Lines in Images

The detection of lines in images is a fundamental low-level task in computer vision. Theoretically, in two-dimensions, line points are detected by considering the second order directional derivative in the gradient direction. For a line point, the second order directional derivative perpendicular to the line is a measure of line contrast, given by

$$\lambda = f_{ww}(x, y) \quad (1)$$

where  $f(x, y)$  is the grey-value function and the indices  $w$  denote differentiation in the gradient direction. Bright lines are observed when  $\lambda < 0$  and dark lines when  $\lambda > 0$ .

In practice, one can only measure differential expressions at a certain observation scale [7, 14]. By considering Gaussian weighted differential quotients in the gradient direction,  $f_{ww}^\sigma = G_{ww}(\sigma) * f(x, y)$ , a measure of line contrast is given by

$$r(x, y, \sigma) = \sigma^2 |f_{ww}^\sigma| \frac{1}{b^\sigma} \quad (2)$$

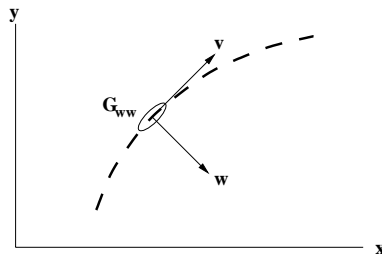
where  $\sigma$ , the Gaussian standard deviation, denotes the scale for observing the line structure, and where line brightness  $b$  is given by

$$b^\sigma = \begin{cases} f^\sigma & \text{if } f_{ww}^\sigma \leq 0, \\ W - f^\sigma & \text{otherwise.} \end{cases} \quad (3)$$

Line brightness is measured relative to black for bright lines, and relative to white level  $W$  (255 for an 8-bit camera) for dark lines.

The response of the second order directional derivative  $\lambda$  does not only depend on the image data, but it is also affected by the Gaussian smoothing scale  $\sigma$ . Because a line has a large spatial extent along the line, and only a small spatial extent (i.e., the line width) perpendicular to the line, the Gaussian filter should be tuned to optimally accumulate line evidence. For directional filtering (see Figure 13), anisotropic Gaussian filters may be used of scale  $\sigma_v$  and  $\sigma_w$ , for longest and shortest axis, respectively. Hence, line contrast is given by

$$r'(x, y, \sigma_v, \sigma_w) = \sigma_v \sigma_w |f_{ww}^{\sigma_v, \sigma_w}| \frac{1}{b^{\sigma_v, \sigma_w}}, \quad (4)$$



**Figure 13:** Directional filtering for line detection. Filter  $\mathbf{G}_{ww}$  is oriented in the line direction; local coordinate system indicated by  $(\mathbf{v}, \mathbf{w})$ .

where  $b^{\sigma_v, \sigma_w}$  is given by (eq. 3). The scale  $\sigma_v$  for the longest axis of the Gaussian accumulates line evidence along the line, whereas  $\sigma_w$  the shortest axis differentiates in the gradient direction and should be tuned to adequately capture line width.

Now that we have established how to filter in a particular direction, the filter must be tuned to the line direction. The optimal filter orientation may be different for each position in the image plane, depending on line evidence at the particular image point under consideration. The final line detection filter, parameterized by orientation  $\theta$ , smoothing scale  $\sigma_v$  in the line direction, and differentiation scale  $\sigma_w$  perpendicular to the line, is given by

$$r''(x, y, \sigma_v, \sigma_w, \theta) = \sigma_v \sigma_w \left| f_{ww}^{\sigma_v, \sigma_w, \theta} \right| \frac{1}{b^{\sigma_v, \sigma_w, \theta}} \quad (5)$$

where  $f_{ww}^{\sigma_v, \sigma_w, \theta} = G_{ww}(\sigma_v, \sigma_w, \theta) * f(x, y)$ . When the filter is correctly aligned with the line, and  $\sigma_v, \sigma_w$  are optimally tuned to capture the line, filter response is maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection,

$$R(x, y) = \arg \max_{\sigma_v, \sigma_w, \theta} r''(x, y, \sigma_v, \sigma_w, \theta) \quad (6)$$

Equation (6) states that a filter bank is applied over the input image, where each filter represents one sample of the parameter space  $(\sigma_v, \sigma_w, \theta)$ . The final result is obtained by considering the maximum response per pixel over all filter results. This yields the optimal orientation  $\theta$ , an estimate of line thickness  $\sigma_w$ , the best smoothing size  $\sigma_v$ , and the line contrast  $R(x, y)$ .

