ELSEVIER

Parallel Computing 28 (2002) 967–993

PARALLEL COMPUTING

www.elsevier.com/locate/parco

A software architecture for user transparent parallel image processing

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

Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Received 1 December 2000; received in revised form 1 August 2001; accepted 1 December 2001

Abstract

This paper describes a software architecture that allows image processing researchers to develop parallel applications in a transparent manner. The architecture's main component is an extensive library of data parallel low level image operations capable of running on homogeneous distributed memory MIMD-style multicomputers. Since the library has an application programming interface identical to that of an existing sequential library, all parallelism is completely hidden from the user.

The first part of the paper discusses implementation aspects of the parallel library, and shows 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 extensive code redundancy is avoided. The second part of the paper describes the application of performance models to ensure efficiency of execution on all target platforms. Experiments show that for a realistic 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 imaging software.

Keywords: Data parallel image processing library; Parallelizable patterns; Abstract parallel image processing machine; Performance modeling; Homogeneous MIMD-style multicomputers

* Corresponding author.

E-mail addresses: fjseins@science.uva.nl (F.J. Seinstra), koelma@science.uva.nl (D. Koelma), geusebroek@science.uva.nl (J.M. Geusebroek).

0167-8191/02/$ - see front matter
PII: S0167-8191(02)00103-5

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 [4,7,23] and dedicated parallel architectures [8,12,17] 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. Unfortunately, the fundamental problem of automatic and optimal partitioning remains unsolved. Another possibility is to design a parallel programming language, either general purpose [20,30] or aimed at image processing specifically [5,28]. However, in accordance with the remarks made in [18], 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.

A 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 (for example, see [13,14,16,27,29]). In many cases, efficiency of execution on a range of parallel machines is obtained by hard-coding a number of different implementations for each operation, one for each platform. We feel that this solution to *intra-operation optimization* requires too much implementation effort, and is impossible to maintain on the long term. Also, 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 strife 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 paper we give an overview of the complete architecture 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 our approach of obtaining efficiency of execution on a range of computers by the application of abstract machine-based performance models. In Section

5 model predictions are compared with results obtained on a machine from the class of target platforms. 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

The data parallel low level image processing library should adhere 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, homogeneous MIMD-style multicomputers. 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++ in combination with MPI [10] is most appropriate to ensure portability to all target machines. In addition, no assumptions should be made about a specific interconnection network topology. All nodes in 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. Efficiency, in this respect, refers to each separate library operation, as well as to multiple operations applied in sequence.

2.2. Architecture overview

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

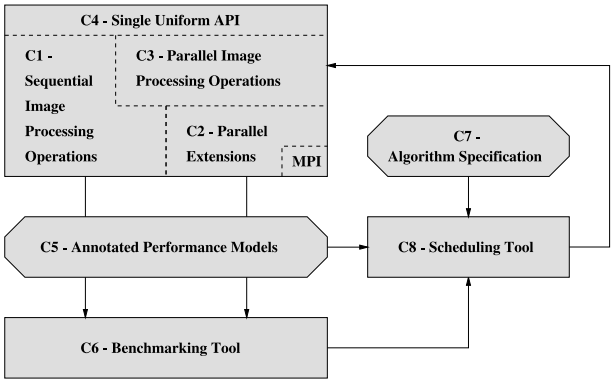


Fig. 1. Architecture overview.

C1. Sequential image processing operations: The first component contains a large set of sequential operations typically used by image processing researchers. As recognized in, for example, Image Algebra [21], a small set of *operation classes* can be identified that covers the bulk of all commonly applied image processing operations. We have implemented each operation class as a *generic algorithm*, using the C++ *function template mechanism* [26]. Each operation that maps onto the functionality as provided by such algorithm is implemented by instantiating the generic algorithm with the proper parameters, including the function to be applied to the individual data elements. In our current 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.

C2. Parallel extensions: Three classes of routines are implemented (using MPI 1.1 [10]) that introduce the parallelism into the library: (1) data *partitioning* routines, to map data structures onto a logical grid of processing units of up to 3 dimensions, (2) data *distribution* and *redistribution* routines, to scatter, gather, broadcast, and redistribute data structures, and (3) routines for *overlap communication*, to exchange *shadow regions*, such as image borders in neighborhood operations.

C3. 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 Fig. 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.

C4. Single uniform API: The image processing library is provided with an application programming interface identical to that of an existing sequential library [15]. As such, the threshold for applying the library is low, as all parallelism is fully transparent to the user.

C5. Annotated performance models: In our library we provide only one parallel implementation of each generic algorithm. To ensure efficiency of execution on all

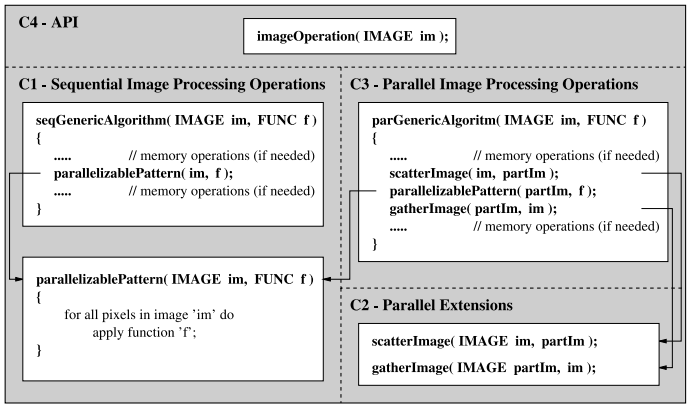


Fig. 2. Relationships between library components C1–C4 (note: actual code may differ).

target platforms, the parallel generic algorithms are implemented such that they are capable of adapting to the performance characteristics of a specific machine. To make these characteristics explicit, each library operation is annotated with a performance model, as described extensively in Section 4.

