User Transparent Parallel Image Processing
Seinstra, F.J.

Citation for published version (APA):
Chapter 3

Parallelizable Patterns in Low Level Image Processing Algorithms*

"One gets to the heart of the matter by a series of experiences in the same pattern, but in different colors."

Robert Graves (1895 - 1985)

As discussed in the previous chapter, a multitude of software libraries for parallel low level image processing has been described in the literature [75, 80, 93, 108, 109, 112, 118, 153, 159]. An important design goal in much of this research is to provide operations that have optimal efficiency on a range of parallel machines. In general, this is achieved by hard-coding a number of different parallel implementations for each operation, one for each platform. Unfortunately, the creation of a parallel library in this manner has several major drawbacks. First, manually creating multiple parallel versions of the many operations commonly applied in image processing research is a laborious task. Second, obeying to requests for library extensions becomes even more troublesome than in the sequential case. Third, as new target platforms are made available at regular intervals, code maintenance becomes hard — if not impossible — on the long term. Finally, with each library expansion it becomes ever more difficult to incorporate a single elegant optimization strategy that can guarantee intra-operation efficiency as well as inter-operation efficiency. For these reasons we take a different approach.

*This chapter combines our papers published in Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS 2001) [141] and Parallel Computing [149].
In the design of our parallel library we strive to minimize the implementation effort, without compromising on the efficiency of execution. The first step in achieving this goal is recognizing that there is a limited number of ways in which the pixels in an image can be processed to produce meaningful results. Important in this respect is the classification of low level image processing operations made in Image Algebra [131]. Originating from this classification, the sequential Horus image processing library [84] (which serves as a basis for the core component of our software architecture, see Section 2.3.2) provides a small set of so-called algorithmic patterns. As will be explained in this chapter, the primary importance of the algorithmic patterns is that each serves as a template operation for a large set of image processing operations with comparable behavior. Also, the algorithmic patterns abstract from the actual datatype each operation is applied upon, to avoid a combinatorial explosion of code that deals with all possible kinds of image datatypes.

The next important step in achieving our goal is recognizing that, for parallel implementation of each algorithmic pattern, much of the related sequential code can be reused. To that end, for each sequential algorithmic pattern present in the Horus library we have defined a so-called parallelizable pattern. Such pattern constitutes the maximum amount of code of an algorithmic pattern that can be performed both sequentially and in parallel — in the latter case without having to communicate to obtain data residing on other processing units.

The final step in reaching our goal is to implement all parallel operations such that they are capable of adapting to the performance characteristics of a parallel machine at hand. As machine-specific performance characteristics should not be incorporated explicitly in any library implementation, an additional automatic code optimization phase is required to be performed at compile time or even at run time.

Hence, apart from giving a detailed overview of the design philosophy of our software library, this chapter primarily focuses on the following research issue: How to implement a parallel image processing library such that code redundancy is avoided as much as possible, and efficiency of execution on all target platforms is guaranteed. We present a solution to the problem in the form of a generic description of parallelizable patterns. Based on the description, we show how parallel versions of many commonly used image processing operations are obtained by concatenating high-level communication routines, basic memory operations, and operations that constitute a specialization of a parallelizable pattern. We demonstrate that, apart from being relatively simple to implement, a parallel library built in this manner is extensible, easily maintainable, and still high in performance.

It is important to stress that this chapter does not touch upon the important topic of inter-operation optimization, that is, optimization across library calls. Parallel operations that are implemented on the basis of parallelizable patterns may still perform many unnecessary communication steps when applied as part of a complete image processing application. As a result, efficiency of execution may not be optimal. As indicated in the previous chapter, our complete software architecture deals with this problem by applying domain-specific performance models in combination with an additional, integrated scheduling tool. These issues are all outside the scope of this chapter, however, and are discussed in extensive detail in Chapters 4, 5, and 6.
3.1 Algorithmic Patterns: The Horus Approach

This chapter is organized as follows. Section 3.1 gives an overview of the design philosophy of the sequential Horus library. Section 3.2 describes the manner in which parallelism is integrated in Horus. Section 3.3 discusses the programming paradigm adopted in all parallel implementations. Section 3.4 gives a generic description of parallelizable patterns, including a default parallelization strategy for image operations. To illustrate the use of parallelizable patterns, the implementation of two example operations is discussed in detail. Finally, conclusions are presented in Section 3.5.