## 5.2 Sequential Implementation

The directional filtering problem can be implemented sequentially in many different ways. For each orientation  $\theta$  it is possible to create a new filter based on  $\sigma_w$  and  $\sigma_v$ . In effect, this yields a rotation of the filters, while the orientation of the input image remains fixed. Another possibility is to keep the orientation of the filters fixed, and to rotate the input image instead. Yet another solution is to integrate the notion of orientation in the filter operation itself. In this case image pixels are not only accessed according to the size of the neighborhood of the filter, but also on the basis of the given orientation.

In this example, we have implemented the operation by applying fixed filters to rotated image data. We have chosen this implementation as we expect it to be the solution preferred by most image processing researchers. As such, the implementation reflects parallelization problems encountered in a realistic situation. We need to stress, however, that we do not claim that this particular implementation provides optimal performance when executed either sequentially or in parallel.

The main body of the sequential implementation is presented in pseudo code in Listing 1. The program starts by rotating the original input image for a given orientation  $\theta$ . In addition, for all  $(\sigma_v, \sigma_w)$  combinations the filtering is performed by six library operations executed in sequence. First,  $f_{ww}^{\sigma_v, \sigma_w, \theta}$  and  $b^{\sigma_v, \sigma_w, \theta}$  (or `Filtered1_IM` and `Filtered2_IM`, respectively) are produced by executing two generalized convolution operations, each with the appropriate parameters. For cost effectiveness the

```

FOR all orientations  $\theta$  DO
  Rotated_IM = GeometricOp(Original_IM, "rotate",  $\theta$ );
  FOR all smoothing scales  $\sigma_v$  DO
    FOR all differentiation scales  $\sigma_w$  DO
      Filtered1_IM = GenConvOp(Rotated_IM, "gauss",  $\sigma_w$ ,  $\sigma_v$ , 2, 0);
      Filtered2_IM = GenConvOp(Rotated_IM, "gauss",  $\sigma_w$ ,  $\sigma_v$ , 0, 0);
      Detected_IM = BinPixImArgOp(Filtered1_IM, "absdiv", Filtered2_IM);
      Detected_IM = BinPixCnstArgOp(Detected_IM, "mul",  $\sigma_v * \sigma_w$ );
      BackRotated_IM = GeometricOp(Detected_IM, "rotate",  $-\theta$ );
      Contrast_IM = BinPixImArgOp(Contrast_IM, "max", BackRotated_IM);
    OD
  OD
OD

```

Listing 1: *Pseudo code for the directional filtering program.*

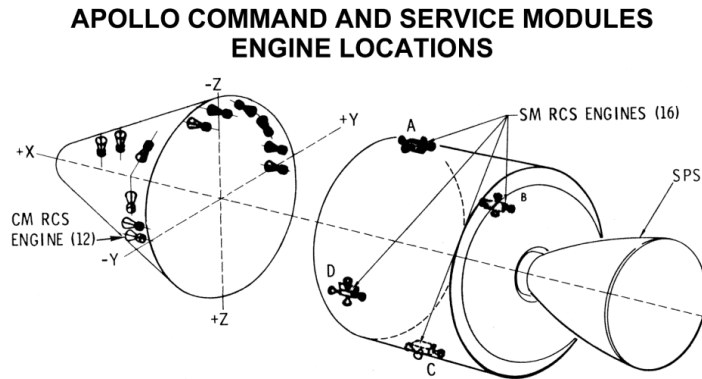
Gaussian convolutions are performed by applying two 1-dimensional filters in both cases. Next, the result of Equation (5) is obtained by executing two binary pixel operations, one having an image, the other having a constant value as argument. Finally, the result image is rotated back to match the orientation of the original input image, and the maximum response image is obtained.