C6. Benchmarking tool: For a specific machine, performance values for the model parameters are obtained by running a set of *benchmarking* operations. Based on the benchmarking results intra-operation optimization can be performed automatically, fully transparent to the user.

C7. Algorithm specification: Besides intra-operation optimization, optimization across library calls can be performed 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 assume that a complete algorithm specification is provided in addition to the program itself.

C8. Scheduling tool: Once the performance models, the benchmarking results, and the algorithm specification are available, a scheduling component is applied to find an optimal solution for the application at hand. It is the task of the scheduler to remove redundant communication steps, and to make optimization decisions regarding: (1) the logical processor grid to map data structures onto (i.e., the actual domain decomposition), (2) the number of processing units, and (3) the routing pattern for the distribution of data [25]. In the library implementation of each parallel generic algorithm, requests for scheduling results are performed in order to correctly execute the optimizations decided upon. Whether scheduling results are static only, or should be generated and updated dynamically is still an important future research issue.

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 what we refer to as *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

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

where $\mathbf{o} = (o_1, o_2, \ldots, o_n)$ represents the *origin* of the image, and k_i represents the extent of the domain in the ith 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 a common data type (e.g., int, or float), or a complex number. 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., $\mathbf{a} : \mathbf{X} \to \mathbb{F}$) is a shorthand notation for

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

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 $\mathbf{a}_{\mathrm{p}_i}$; its domain is denoted by $\mathbf{X}_{\mathrm{p}_i}$. As data spreading cannot result in a loss of data, for each image $\mathbf{a} \in \mathbb{F}^{\mathbf{X}}$ distributed over n processors, the union of the domains of all its partial images equals the domain of **a**.

The n partial images related to **a** together form one logical structure, referred to as a *distributed image*. A distributed image is denoted by $\mathbf{a}_{\mathrm{d}}$, and differs from a partial image in that it does not reside as a physical structure in the memory of one processing unit (unless it is formed by one partial image only). A distributed image's domain $\mathbf{X}_{\mathrm{d}}$ 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 Fig. 3).

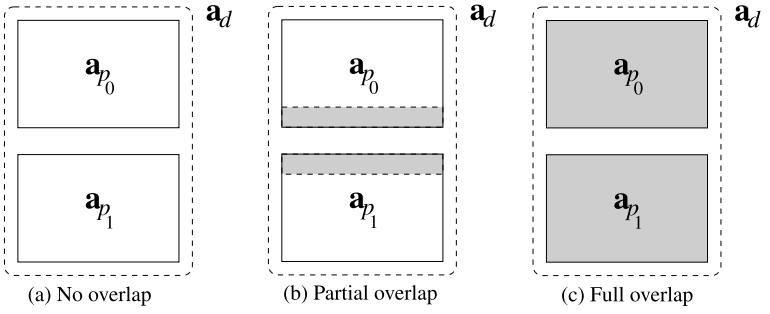


Fig. 3. Three examples of a distributed image $\mathbf{a}_{\mathrm{d}}$ comprising of two partial images, $\mathbf{a}_{\mathrm{p}_0}$ and $\mathbf{a}_{\mathrm{p}_1}$. The gray areas represent domain overlap; the white areas represent the unique domain parts.

Essentially, it is possible for each node to perform operations on partial images *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 important to avoid inconsistencies in distributed image data.

3.2. Parallelizable patterns

As stated in Section 2.2, 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 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 communication. In other words, in a parallelizable pattern all internal data accesses refer to data *local* to the node that executes the operation. In the following we give a generalized description of parallelizable patterns, and show their usage 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 $i \in \{1, 2, \ldots, n\}$. 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, all of which are processed in a loop of n iterations. Each task location $\mathbf{x}_i$ is related 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 $i \in \{1, 2, \ldots, n\}$) 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 $i \in \{1, 2, \ldots, n\}$) 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 $i \in \{1, 2, \ldots, n\}$) no data points are accessed other than those within the *neighborhood* of $\mathbf{x}_i$. As an example, the 5×3 neighborhood of a point $\mathbf{x} = (x_1, x_2) \in \mathbf{X} \subset \mathbf{Y}$ is given by

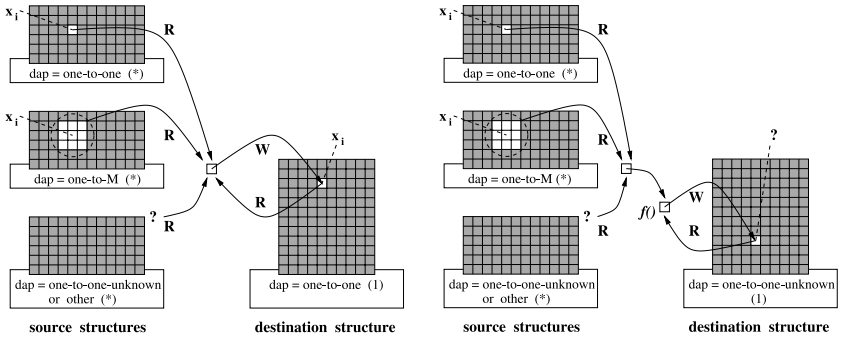
$$N(\mathbf{x}) = \{\mathbf{y} \in \mathbf{Y} : \mathbf{y} = (x_1 \pm j, x_2 \pm k), \quad j \in \{0, 1, 2\}, \quad k \in \{0, 1\}\}. \tag{3}$$

- *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' type. This is because—by definition—in each task only one data point is accessed in the destination structure.