3.1 Algorithmic Patterns: The Horus Approach

Whereas implementation of a single sequential image processing routine is often easy, creating a software library that is to contain an extensive set of such operations is notoriously hard. This is because image library users need operations that can be applied to a large number of (combinations of) different data structures, whose individual data elements in turn can be of many different types. More specifically: although two-dimensional image structures are most commonly used, the bulk of all library functionality also should be applicable to three- (or higher-) dimensional images, image regions, and other types of dense datafields (e.g., histograms). In addition, the type of each individual element in a data structure can be scalar (e.g., int, float, Boolean), complex, compound (e.g., a vector representing RGB color), and so forth.

Providing support for a combinatorial explosion of code that deals with all these data structures and types is by no means an easy task. Consequently, many existing sequential image processing libraries usually restrict support to a small set of datastructures, datatypes, and even operations [9, 45]. It is clear that such limitations have a negative effect on a library's popularity and expected lifespan.

To deal with these problems, the design and implementation of the Horus image processing library [83, 84, 85] is based on a generic programming approach. The Free On-line Dictionary of Computing [71] defines this approach as follows:

Generic programming is a technique that aims to make programs more adaptable by making them more general. Generic programs often embody non-traditional kinds of polymorphism; ordinary programs are obtained from them by suitably instantiating their parameters. In contrast with normal programs, the parameters of a generic program are often quite rich in structure. For example, they may be other programs, types or type constructors or even programming paradigms.

To be more specific: given X datatypes, Y containers (data structures), and Z algorithms as essential software library components, abstraction by way of generic programming reduces the possible $X \times Y \times Z$ implementations to $X + Y + Z$ implementations. Consequently, generic programming greatly enhances library maintainability.

In Horus, generic data structures (i.e., container structures that are made independent of the type of the contained object) are implemented by way of the C++ template mechanism [158] — a programming concept that allows a type to be a parameter in the definition of a class or a function. Using the same mechanism, Horus also provides
generic algorithms that can work on the generic data structures. Because we feel that the importance of the Horus library lies more in its concepts than in its implementation, we refrain from presenting actual template code here. For more information and implementation details we refer to the Horus documentation [84, 85].

Apart from abstracting from the actual datatype each operation is applied upon, the amount of Horus library code is reduced even further by implementing only a small number of algorithmic patterns that covers the bulk of all commonly applied image processing operations. An algorithmic pattern corresponds to one of the operation classes defined in Image Algebra [131], each of which gives a generic description of a large set of operations with comparable behavior. As such, each image operation that maps onto the functionality as provided by an algorithmic pattern is implemented in Horus by instantiating the algorithmic pattern with the proper parameters, including the function to be applied to the individual data elements. As an example, an algorithmic pattern may produce a result image by applying a unary function to each pixel in a given input image. By instantiating the pattern 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.

The version of Horus that serves as the basis for all further discussions provides the following set of algorithmic patterns:

- **Unary pixel operation.** Operation in which a unary function is applied to each pixel in the image. Examples: negation, absolute value, square root.

- **Binary pixel operation.** Operation in which a binary function is applied to each pixel in the image. Examples: addition, multiplication, threshold.

- **Reduce operation.** Operation in which all pixels in the 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 the 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 the image’s domain is transformed. Examples: translation, rotation, scaling.

The presented set of algorithmic patterns is not complete, as it does not cover all functionality required in the early stages of algorithm or application development. The Horus library, however, is subject to continuing research and extensions. Among the most important current and expected future library additions are algorithmic patterns that can be used to instantiate (1) multi pixel operations, (2) iterative and recursive neighborhood operations, and (3) queue based algorithms. Also, apart from the algorithmic pattern for geometric operations, all of the patterns that are currently incorporated in Horus are restricted to instantiating translation invariant operations only. Translation variant versions of the presented algorithmic patterns will be incorporated in the future as well.
3.2 Integration of Parallelism in Horus

As discussed in Section 2.3.2, the parallel library that constitutes the core of our software architecture is an extended version of the sequential Horus library. In the parallel version all additional functionality is implemented such that it does not interfere with the existing sequential code. As such, the parallel library can still be instructed to resort to traditional sequential operation — which generally is preferred over single node parallel operation due to additional overhead costs.