Figure 14 gives a typical example of an image that is used as input to the program. The result obtained after applying the program for a reasonably large parameter subspace of  $(\sigma_v, \sigma_w, \theta)$  is shown in Figure 15. On a state-of-the-art sequential machine the program may take from a few minutes up to several hours to complete, depending on the size of the input image and the extent of the chosen parameter subspace. Consequently, for the directional filtering program parallel execution is highly desired.

### 5.3 Parallel Execution

As all parallelization issues are shielded from the user, the pseudo code of Listing 1 directly constitutes a program that can be executed in parallel as well. Optimization of the efficiency of the program is to be taken care of by the architecture's scheduling component. As a fully functional scheduling tool is not yet available in the current version of our architecture, we have created two different schedules for the program by hand. In the first schedule *all* library operations are forced to run in parallel, using all available processing units. The second schedule differs from the first in that the last two operations in the innermost loop of the program are run on one node only.

In both schedules the `Original_IM` structure must be broadcast to all nodes. This is because the structure is applied in the initial rotation operation, which expects it to have a data access pattern of type 'other'. This broadcast needs to be performed only once, as `Original_IM` is not updated in subsequent operations. In addition, in both schedules the first four operations in the innermost loop can be executed locally on partial image data structures. The only need for communication is in the exchange of image borders (shadow regions) in the two Gaussian convolution



**Figure 14:** Typical 1000\*554 input image obtained from the Apollo training manual "Apollo Spacecraft & Systems Familiarization" (March 13, 1968). National Aeronautics and Space Administration (NASA), Office of Policy and Plans, NASA History Office. Used by kind permission.



**Figure 15:** Maximum response image obtained after application of the directional filtering program.

operations.

In the first schedule the last two operations in the innermost loop are run in parallel as well. This requires the distributed image `Detected_IM` to be available in full at each node, because it has an access pattern of type 'other' in the back-rotation operation. This can be achieved by executing a gather-to-all operation, which is logically equivalent to a gather operation followed by a broadcast. Finally, a partial maximum response image `Contrast_IM` is calculated on each node, which requires a final gather operation to be executed just before termination of the program.

In the second schedule the last two operations are not executed in parallel. As a result, the intermediate result image `Detected_IM` needs to be gathered to the single

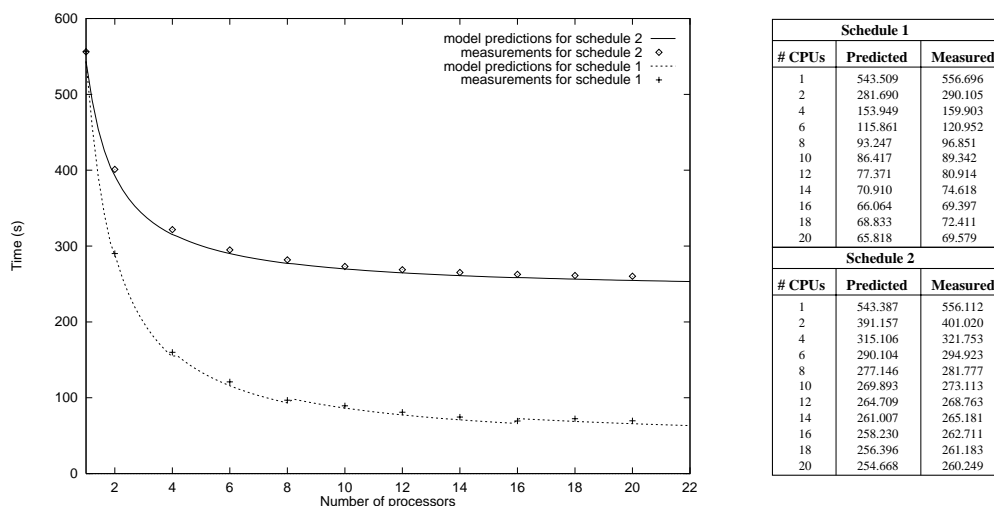
node that produces both the back-rotated image, as well as the complete maximum response image.