Fig. 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 the access pattern type related to the destination structure is of type 'one-to-one-unknown'. The two parallelizable patterns also 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 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 patterns give a generalization of a large set of sequential image operations, e.g. all operations in component C1 in Section 2.2. It should be noted that the two types do not capture the complete set of all possible 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. Still, all generic algorithms that do map onto the given generalization are



(a) Parallelizable pattern, type 1. (b) Parallelizable pattern, type 2.

Fig. 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.

applicable in the process of 'parallelization by concatenation of library operations', as introduced in Section 2.2

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 $X \subset \mathrm{TL}$, 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 not 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

$$k = \Gamma\mathbf{a} = \Gamma_{\mathbf{x}\in\mathbf{X}}\mathbf{a}(\mathbf{x}) = \Gamma_{i=1}^{n}\mathbf{a}(x_i) = \mathbf{a}(x_1)\gamma\mathbf{a}(x_2)\gamma\cdots\gamma\mathbf{a}(x_n), \tag{4}$$

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

Fig. 5 shows that at least two 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 sec-

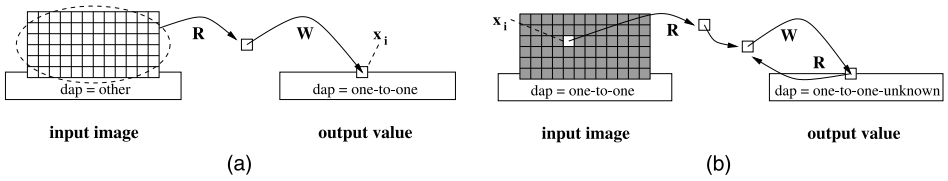


Fig. 5. Sequential reduction—two implementations.

ond implementation, the operation requires n steps. In each step, one data point in **a** is read and combined with the current value of k.

The two implementations both constitute a specialization of one of the parallelizable pattern types 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 processor. 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 is easily run in parallel, as TL contains all locations in **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'. As a result, a parallel implementation of the generic reduction operation follows directly from the generalization of Section 3.2.2 (see Fig. 6).

3.2.4. Example: parallel generalized convolution

A generalized convolution on image **a**, producing image **c**, given kernel **t**, is defined as follows:

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

$$\mathbf{c} = \mathbf{a} \circledcirc \mathbf{t} = \{(\mathbf{x}, \mathbf{c}(\mathbf{x})) : \mathbf{c}(\mathbf{x}) = \Gamma_{\mathbf{y}\in\mathbf{Y}} \mathbf{a}(\mathbf{x}+\mathbf{y}) \bigcirc \mathbf{t}(\mathbf{y}), \mathbf{x} \in \mathbf{X}\}, \tag{5}$$

where $\bigcirc$ and γ are binary operations on $\mathbb{F}$, and γ is associative and commutative. The extent of the domain in the ith dimension of kernel **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

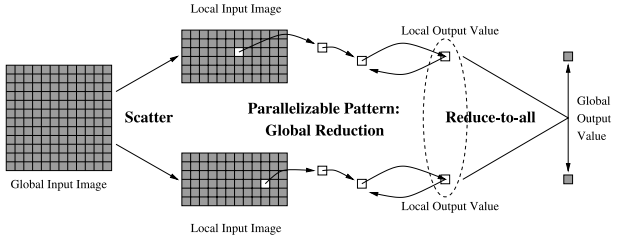


Fig. 6. Example reduce-to-all operation executed on two processing units.

Table 1
Example generalized convolution instantiations

Kernel operation	$\bigcirc$	γ
Convolution	Multiplication	Addition
Dilation	Addition	Maximum
Erosion	Addition	Minimum

the operation is presented in Fig. 7. Again, set TL is implicit, and contains all pixel positions in the input image or the output image.

When comparing Fig. 4(a) to 7(a) it may seem that the operation directly constitutes a parallelizable pattern. Fig. 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 robust and, in general, also faster. For parallel implementation this strategy has the additional advantage that the generic algorithm 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. As each local scratch image has a one-to-M access pattern, an overlapping scatter of the global input image is required. As shown in Fig. 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 Fig. 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.

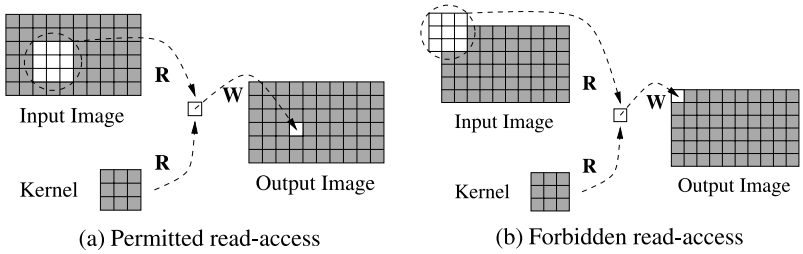


Fig. 7. Sequential generalized convolution. Does not represent a parallelizable pattern, as read accesses outside the domain of the input image are possible (see (b)).