From a design perspective the extended library consists of four logical components, as shown in Figure 3.1. The following discusses each in turn, and identifies the relationships among them:

Component C1: Sequential Algorithmic Patterns

The first component (C1) consists of the set of sequential algorithmic patterns introduced in Section 3.1. As indicated in Figure 3.1, each algorithmic pattern present in this component is implemented as a sequence of sequential routines. All operations in such sequence must be separately available in the library — but not necessarily as user-callable routines. Apart from memory operations that may be required for the creation or destruction of internal data structures, the most important operation

Figure 3.1: Relationships between library components C1-C4 (note: the actual code differs substantially). Sequential code blocks that constitute a specialization of a parallelizable pattern are used in the implementation of sequential algorithmic patterns as well as in the implementation of the related parallel counterparts. All functionality is provided to the user through a sequential application programming interface (API) that contains no references to the library’s parallel processing capabilities.
in such sequence is what we refer to as a parallelizable pattern. Here it is sufficient to indicate that a parallelizable pattern constitutes a code block that incorporates only those instructions in a sequential algorithmic pattern that can be applied in the implementation of the related parallel algorithmic pattern as well — in the latter case without having to communicate to obtain essential data residing at any other processing unit. For a much more formal description of parallelizable patterns we refer to Section 3.4.

Component C2: Parallel Extensions

Next to the sequential algorithmic patterns, an additional set of routines is implemented to introduce parallelism into the library (component C2 in Figure 3.1). The parallel extensions deal with all aspects of parallelization, ranging from the logical partitioning of data structures to the actual exchange of data among processing units. To have full control over all interprocess communication\(^1\), all extensions are implemented using MPI [104]. The implemented set of parallel extensions is divided into three classes:

1. Routines for data structure partitioning. These routines are used to specify the data structure responsibilities for each processing unit, i.e. to indicate which data parts should be processed by each node. In practice, a data structure is mapped onto a logical grid of processing units of up to 3 dimensions, which allows for optimal domain decomposition of the bulk of all image data structures (see also Chapter 5). The mapping is performed in such a way that the number of data elements each node is responsible for is well-balanced.

   The most important routines in this class are the 'doPartition()' and 'rePartition()' operations, which define the (new) responsibilities for a given data structure. Responsibilities are based on the logical grid of processing units, and the dimensionality and size of a data structure. All other routines in this class are requests for partitioning information (for example, to obtain the size and dimensionality of partial data structures other processing units are made responsible for).

2. Routines for data distribution and redistribution. These operations are used for the actual spreading (either scattering or broadcasting), gathering, and redistribution of data structures. Although the MPI 1.1 standard provides most of this functionality and the new MPI 2.0 standard defines all, we have made multiple implementations ourselves using the standard blocking MPI send and receive operations. We refer to Chapter 5 for a detailed discussion on the rationale, and the implications for application performance and optimization.

   It should be noted that data distribution could have been regarded independent from data partitioning. To avoid any unnecessary communication, however, we have made the distribution of data structures dependent on the assigned data

\(^1\)Note that in our implementations (and also in the remainder of this thesis) we assume a one-to-one relationship between processes and processing units.
3.3 Data Parallel Image Processing

responsibilities. Data partitioning is therefore always applied as part of a data distribution operation.

3. Routines for *overlap communication*. These operations are used to exchange *shadow regions* (e.g., image borders in neighborhood operations) among neighboring nodes in a logical grid of processing units.

All operations in component C2 are kernel routines, and are not made available to the library user.

**Component C3: Parallel Algorithmic Patterns**

To reduce code redundancy and enhance library maintainability as much as possible, much of the source code for the sequential algorithmic patterns is reused in the implementation of their respective parallel counterparts. More specifically, the implementation of each parallel algorithmic pattern is obtained by inserting communication operations from component C2 in the sequence of routines that constitutes the implementation of the related sequential algorithmic pattern. The communication routines are to obtain all non-local data (i.e., data residing on other processing units) required during execution of the parallelizable pattern. The communication routines also gather partial results data from all processing units to a single (root) node as soon as the execution of the parallelizable pattern has finished. As such, during execution all instantiations of the parallel algorithmic patterns run in a Bulk Synchronous Parallel manner [103, 162].

**Component C4: Fully Sequential API**

The extended image processing library is provided with an application programming interface (component C4 in Figure 3.1) identical to that of the original sequential Horus library. Due to the fact that the API contains no references to the library's parallel processing capabilities, no additional effort is required from the application programmer to obtain a parallel program. In other words: any application implemented for a sequential machine — after recompilation — can be executed on a cluster as well. As such, the library fully adheres to the first requirement of user transparency as defined in Section 2.2.