As stated before, it is the purpose of the scheduling tool to correctly pick the optimal solution out of the two competing schedules. In the next section we will show, among other things, that the performance models as used in our architecture are powerful enough to allow the scheduler to make such decisions correctly. Note, however, that the schedules as presented here only refer to optimization across library calls. As will also be shown in the next section, intra-operation optimization (such as choosing the optimal mapping of data structures on a logical grid of processing units) can be performed on the basis of our performance models as well.

## 5.4 Performance Evaluation

To initialize the APIPM-based performance models we have performed a small set of benchmarking operations. For each instruction used in the directional filtering program not more than two measurements were performed, i.e. for input (image) sizes of  $200^2$  and  $1000^2$  elements. Based on the measurements, model predictions for each instruction and for each required input size were obtained as indicated in Section 4.3.1. A model for the complete program was obtained by adding the measured performance values for all APIPM instructions executed by the program in sequence.

The benchmarking operations, as well as the directional filtering program were executed on the 24-node homogeneous DAS-cluster (Distributed ASCII Supercomputer [24]) located at the University of Amsterdam. All nodes in the cluster contain a 200 Mhz Pentium Pro with 64 MByte of EDO-RAM, and are connected by a 1.2 Gbit/sec full-duplex Myrinet SAN network. The nodes run the RedHat Linux



**Figure 16:** Comparison of model predictions and measurements for the two program schedules. Results for directional filtering of extended Apollo image of size  $1098 \times 1098$ , and for a parameter subspace including 12 orientations and 4  $(\sigma_v, \sigma_w)$  combinations.

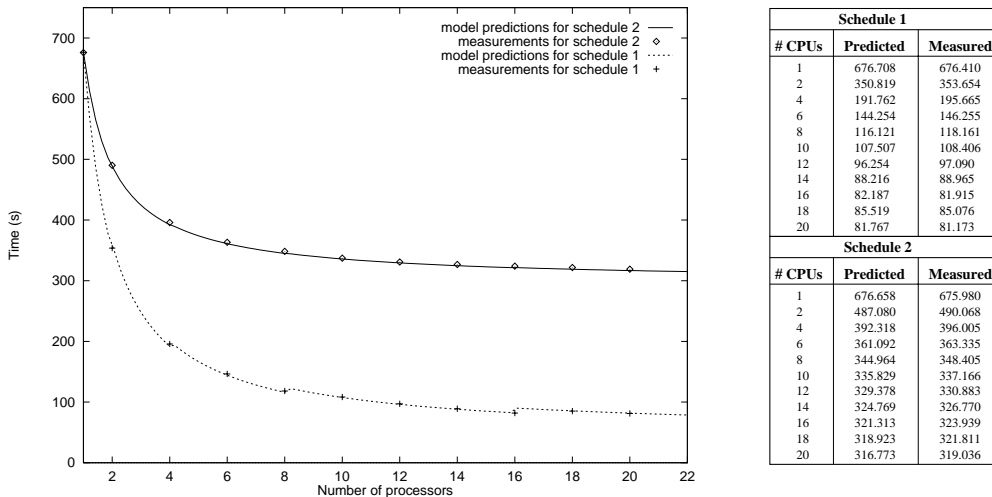
6.2 operating system. At the time of writing, 4 nodes in the system were unusable due to a malfunction in the related network cards. As a consequence performance results are presented only for a system of up to 20 processing units.