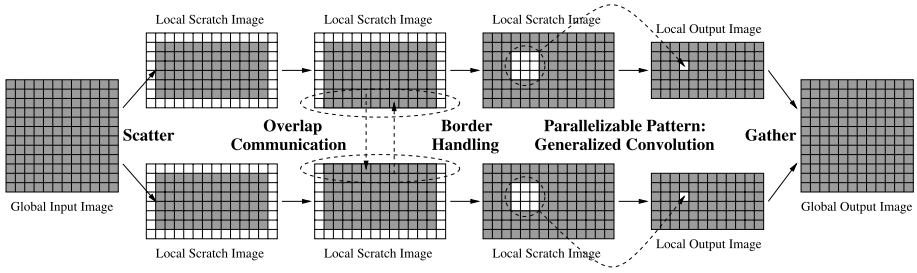


Fig. 8. Example kernel operation performed using two processing units (simplified).

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. It is the task of the scheduler of Section 2.2 to make additional optimization decisions, e.g. relating to the type of domain decomposition. In our architecture we rely on performance models for the scheduler to perform this task correctly.

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 data parallel image processing operations. 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.

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:

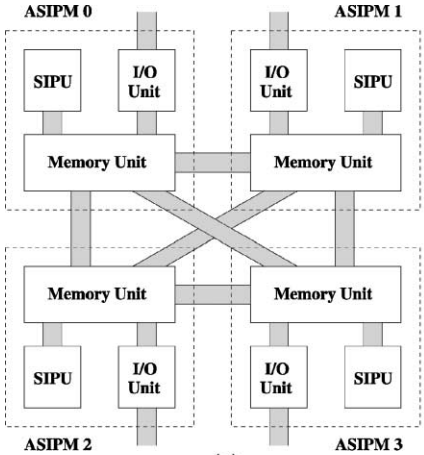
1. *Simplicity:* The more detailed a model, the less manageable it is and the more expense will go into obtaining its performance measures. To reduce the costs of static or run-time model evaluation the number of parameters in the model should be kept to a minimum.
2. *Accuracy:* To make sure the scheduler 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:* In our implementations we make sure that the library is portable to all machines in the class homogeneous distributed memory MIMD-style multicomputers. Consequently, the related performance models must be applicable to all such platforms as well.

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

4.2. Abstract parallel image processing machine

The design of the annotated performance models is based on the definition of an abstract parallel image processing machine (or APIPM, see Fig. 9(a)). An APIPM consists of one or more identical abstract sequential image processing machines (ASIPMs), each consisting of four related components: (1) a *sequential image processing unit* (SIPU), 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 between the memory unit and external sensing or storage devices, and (4) *data channels*, the means by which data is transported between ASIPM units and external devices. In a complete APIPM the memory unit of each ASIPM is connected with those of all other ASIPMs.

The APIPM instruction set (Fig. 9(b)) consists of four classes of operations: (1) *generic image instructions*, i.e. the specialized parallelizable patterns of Section 3.2, (2) *memory instructions*, for allocation and copying of (image) data, (3) *I/O instructions*, for transporting data between memory and external devices, and (4) *communication instructions*, for exchanging data among ASIPM units. For simplicity, in Fig. 9(b) the operands for each opcode are left out.



opcode	generic image instructions
UPOP	unary pixel operation
BPOPC	binary pixel operation (arg: constant)
BPOPI	binary pixel operation (arg: image)
REDUCOP	global reduction operation
NEIGHOP	neighborhood operation
GCONVOP	generalized convolution
GEOMAT	geometric transformation (arg: matrix)
GEOROI	geometric transformation (region of interest)
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	**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

(b)

Fig. 9. APIPM comprising of four ASIPMs, and related instruction set.

The description of the APIPM reflects a state-of-the-art homogeneous distributed memory MIMD-style multicomputer. It differs from a general purpose machine in that each SIPU is designed for imaging tasks only. Although a fully connected network is often not present, we still have included one in the APIPM. This is because in most multicomputer systems communication is based on circuit-switched message routing, which makes a network *virtually* fully connected.

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 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.

4.3. APIPM-based performance models

All library operations are assumed to be implemented by concatenation of APIPM instructions only. Also, we assume that the execution time of each library

operation can be partitioned into *independent* intervals, each corresponding to the cost of a single APIPM instruction. The performance of a library operation is obtained by adding the run-time costs of all APIPM instructions used.

This idea is formalized as follows. Let $\mathbf{I} = \{I_1, I_2, \ldots, I_n\}$ be the APIPM instruction set. Let $\mathbf{P} = \{P_{I_1}, P_{I_2}, \ldots, P_{I_n}\}$ 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 $\mathbf{L} = \{\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_m\}$ be the set of all m operations implemented using instructions in $\mathbf{I}$ only. For all library operations $\mathbf{L}_x$ $(x \in \{1, \ldots, m\})$ we define $\mathbf{L}_x = \{I_1, I_2, \ldots, I_n\}$, in combination with the total number of occurrences (or *count*) of each APIPM instruction in $\mathbf{L}_x$: $\mathbf{C}_x = \{C_{I_1,x}, C_{I_2,x}, \ldots, C_{I_n,x}\}$. The expected total execution time of each 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 $I_i('\text{int}')$ and $I_i('\text{float}')$). Also, a 'data-input-size' parameter is required for most performance values in $\mathbf{P}$ (e.g., giving $P_{I_i(\text{datatype})}(\text{size})$). Note that the choice of model parameters is dependent on the actual library implementation of each APIPM instruction. For a complete overview we refer to [24].

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. For this reason we take a *semi-empirical* modeling approach. This means that benchmarking is performed for *multiple* data input sizes. To capture non-linear performance growth, for each instruction, between each pair of measured performance values, performance growth is then assumed to be piecewise linear. As benchmarking is outside the scope of this paper, no additional details will be given here.