### 3.3 Data Parallel Image Processing

The previous sections implicitly indicated that we have adopted *data parallelism* as the programming model for implementing all parallel algorithmic patterns. In the following we clarify why we have adopted this approach as the sole technique for parallelization, rather than any other approach or even a combination of approaches. Also, to lay the foundations for the generic description of parallelizable patterns presented in Section 3.4, we give a formal description of the manner in which image data structures are represented in our data parallel library.
3.3.1 Data Parallelism versus Task Parallelism

Although many more programming paradigms for parallel computing exist, the models of data parallelism and task parallelism are used most frequently because of their effectiveness and general applicability. As defined in [50], the data parallel model focuses on the exploitation of concurrency that derives from the application of the same operation to multiple elements of a data structure. In other words, it is a programming model in which a single routine is applied to all elements of a data structure simultaneously. In contrast, the task parallel paradigm constitutes a model of parallel computing in which many different operations may be executed concurrently [170].

In the literature, a multitude of papers exists in which each of these paradigms is used effectively for parallelizing (low level) image processing operations (e.g., see [19, 32, 134, 161]). Also, for certain image processing problems it has been shown that application of a combination of the two paradigms in a single program is more effective than using either paradigm exclusively (e.g., see [112, 126]).

Despite the potential benefits of applying task parallelism, we have decided to restrict all parallel implementations in our library to the data parallel model. The reasons for this decision are as follows. First, the application of data parallelism is a natural approach for low level image processing, as many operations require the same function to be applied to each individual data element (or small set of elements around each data element) present in an image data structure. Second, as our parallel library is to serve as an aid in image processing research, the number of independent tasks available in most applications is expected to be small. This is because in the design phase of algorithms or applications, testing and evaluation generally is performed using relatively small problem sizes (e.g., using a single image rather than a database of thousands of images). A third reason is related to the scalability in the number of processing units. As the number of independent tasks in most image processing applications generally is much smaller than the number of elements present in the input (image) data structures, the number of processors that can be applied effectively is generally much larger in the data parallel case. Another important reason is that load balancing (i.e., evenly distributing all work among the available processing units) is generally much more difficult in the task parallel model. Especially in case independent parallel tasks represent highly varying workloads, it is difficult to ensure that each processor has exactly the same amount of work to do.

The decisive factor for not incorporating task parallelism in our software architecture, however, is the difficulty of combining this programming paradigm with the requirements of user transparency. The presence of a fully sequential API implies that we would have to incorporate a separate interpretation and optimization strategy to find all independent tasks available in an application. Effectively, this implies that we would have to develop, at least in part, a parallelizing compiler. For reasons explained in Section 2.2.1 we expect such compiler not to yield a desirable solution.

It would have been possible to incorporate the notion of task parallelism in the library’s API, e.g. by providing aggregated operations that can work on sets of images. However, this approach would dramatically reduce the library’s chances of widespread acceptance, as it would require most existing applications to be rewritten by hand.
3.3. Data Parallel Image Processing

(e.g., by replacing loop constructs by a single call to an aggregated library operation). This is not trivial, as it requires the user to personally identify dependencies among tasks (which is often difficult due to the presence of indirections in the C or C++ code). To shield the user from having to deal with any of these issues, and also to avoid having to implement any optimization strategy that can detect independent tasks automatically, we have refrained from incorporating task parallelism altogether. As will be shown in Chapter 7, despite the fact that all implementations are restricted to the data parallel approach, obtained performance improvements are generally well within the efficiency requirements as put forward in Chapter 2.

3.3.2 Representation of Digital Images

An image data structure in our library 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, 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

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

where \( o = (o_1, o_2, \ldots, o_n) \) represents the origin of the image, and \( k_i \) represents the extent of the domain in the \( i \)-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

![Figure 3.2: 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.](image-url)
Scalar value is represented by one of the common datatypes, such as byte, int, or float. The set of all images having range $\mathbb{F}$ and domain $\mathbb{X}$ is denoted by $\mathbb{F}^\mathbb{X}$. In summary, $a \in \mathbb{F}^\mathbb{X}$ (i.e., $a : \mathbb{X} \rightarrow \mathbb{F}$) is a shorthand notation for

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

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 resulting from a scatter or broadcast operation is called a partial image. For each partial image additional partitioning and distribution information is available. The 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 $a_{p_i}$; its domain is denoted by $X_{p_i}$. As data spreading can not result in a loss of data, for each image $a \in \mathbb{F}^\mathbb{X}$ distributed over $n$ processing units:

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

The $n$ partial images related to $a$ together form one logical structure, referred to as a distributed image. A distributed image is denoted by $a_d$, 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 $X_d$ 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 3.2).