Based on intuition alone a programmer would have great difficulty deciding which of the two schedules described in the previous section should be executed. Clearly, a schedule is preferred if the set of operations unique to that schedule is faster than the set of operations unique to another schedule (i.e., not in the set of operations common to both schedules). Hence, for the directional filtering program the schedule in which *all* operations are run in parallel is preferred if:

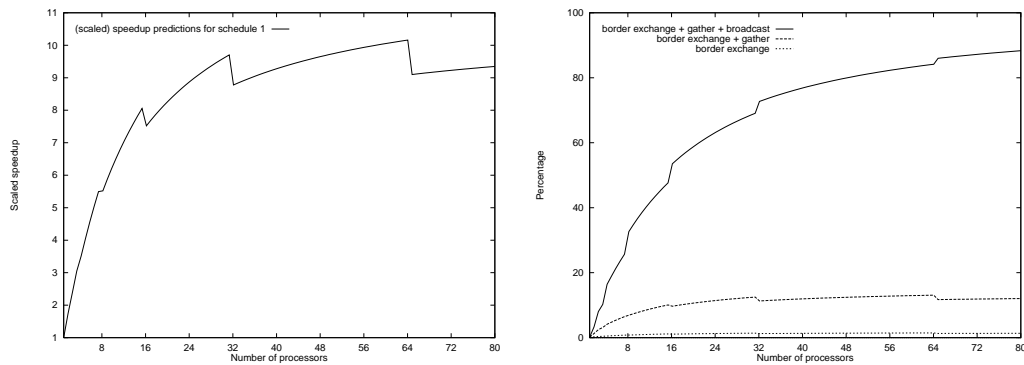
$$\theta\sigma(P_{rotate}(size/N) + P_{max}(size/N) + P_{bcast}(size)) + P_{gather}(size) < \theta\sigma(P_{rotate}(size) + P_{max}(size))$$

where  $N$  denotes the number of processing units, and  $\theta\sigma$  denotes the size of the parameter subspace (note that, for reasons of simplicity, in the equation performance values for library operations are used instead of values for APIPM instructions). For the first schedule the large number of broadcast operations is expected to have the most significant impact on performance. For the second schedule, on the other hand, the many rotations of non-partitioned image data is expected to be costly.

Based on the benchmarking results we are able to decide which schedule is optimal. As shown in Figure 16 (depicting the *complete* execution time of both schedules), our models indicate that the first schedule is always preferred - for any number of processing units. Clearly, broadcasting a full-sized image structure is not as expensive as performing the complete image rotation sequentially on one node. The 'hops' in the graph of schedule 1 are explained by the fact that the broadcast operation is implemented using a spanning binomial tree (SBT), which has a cost related to  $\log N$ .



**Figure 17:** Comparison of predictions and measurements for input image of size  $707 \times 707$ , and for a parameter subspace including 36 orientations and 4  $(\sigma_v, \sigma_w)$  combinations.