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. Firstly, this is because an accurate prediction of the *end-to-end communication time* usually cannot be obtained by considering the time a processor is busy executing a `SEND` or a `RECV` instruction alone. Secondly, in its current form the model does not closely match the capabilities of the communication instructions as defined in MPI. Most notably, the impact of a message's memory layout on communication costs is not incorporated in the model. This is an important point, as one of the tasks of the scheduler of Section 2.2 is to make decisions regarding the domain decomposition of a given application. Depending on the type of such domain decomposition, it may be necessary to communicate data stored non-contiguously in memory. Using MPI's *derived*

datatype mechanism it is possible to send such data in a single communication step. As was shown in the recent work of Prieto et al. [19], knowledge of a message's memory layout is important, as non-unit-stride memory access may have a severe impact on performance due to caching. Also, the MPI operations may handle the transmission of non-contiguous data differently from contiguous blocks, possibly causing additional overheads due to the packing of data into a contiguous buffer.

For these reasons 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 [3,6], LogP [9], and LogGP [1]). As we have shown in [25], the model is capable of modeling many essential communication patterns as used in data parallel image processing applications. In addition, and in contrast to the models mentioned above, it is also capable of accurately predicting the communication costs related to any type of domain decomposition.

The model introduces the notion of the *three paths of communication*, and assumes 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 sender (i.e., the time required for executing the `SEND` instruction).
- T_{recv}: the cost related to the communication path at the receiver (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 $t_{\text{cs}}, t_{\text{cr}}$, and t_{cf} (for the sender, receiver, and full path respectively). In addition, for each communication path we assume an 'additional time' ($t_{\text{as}}, t_{\text{ar}}$, and t_{af} respectively) which is a function of the number of bytes transmitted. To capture differences in the sending of contiguous and non-contiguous 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 Fig. 10) involved in the transmission of a message containing n bytes are then given by:

$$\begin{aligned} T_{\text{send},M}(n) &= t_{\text{cs}} + t_{as,M}(n), \\ T_{\text{recv},M}(n) &= t_{\text{cr}} + t_{ar,M}(n), \\ T_{\text{full},M}(n) &= t_{\text{cf}} + t_{af,M}(n), \end{aligned} \tag{6}$$

where $\mathtt{M} \in \{\text{cc},\ \text{cn},\ \text{nc},\ \text{nn}\}$. 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

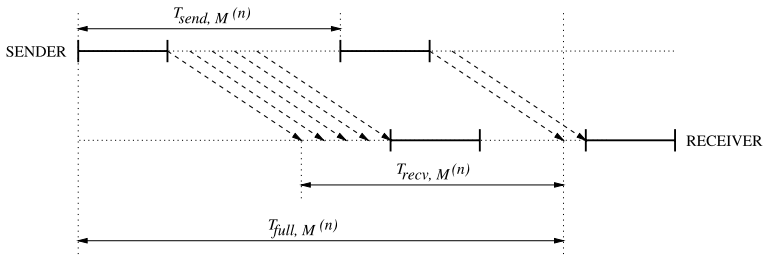


Fig. 10. Communication according to the P-3PC Model. In this particular example a single node transmits two separate messages to the same receiver in sequence. The three communication paths are indicated according to the definitions in the text.

accepted non-contiguously by the receiver. For a complete description of the P-3PC model we refer to [25].

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 [22], 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, implemented using our library, is executed in parallel. The application is highly relevant as it in-

corporates all generic algorithms as referred to in Section 2.2. First, a description is given of the underlying algorithm. Next, both a straightforward sequential implementation as well as the related parallel implementation are discussed. Finally, measured results are compared with APIPM model predictions.

5.1. Detection of lines in images

As discussed in [11], the problem of detecting (curved) lines in images is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_v in the line direction, and differentiation scale σ_w perpendicular to the line, given by

$$r''(x, y, \sigma_v, \sigma_w, \theta) = \sigma_v \sigma_w |f_{ww}^{\sigma_v, \sigma_w, \theta}| \frac{1}{b^{\sigma_v, \sigma_w, \theta}}. \tag{7}$$

When the filter is correctly aligned with a line in the image, and σ_v, σ_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:

$$R(x, y) = \arg \max_{\sigma_v, \sigma_w, \theta} r''(x, y, \sigma_v, \sigma_w, \theta). \tag{8}$$

This directional filtering problem can be implemented sequentially in many different ways. For each orientation θ it is possible to create a new filter based on σ_w and σ_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 accessed according to the size of the neighborhood as well as 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 do not claim, however, that this implementation provides optimal performance when executed either sequentially or in parallel.

The main body of the sequential implementation is presented in pseudocode in Listing 1. The program starts by rotating the original input image for a given orientation θ. In addition, for all (σ_v, σ_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 `Filteredl_IM` and `Filtered2_IM`, respectively) are produced by executing two generalized convolutions, each with the appropriate parameters. For cost effectiveness the Gaussian convolutions are performed by applying two 1-dimensional filters in both cases. Next, the result of Eq. 7 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 input image, and the maximum response image is obtained.

```
FOR all orientations θ DO
    Rotated_IM = GeometricOp(Original_IM, "rotate", θ);
    FOR all smoothing scales σ_v DO
        FOR all differentiation scales σ_w DO
            Filtered1_IM = GenConvOp(Rotated_IM, "gauss", σ_w, σ_v, 2, 0);
            Filtered2_IM = GenConvOp(Rotated_IM, "gauss", σ_w, σ_v, 0, 0);
            Detected_IM = BinPixImArgOp(Filtered1_IM, "absdiv", Filtered2_IM);
            Detected_IM = BinPixCnstArgOp(Detected_IM, "mul", σ_v * σ_w);
            BackRotated_IM = GeometricOp(Detected_IM, "rotate", −θ);
            Contrast_IM = BinPixImArgOp(Contrast_IM, "max", BackRotated_IM);
```

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

Fig. 11(a) 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 Fig. 11(b). 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.