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.

### 3.4 Parallelizable Patterns

As stated in Section 3.2, we try to enhance library maintainability by reusing as much sequential code as possible in the implementations of the parallel algorithmic patterns. To that end, for each sequential algorithmic pattern 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 generic description of parallelizable patterns, and show their application in parallel implementations.
3.4.1 Generic Description

A parallelizable pattern is a sequential generic operation that takes zero or more source structures as input, and produces one destination structure as output. A 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 a 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 in each task is limited to the single data point to be updated.

All $n$ tasks are tied to a different task location $x_i$, with $i \in \{1, 2, \ldots, n\}$. The set $L$ 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, $L$ 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 $x_i$ 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 $i \in \{1, 2, \ldots, n\}$) no data point is accessed other than $x_i$.

- **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 $x_i$.

- **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 $x_i$. As an example, the $5 \times 3$ neighborhood of a point $x = (x_1, x_2) \in X$ is given by

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

  where $X \subset Y$.

- **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 3.3 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 differ in the type of combination operation that is permitted. In a parallelizable
Chapter 3. Parallelizable Patterns in Low Level Image Processing Algorithms

Figure 3.3: 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.

pattern of type 1 no restrictions are imposed on 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 in the destination structure 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 $f()$. For 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 routines, e.g. incorporating all algorithmic patterns of Section 3.1. As such, the presented generalization captures a large majority of all operations commonly applied in image processing research (i.e., it comprises an estimated coverage of over 90%). It should be noted, however, that the two types do not present a complete coverage of the typical implementations of all operations in this particular field of research. For example, algorithms in which write access is to multiple data structures is required 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. In how far these limitations pose any unreasonable restrictions on future library adaptations (and thus necessitates extension of the generic description of parallelizable patterns) is as of yet unknown (see also Section 3.4.5).

All algorithmic patterns that do fit into the given generalization are applicable in the process of 'parallelization by concatenation of library operations', described in Section 3.2. As discussed in the remainder of this section, on the basis of the generic description we define a standard parallelization strategy that always results in a correct data parallel implementation for any algorithmic pattern that maps onto at least one of the two parallelizable pattern types.
3.4.2 Default Parallelization Strategy

The number of elements in the set of task locations $L$ determines the number of steps executed by a parallelizable pattern. Hence, by providing each node in a parallel system with a set $X \in L$, the work is distributed (i.e., in a data parallel manner). In addition, the access pattern type associated with each structure involved in the operation prescribes how non-local data accesses are avoided with minimal communication overhead. As such, an optimal\(^\dagger\) default parallelization strategy is obtained for any operation that maps onto one of the presented parallelizable pattern types.

First, before executing a type 1 parallelizable pattern each processing unit is 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 parallelizable pattern, each processing unit creates a fully overlapping destination structure locally. This is always possible, as the value of all data points are given a 'neutral value', as defined by the operation.

Next, source data structures are obtained by executing (1) a non-overlapping scatter operation for each structure having a one-to-one access pattern, (2) a partially overlapping scatter operation 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 operation for all other structures. In case the values of a source structure can be calculated locally, and if it is less time-consuming to do so, no communication routines are performed at all.

Finally, when a type 1 pattern has finished, the complete destination structure is obtained by executing a gather operation. For a type 2 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, assuring the correctness of the final reduction. In both cases, the result structure is returned either to one node, or to all.

On the basis the generic description of parallelizable patterns, the following shortly discusses parallel implementation of two example algorithmic patterns, i.e. global reduction and generalized convolution.

3.4.3 Example 1: Parallel Reduction

A sequential generic reduction operation performed on input image $a$, producing a single scalar or vector value $k$, is defined as follows:

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

$$k = \Gamma a = \Gamma_x a(x) = \Gamma_i^n a(x_i) = a(x_1) \gamma \ a(x_2) \gamma \cdots \gamma \ a(x_n),$$

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

\(^\dagger\)The default strategy is optimal for operations executing in isolation only. In case multiple operations are executed in sequence, additional inter-operation optimization is required (see Chapter 6).
As shown in Figure 3.4, 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 \( a \) 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 \( a \) is read and combined with the current value of \( k \).