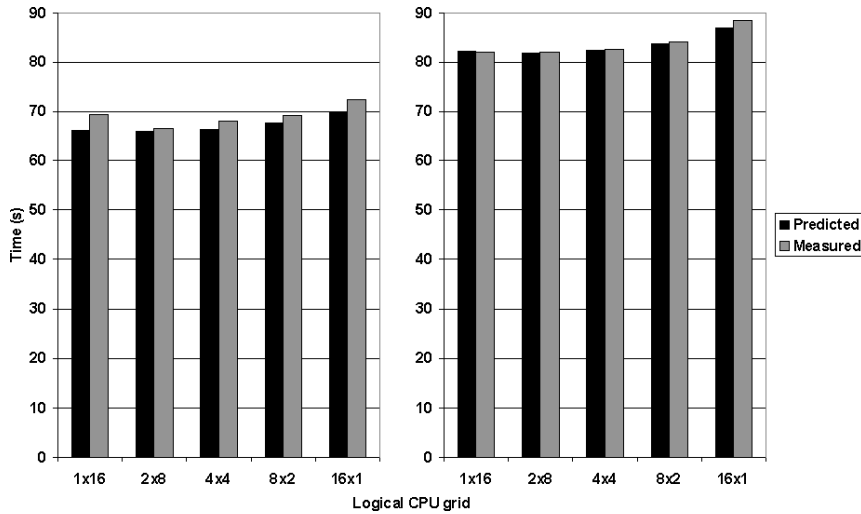


**Figure 18:** Predicted scaled speedup and predicted impact on communication, both for schedule 1.

To test the accuracy of our performance models we have executed the directional filtering program for both schedules. The resulting mean execution times for each run are included in the graph of Figure 16 as well. Error bars are not shown, as the performance of the DAS is quite stable. In most situations measured lower and upper bounds are within 0.5 seconds of the mean execution times. The presented results indicate that the model predictions for both schedules are highly accurate - for any number of processors. Even worst case predictions are within 5.5% of the measured values. It is noteworthy, however, that our models are slightly optimistic in all situations. This is explained by the well-known fact that the performance measured in a benchmarking process tends to be somewhat higher than what is actually obtained in a real application. Similar results obtained for a smaller input image, but for a larger parameter subspace are shown in Figure 17.

Given schedule 1, it is possible to calculate the number of processing units for which the directional filtering program is fastest. As is shown in the left half of Figure 18, our models predict that maximum speedup (10.16) is obtained on 64 nodes; adding more processing units is counterproductive. It can be derived from the graph that the efficiency of the program drops dramatically from 96.5%, 88.2%, and 72.9% for 2, 4, and 8 nodes respectively, to 51.4%, 30.5%, and 15.9% for 16, 32, and 64 nodes respectively. This is due to the large impact of communication, and especially the repeated broadcast. The right half of Figure 18 shows that for 16 processing units the program spends almost half of its time communicating. For 64 processing units 84.1% of the time is lost in all communication steps combined, and 71.1% in broadcasting only. If the image processing researcher would have produced a sequential implementation with rotating filters instead of a rotating image, parallel performance may have been significantly better.

In the results given thus far, we have implicitly assumed that all image data was partitioned in a *row-wise* manner - or, in the terminology of Section 2.2.2, mapped onto a logical  $1 \times N$  grid of processing units. In certain situations it may be beneficial to use a different type of CPU grid. This is because a different data mapping may result in a change in the set of communication operations to be performed (e.g., overlap communication in generalized convolution). Also, due to a difference in the



**Figure 19:** Comparison of predicted and measured execution times for multiple processor grids. Left:  $1098^2$  image, 48  $(\theta, \sigma_v, \sigma_w)$  combinations. Right:  $707^2$  image, 144  $(\theta, \sigma_v, \sigma_w)$  combinations.

memory layout of data communicated between nodes performance gains may be obtained (for example, due to the fact that the communication subsystem can not directly send out data stored noncontiguously in memory, see [21]). In Figure 19 it is shown that our performance models indeed can make a distinction between multiple logical processor grids consisting of 16 nodes. For the two cases shown, a  $2 \times 8$  grid is optimal, which is both predicted by our models as well as measured. For this example application the execution times for each logical grid differ only marginally. We need to stress, however, that for time-critical (i.e., real-time) applications such differences may be relevant indeed. For more results related to this issue, we refer to [21].

## 6 Conclusions and Future Work

In this report we have described a software architecture that allows an image processing researcher to develop parallel applications in a transparent manner. The core of the architecture is formed by an extensive parallel image processing library that has a programming interface identical to that of an existing sequential library. Application of the library is not expected to be considered 'cumbersome', as it fully adheres to the image processing researcher's frame of reference.

In the report we have addressed two important architecture design issues. First, it is shown how maintainability problems, as encountered in similar architectures, are resolved. We have described how code reusability is enhanced by the application of so-called *parallelizable patterns*. Essentially, such patterns define the maximum amount of work that can be executed by a single processing unit without having to communicate to obtain data values that reside elsewhere. As such, the patterns

are applicable both in sequential as well as in parallel implementations of library operations. In addition, we have shown how specialized parallelizable patterns are obtained for typical low level image processing operations. In conclusion, by incorporating specialized parallelizable patterns, we feel that our library is extensible, easily maintainable, and still high in performance.