5.2. Parallel execution

As all parallelization issues are shielded from the user, the pseudocode 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 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 a data parallel manner, using all available processors. 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

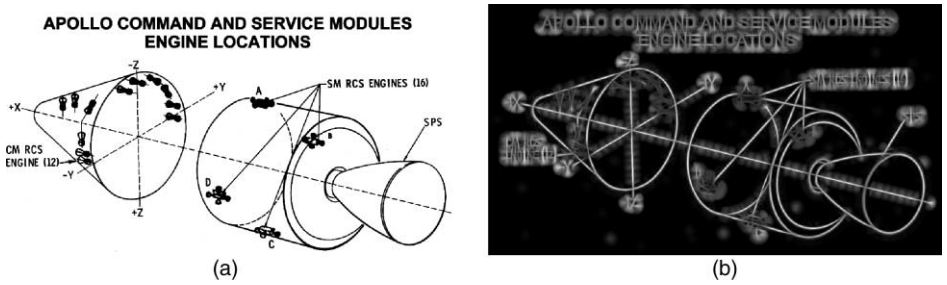


Fig. 11. (a) Typical 1000 × 554 input image obtained from the Apollo training manual "Apollo Spacecraft & Systems Familiarization" (March 13, 1968). National Aeronautics and Space Administration (NASA), Office of Policy and Plans, NASA History Office. Used by kind permission. (b) Maximum response image obtained after application of the directional filtering program.

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 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.

5.3. 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 sizes of 200^2 and 1000^2 elements. Model predictions for each instruction and for each required input size were obtained as indicated in Section 4.3.

The benchmarking operations, as well as the directional filtering program were executed on the 24-node homogeneous DAS-cluster (Distributed ASCI Supercomputer [2]) 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 6.2 operating system. At the time of writing, four nodes in the system were unusable. As a consequence, performance results are presented only for a system of up to 20 processing units.

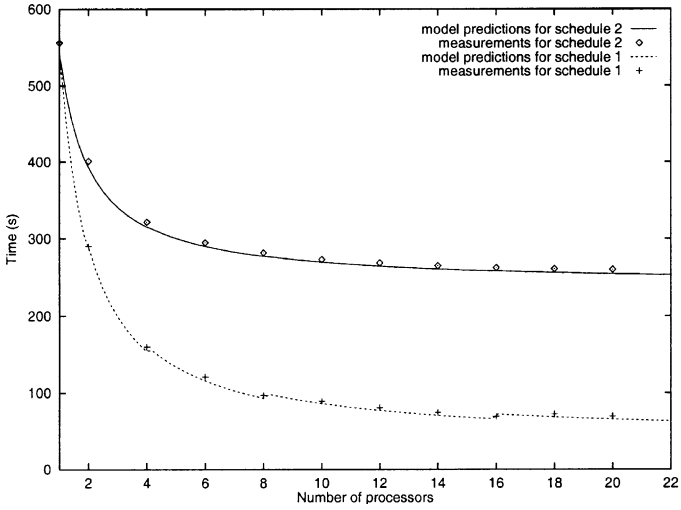
Based on intuition alone a programmer would have great difficulty deciding which of the two schedules described in the previous section should be executed. Clearly, a schedule is preferred if the set of operations unique to that schedule is faster than the set of operations unique to another schedule. Hence, for the directional filtering program the first schedule is preferred if:

$$\theta\sigma(P_{\text{rotate}}(\text{size}/N) + P_{\max}(\text{size}/N) + P_{\text{bcast}}(\text{size})) + P_{\text{gather}}(\text{size}) < \theta\sigma(P_{\text{rotate}}(\text{size}) + P_{\max}(\text{size})), \quad (9)$$

where N denotes the number of nodes, and $\theta\sigma$ the size of the parameter subspace. For the first schedule the large number of broadcasts is expected to have a significant impact on performance. For the second schedule the many rotations of non-partitioned image data is expected to be costly.

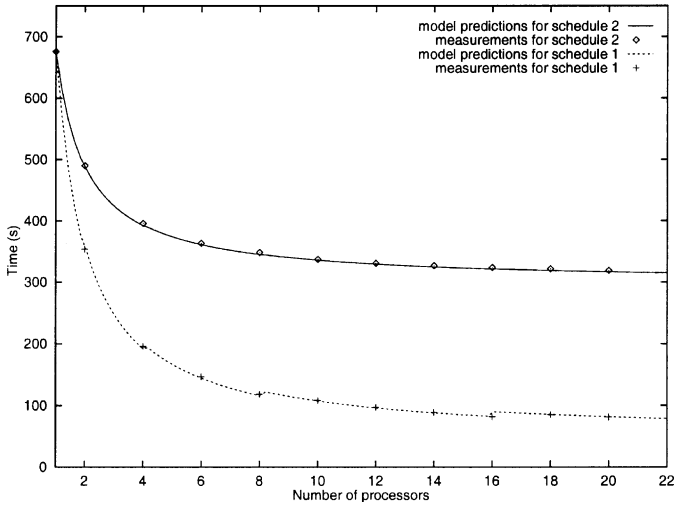
Based on the benchmarking results we are able to decide which schedule is optimal. As shown in Fig. 12 (depicting the *complete* execution time of both schedules), our models indicate that the first schedule is always preferred—for any number of processing units. Clearly, broadcasting a full-sized image 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 $\log N$.