The first implementation is a specialization of the parallelizable pattern of type 1 as described in Section 3.4; the second implementation is a specialization of the type 2 parallelizable pattern. The first implementation is not useful for our purposes, however, as its execution is limited to a single processing unit. This is because the set of task locations \( L \) 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 \( L \) contains all locations in input image \( a \). 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 3.4.2. A pictorial view of the operation executed in parallel is given in Figure 3.5.

![Figure 3.4: Sequential reduction - two possible implementations.](image)

![Figure 3.5: Example reduce-to-all operation executed on 2 processing units.](image)
3.4.4 Example 2: Parallel Generalized Convolution

A generalized convolution performed on input image \( a \), producing output image \( c \), given a kernel \( t \), is defined as follows:

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

\[
c = a \ast t = \{(x, c(x)) : c(x) = \Gamma_y a(x + y) \circ t(y) \},
\]

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 \( t \) is given by \( 2k_i + 1 \). Several common generalized convolution instantiations are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Kernel Operation</th>
<th>( \circ )</th>
<th>( \gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>multiplication</td>
<td>addition</td>
</tr>
<tr>
<td>Dilation</td>
<td>addition</td>
<td>maximum</td>
</tr>
<tr>
<td>Erosion</td>
<td>addition</td>
<td>minimum</td>
</tr>
</tbody>
</table>

Table 3.1: Example generalized convolution instantiations.

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 3.6. Again, set \( L \) is implicit, and contains all pixel positions in either the input image or the output image.

When comparing Figure 3.3(a) to Figure 3.6(a) it may seem that the operation directly constitutes a parallelizable pattern. Figure 3.6(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

![Figure 3.6](image-url): Sequential generalized convolution. Does not represent a parallelizable pattern, as read accesses outside the domain of the input image are possible (see (b)).
handling from the actual convolution operation. This makes implementations more robust and generally also faster, due to the fact that irregular memory accesses are avoided. For parallel implementation this strategy has the additional advantage that the algorithmic pattern 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 3.7 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 3.7 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 3.7 gives a simplified view, as some steps of the operation are not shown. For example, depending on the type of operation, the kernel structure is either broadcast or calculated locally.

3.4.5 Discussion

The generic description of parallelizable patterns is important as it states the requirements for sequential implementations that are to be reused in related parallel counterparts. In addition, for each specialized parallelizable pattern implemented on the basis of the generic 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 generic 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.
3.5 Conclusions and Future Work

In this chapter we have indicated how an extensive parallel image processing library is constructed with minimal implementation effort, whilst ensuring efficiency of (sequential and parallel) execution at the same time. We have focussed on the notion of parallelizable patterns, and discussed how parallel implementations are easily obtained by sequential concatenation of operations that are separately available in the library. More specifically, on the basis of a set of four data access pattern types, we have obtained a default parallelization strategy for any operation that maps onto one of two parallelizable pattern types. For each image processing operation executed in isolation, this default parallelization strategy is optimal. This is because communication overhead is minimized, while — for the given parallelization granularity — the available parallelism is fully exploited. As such, we have shown how a parallel image processing library can be made extensible, and easily maintainable.

It is important to note, however, that in this chapter we have not discussed the important issue of inter-operation optimization, or: optimization across library calls. To obtain high performance for sequences of library routines, or for complete applications, it is not sufficient to consider parallelization and optimization of each library operation in isolation. This is because code consisting of a given sequence of parallel routines, where each routine is parallelized as described in this chapter, often contains many redundant communication steps. Also, it is often possible to further reduce communication overhead by combining multiple messages in a single transfer. Our solution to this fundamental problem, and the integration of this solution in our software architecture, is discussed extensively in Chapters 4, 5, and 6 of this thesis.

In the future the generic description of parallelizable patterns may need to be extended or adapted to capture image library additions and extensions. For example, at the time of writing it is not entirely clear whether the optimal parallel implementation of recursive filter operations as described in [28, 40] can be derived from one of the presented parallelizable pattern types. As a consequence, we may need to investigate for what type of image operations the strategy of 'parallelization by concatenation of library routines' breaks down, i.e., does not provide efficient implementations, or can not be applied at all. Still, the presented description of parallelizable patterns is important, as it prescribes the sequential implementation of a large majority (i.e., over 90%) of all operations commonly applied in image processing research. Any implementation obtained in this manner can be applied without change in efficient parallel implementations as well — thus avoiding unnecessary code redundancy, and minimizing the required implementation effort.