The second important topic deals with the problem of obtaining efficiency of execution on a range of parallel machines. We have shown that, by applying domain-specific performance models, knowledge is obtained regarding the execution behavior of all library operations. Each operation is implemented such that, based on this knowledge, its execution can be adapted to obtain higher performance. Also, we have shown how the models are applied to perform optimization across library calls. Experiments show that, for a realistic application, our performance models are highly accurate. Given these results we are confident in that the architecture's core forms powerful basis for automatic parallelization and optimization of a wide range of image processing applications.

As an important note we should state that, although all parallelism is hidden inside the library, much of the efficiency of parallel execution is still in the hands of the library user. As shown in the previous section, if a sequential implementation is provided that requires expensive communication operations when run in parallel, program efficiency may be disappointing. Therefore, the library user should be aware of the fact that certain operations are expensive, and should be avoided as much as possible. Any programmer knows that this requirement is not new, however, as a similar requirement holds for sequential implementation as well.

In the near future we will focus our attention on the creation of a fully functional scheduling component. Also, we will extend the set of generic algorithms described in Section 2.2.1. If required, the generalized definition of parallelizable patterns will be adapted accordingly. Finally, we will continue implementing example programs to investigate the implication of parallelization of typical applications, especially in the area of real-time image processing.

In conclusion, our approach to implementing an architecture for parallel image processing resolves many problems often encountered in comparable environments. Most importantly, our work shows that it is possible to ensure architecture maintainability, without having to compromise on the efficiency of execution. Given this result, we strongly believe that our approach is applicable in other research areas as well, especially when the set of typical operations is limited - as is the case in low level image processing.

## References

- [1] A. Alexandrov, M. Ionescu, K. Schauer, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 95–105, Santa Barbara, California, July 1995.

- 
- [2] A. Bar-Noy and S. Kipnis. Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.
- [3] J. Brown and D. Crookes. A High Level Language for Parallel Image Processing. *Image and Vision Computing*, 12(2):67–79, March 1994.
- [4] J. Bruck et al. On the Design and Implementation of Broadcast and Global Combine Operations Using the Postal Model. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):256–265, March 1996.
- [5] M. Crochemore and W. Rytter. Note on Two-Dimensional Pattern Matching by Optimal Parallel Algorithms. In *Proceedings of the Second International Conference on Parallel Image Analysis, ICPIA '92*, pages 100–112, Ube, Japan, December 1992.
- [6] D. Culler et al. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, California, May 1993.
- [7] L.M.J. Florack, B.M. ter Haar Romeny, J.J. Koenderink, and M.A. Viergever. Scale and the differential structure of images. *Image and Vision Computing*, 10(6):376–388, 1992.
- [8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1). Technical report, University of Tennessee, Knoxville, Tennessee, June 1995. Available at <http://www.mpi-forum.org/>.
- [9] J.M. Geusebroek. *Color and Geometrical Structure in Images*. PhD thesis, Faculty of Science, University of Amsterdam The Netherlands, November 2000.
- [10] D.W. Hammerstrom and D.P. Lulich. Image Processing Using One-Dimensional Processor Arrays. *Proceedings of IEEE*, 84(7):1005–1018, 1996.
- [11] L.H. Jamieson, E.J. Delp, and C.-C. Wang. A Software Environment for Parallel Computer Vision. *IEEE Computer*, 25(2):73–75, February 1992.
- [12] Z. Juhasz and D. Crookes. A PVM Implementation of a Portable Parallel Image Processing Library. In *Parallel Virtual Machine - EuroPVM'96, Third European PVM Conference*, pages 188–196, Munich, Germany, 1996.
- [13] D. Koelma, E. Poll, and F. Seinstra. Horus (Release 0.9.2). Technical report, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, February 2000.
- [14] J.J. Koenderink. The structure of images. *Biological Cybernetics*, 50:363–370, 1984.
- [15] M. van der Molen and P. Jonker. A Comparison of Linear Processor Arrays for Image Processing. Technical report, Pattern Recognition Group, Faculty of Applied Sciences, Delft University of Technology, Delft, The Netherlands, 1998.