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 Fig. 12 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 s 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



Schedule 1		
# CPUs	Predicted	Measured
1	543.509	556.696
2	281.690	290.105
4	153.949	159.903
6	115.861	120.952
8	93.247	96.851
10	86.417	89.342
12	77.371	80.914
14	70.910	74.618
16	66.064	69.397
18	68.833	72.411
20	65.818	69.579
Schedule 2		
# CPUs	Predicted	Measured
1	543.387	556.112
2	391.157	401.020
4	315.106	321.753
6	290.104	294.923
8	277.146	281.777
10	269.893	273.113
12	264.709	268.763
14	261.007	265.181
16	258.230	262.711
18	256.396	261.183
20	254.668	260.249

Fig. 12. Comparison of model predictions and measurements for the two program schedules. Results for directional filtering of extended Apollo image of size 1098×1098, and for a parameter subspace including 12 orientations and 4 (σ_v, σ_w) combinations.



Schedule 1		
# CPUs	Predicted	Measured
1	676.708	676.410
2	350.819	353.654
4	191.762	195.665
6	144.254	146.255
8	116.121	118.161
10	107.507	108.406
12	96.254	97.090
14	88.216	88.965
16	82.187	81.915
18	85.519	85.076
20	81.767	81.173
Schedule 2		
# CPUs	Predicted	Measured
1	676.658	675.980
2	487.080	490.068
4	392.318	396.005
6	361.092	363.335
8	344.964	348.405
10	335.829	337.166
12	329.378	330.883
14	324.769	326.770
16	321.313	323.939
18	318.923	321.811
20	316.773	319.036

Fig. 13. Comparison of predictions and measurements for input image of size 707×707, and for a parameter subspace including 36 orientations and 4 (σ_v, σ_w) combinations.

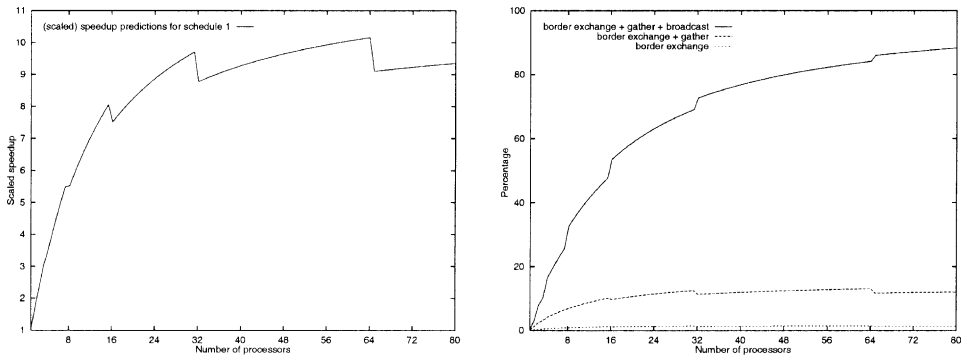


Fig. 14. Predicted scaled speedup and predicted impact on communication, both for schedule 1.

a real application. Similar results obtained for a smaller input image, but for a larger parameter subspace are shown in Fig. 13.

Given schedule 1, we can calculate the number of nodes for which the program is fastest. As is shown in the left half of Fig. 14, our models predict that maximum speedup (10.16) is obtained on 64 nodes; adding more processors 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 Fig. 14 shows that for 16 nodes the program spends almost half of its time communicating. For 64 nodes 84.1% of the time is lost in all communication steps combined, and

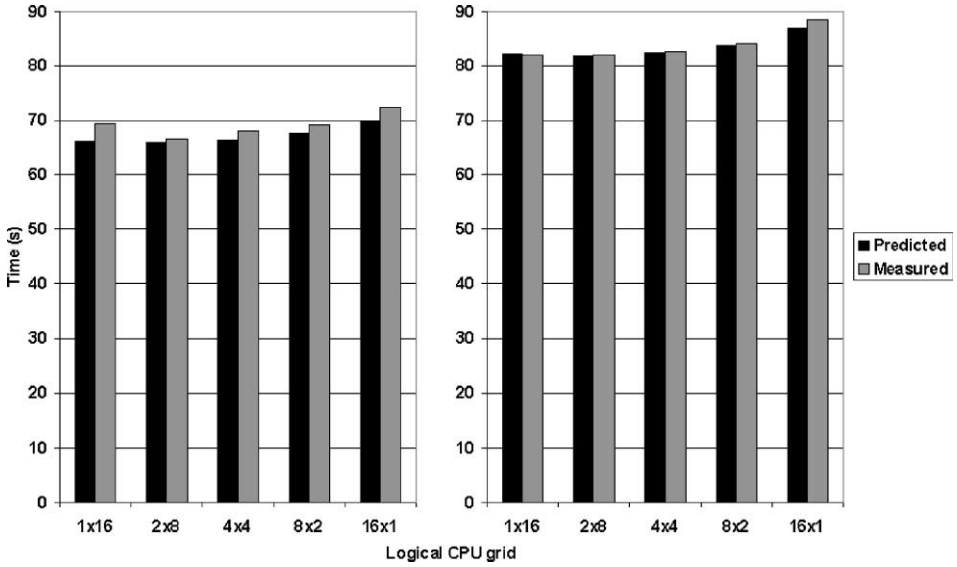


Fig. 15. Comparison of predicted and measured execution times for multiple processor grids. Left: 1098^2 image, 48 $(\theta, \sigma_v, \sigma_w)$ combinations. Right: 707^2 image, 144 $(\theta, \sigma_v, \sigma_w)$ combinations.

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, 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 memory layout of data communicated between nodes performance gains may be obtained (see Section 4.4). In Fig. 15 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 2×8 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 [25].

6. Conclusions and future work

In this paper 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 data 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 paper 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 in component C1 of Section 2.2. 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. Schauser, 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), Santa Barbara, CA, July, 1995, pp. 95–105.
[2] H.E. Bal et al., The distributed ASCI supercomputer project, Operating Systems Review 34 (4) (2000) 76–96.
[3] A. Bar-Noy, S. Kipnis, Designing broadcasting algorithms in the postal model for message-passing systems, Mathematical Systems Theory 27 (5) (1994) 431–452.
[4] S. Boussakta, A novel method for parallel image processing applications, Journal of Systems Architecture 45 (1999) 825–839.
[5] J. Brown, D. Crookes, A high level language for parallel image processing, Image and Vision Computing 12 (2) (1994) 67–79.
[6] 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) (1996) 256–265.
[7] M. Crochemore, 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, Ube, Japan, December, 1992, pp. 100–112.
[8] D. Crookes, Architectures for high performance image processing: the future, Journal of Systems Architecture 45 (1999) 739–748.
[9] 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, San Diego, California, May, 1993, pp. 1–12.
[10] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard (version 1.1). Technical report, University of Tennesser, June 1995. See <http:/ /www.mpi-forum.org/>.
[11] J.M. Geusebroek, A.W.M. Smeulders, H. Geerts, A minimum cost approach for segmenting networks of lines, International Journal of Computer Vision 43 (2) (2001) 99–111.
[12] D.W. Hammerstrom, D.P. Lulich, Image processing using one-dimensional processor arrays, Proceedings of IEEE 84 (7) (1996) 1005–1018.
[13] L.H. Jamieson, E.J. Delp, C.-C. Wang, A software environment for parallel computer vision, IEEE Computer 25 (2) (1992) 73–75.
[14] Z. Juhasz, D. Crookes, A PVM implementation of a portable parallel image processing library, in: EuroPVM' 96, Munich, Germany, 1996, pp. 188–196.
[15] D. Koelma, E. Poll, F. Seinstra, Horus (Release 0.9.2). Technical report, ISIS, Faculty of Science, University of Amsterdam, The Netherlands, February 2000.
[16] C.-K. Lee, M. Hamdi, Parallel image processing applications on a network of workstations, Parallel Computing 21 (1) (1995) 137–160.
[17] M. van der Molen, P. Jonker, A Comparison of Linear Processor Arrays for Image Processing, Technical report, Pattern Recognition Group, Faculty of Applied Sciences, Delft University of Technology, The Netherlands, 1998.
[18] C.M. Pancake, D. Bergmark, Do parallel languages respond to the needs of scientific programmers, IEEE Computer 23 (12) (1990) 13–23.
[19] M. Prieto, I.M. Llorente, F. Tirado, Data locality exploitation in the decomposition of regular domain problems, IEEE Transactions on Parallel and Distributed Systems 11 (11) (2000) 1141–1149.
[20] K. van Reeuwijk, A.J.C. van Gemund, H.J. Sips, Spar: a programming language for semi-automatic compilation of parallel programs, Concurrency: Practice and Experience 9 (11) (1997) 1193–1205.
[21] G.X. Ritter, J.N. Wilson, Handbook of Computer Vision Algorithms in Image Algebra, CRC Press Inc, BocaRaton, 1996.
[22] R.H. Saavedra-Barrera, A.J. Smith, E. Miya, Machine characterization based on an abstract high-level language machine, IEEE Transactions on Computers 38 (12) (1989) 1659–1679.

[23] A. Saoudi, 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, Ube, Japan, December, 1992, pp. 240–246.
[24] F.J. Seinstra, D. Koelma, Accurate Performance Models of Parallel Low Level Image Processing Operations Based on a Simple Abstract Machine. Technical report, ISIS, Faculty of Science, University of Amsterdam, The Netherlands, September, 2000.
[25] F.J. Seinstra, D. Koelma, Incorporating memory layout in the modeling of message passing programs, in: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (EURO ICRO-PDP 2002), Las Palmas de Gran Canaria, Canary Islands, Spain, January, 2002, pp. 293–300.
[26] B. Stroustrup, The C++ Programming Language, third ed., Addison-Wesley, UK, 1997.
[27] R. Taniguchi et al., Software platform for parallel image processing and computer vision, in: Parallel and Distributed Methods for age Processing, Proceedings of SPIE, vol. 3166, 1997, pp. 2–10.
[28] J.A. Webb, Steps toward architecture independent image processing, IEEE Computer 25 (2) (1992) 21–31.
[29] F. Weil, L.H. Jamieson, E.J. Delp, Dynamic intelligent scheduling and control of reconfigurable parallel architectures for computer vision/image processing, Journal of Parallel and Distributed Computing 13 (1991) 273–285.
[30] G.V. Wilson, P. Lu, Parallel Programming Using C++. Scientific and Engineering Computation Series, The MIT Press, 1996.