- 
- [16] C.M. Pancake and D. Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers. *IEEE Computer*, 23(12):13–23, December 1990.
- [17] G.X. Ritter and J.N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Inc, 1996.
- [18] R.H. Saavedra-Barrera, A.J. Smith, and E. Miya. Machine Characterization Based on an Abstract High-Level Language Machine. *IEEE Transactions on Computers*, 38(12):1659–1679, December 1989.
- [19] A. Saoudi and M. Nivat. Optimal Parallel Algorithms for Multidimensional Template Matching and Pattern Matching. In *Proceedings of the Second International Conference on Parallel Image Analysis, ICPIA '92*, pages 240–246, Ube, Japan, December 1992.
- [20] F.J. Seinstra and D. Koelma. Accurate Performance Models of Parallel Low Level Image Processing Operations Based on a Simple Abstract Machine, September 2000. Internal report, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands.
- [21] F.J. Seinstra and D. Koelma. P-3PC: A Simple and Accurate Model of Point-to-Point Communication. Technical Report Series, Vol. 15, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, December 2000.
- [22] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.
- [23] R. Taniguchi et al. Software Platform for Parallel Image Processing and Computer Vision. In *Parallel and Distributed Methods for Image Processing, Proceedings of SPIE*, volume 3166, pages 2–10, 1997.
- [24] Vrije Universiteit. The Distributed ASCI Supercomputer, September 1998. Available at <http://www.cs.vu.nl/das/>.
- [25] G.V. Wilson and P. Lu. *Parallel Programming Using C++*. Scientific and Engineering Computation Series. The MIT Press, 1996.



## **Acknowledgements**

## ISIS reports

This report is in the series of ISIS technical reports. The series editor is Rein van den Boomgaard ([rein@science.uva.nl](mailto:rein@science.uva.nl)). Within this series the following titles are available:

## References

- [1] M. Worring and A.W.M. Smeulders. Content based internet access to paper documents. Technical Report 6, Intelligent Sensory Information Systems Group, University of Amsterdam, December 1998.
- [2] S.D. Olabarriaga, P.R. Pfluger, and A.W.M. Smeulders. Piecewise dm: A locally controllable deformable model. Technical Report 7, Intelligent Sensory Information Systems Group, University of Amsterdam, 1999.
- [3] S.D. Olabarriaga and A.W.M. Smeulders. Interaction in the segmentation of medical images, a survey. Technical Report 8, Intelligent Sensory Information Systems Group, University of Amsterdam, 1999.
- [4] R. v.d. Boomgaard, E.A. Engbers, and A.W.M. Smeulders. Decomposition of separable concave structuring functions. Technical Report 9, Intelligent Sensory Information Systems Group, University of Amsterdam, 1999.
- [5] G. Stijnman and R. v.d. Boomgaard. Background estimation in video sequences. Technical Report 10, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.
- [6] T.V. Pham and M. Worring. Face detection methods: A critical evaluation. Technical Report 11, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.
- [7] Hieu, M. Worring, and R. v.d. Boomgaard. Watersnakes. Technical Report 12, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.
- [8] Hieu, M. Worring, and R. v.d. Boomgaard. Contourtracking. Technical Report 13, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.
- [9] F.J. Seinstra, D. Koelma, and J.M. Geusebroek. A software architecture for user transparent parallel image processing. Technical Report 14, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

You may order copies of the ISIS technical reports from the corresponding author or the series editor. Most of the reports can also be found on the web pages of the ISIS group (<http://www.science.uva.nl/research/isis>).



**Intelligent Sensory Information Systems**  
*University of Amsterdam*  
*The Netherlands*

