User Transparent Parallel Image Processing
Seinstra, F.J.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 2

A Sequential Programming Model for Efficient Parallel Image Processing*

"O Freunde, nicht diese Töne!
Sondern laßt uns angenehmere anstimmen..."

Ludwig van Beethoven - Symphony No. 9 "Choral" (1824)

Parallel and distributed computing architectures, whose performance far exceeds that of traditional sequential systems, have been available for decades. As an example, the development of the Illiac IV [12], a machine commonly seen as the first true parallel system, started as early as 1965. In recent years, high performance computing systems have become more and more widespread, especially with the advent of highly flexible Field-Programmable Gate Arrays (FPGAs [18, 24, 66]) and relatively cheap Beowulf clusters [7, 157]. Also, specialized digital signal processing (DSP) devices and dedicated hardware architectures have become widely available [48, 91, 127].

As discussed in Chapter 1, the processing power as provided by parallel and distributed systems is essential for many image processing applications. Also, it has been recognized for years that the application of parallelism in imaging can be highly beneficial [161]. As a result, collaboration between the research communities of high performance computing and imaging has been commonplace, and typically resulted in specialized hardware configurations (e.g., see [47, 65, 88, 89, 107]) capable of efficiently executing domain-specific routines [19, 32, 134]. Yet, in spite of the importance of these achievements, the application of parallelism in imaging research is not widespread.

*This chapter contains portions of our paper as appeared in Proceedings of the 15th International Conference on Pattern Recognition (ICPR 2000) [140]. An extended version of this chapter is to appear in Concurrency and Computation: Practice and Experience [146].
Primarily, we ascribe the rather small user base of parallel computing within the image processing research community to the high threshold associated with the use of high performance computing architectures. One determinative factor for the existence of the threshold is the relatively high cost involved in using a specific parallel machine. In general, the image processing community can not afford to acquire and maintain such systems, and has to rely on hardware and support provided by the computing community. More importantly, the threshold exists due to a common characteristic of parallel and distributed systems, namely: they are much harder to program than sequential computers. Although several attempts have been made to alleviate the problem of software design for parallel and distributed systems, as of yet no solution is available that has found widespread acceptance.

As will be discussed extensively in this chapter, the latter problem is due to the fact that no efficient parallelization tool exists that is provided with a programming model that matches the entrance level of the average image processing practitioner. Most existing software development tools require the user to explicitly identify the available parallelism, often at a level of detail beyond the expertise and interest of most image processing researchers. Hence, it is essential to provide an alternative tool that offers a more 'familiar' programming model.

In this chapter we argue that a parallelization tool for the image processing research community is acceptable only if it hides all parallelism from the application programmer, and produces highly efficient code in most situations. Stated differently, we argue that a programming model is considered 'familiar' only, if it offers complete user-transparent parallel image processing.

Several solutions have been described in the literature that allow for user transparent implementation of high performance image processing applications. In all cases, solutions are being provided in the form of an extensive software library containing parallel versions of fundamental image processing operations. The solutions, however, all suffer from one of several obstacles for widespread acceptance. Most significantly, the efficiency of parallel execution of complete applications often is far from optimal. In addition, the provided software library often does not incorporate a sufficiently high level of sustainability, thus dramatically reducing the chance on long term success.

Given these observations, the primary research issue addressed in this chapter is: How to provide the average image processing practitioner with a fully sequential programming model that allows for implementation of efficient parallel imaging applications such that the user is shielded from all issues related to parallelization and performance optimization? The second research issue addressed here is the following: How to incorporate such sequential programming model in an efficient parallelization tool that allows its developers to respond to changing demands quickly and elegantly?

In this chapter we propose a complete software architecture for user transparent parallel image processing that is specifically designed to deal with these issues. We discuss the requirements put forward for such programming tool, and provide a general overview of the architecture's constituent components. Essentially, this overview serves as a roadmap for the remaining chapters of the thesis.

This chapter is organized as follows. Section 2.1 presents a list of requirements a potential target hardware architecture should adhere to for it to be used in image
processing research. Based on the requirements one particular class of platforms is indicated as being most appropriate. In Section 2.2 the notion of user transparency is introduced, and used as a basis for an investigation of available tools for implementing parallel imaging applications on the selected set of target platforms. As the investigation shows that no existing development tool is truly satisfactory, our new software architecture for user transparent parallel image processing is introduced in Section 2.3. Concluding remarks are given in Section 2.4.

2.1 High Performance Computing Architectures

A parallelization tool intended as a programming aid in imaging research is more likely to find widespread acceptance if it is targeted towards a machine that is favored by the image processing research community. Implicitly (i.e., by ignoring inappropriate architectures), the imaging community has defined several requirements for such machines. These are formulated as follows:

- **Wide availability.** To ensure that the imaging community at large can benefit from a parallelization tool, it is essential that the target platform is widely available. Less popular or experimental architectures tend to suffer from a lack of continuity, thus hindering the ever present desire for hardware upgrades.

- **Ease of accessibility.** The target platform should be easily accessible to the image processing practitioner. This refers to the manner in which one logs on to the machine, how programs are to be compiled and run, and to the ease by which a set of processing elements is obtained. The last issue is particularly important where multiple users share a pool of processing elements.

- **Unrestricted programmability.** The hardware platform should not restrict the application programmer. It should be capable of executing the various operations commonly used as a basis in image and video processing applications.

- **Ready upgradability.** It is essential that the software developed for the target platform should be executable after each upgrade to the next generation of the same architecture. In other words, the desired continuity of the target platform requires a high degree of backward compatibility.

- **High efficiency.** The target platform should be capable of obtaining significant performance gains, especially for the most common imaging operations. If no significant improvements are to be expected, the process of accessing a parallel machine, and implementing and optimizing code for it, would be useless.

- **Low cost.** Even when significant speedups are to be expected, the financial burden of executing imaging software on the target platform should be kept to a minimum. As high performance computing is not a goal in itself in imaging research, the amount of money that may be spent on computing resources is small compared to the amount of money that flows to more fundamental research.
We are aware of the fact that additional requirements may hold in other application areas. Here, we deem such requirements to be either inherent to parallel systems in general (such as the desire for hardware scalability), or unimportant to most image processing practitioners (such as the amount of control over the structure, processing elements, operation, and evolution of a particular parallel system). Also, for specific image processing research directions additional requirements may be of significant importance. For example, in certain application areas strict limitations may be imposed on the target platform's size, or the amount of power consumption. In this thesis, however, we restrict ourselves to the list as presented here, as this represents the set of general requirements that holds for most image processing research areas.

**Favoring Beowulf-type Commodity Clusters**

As described in [33, 78, 146], several machines in the classes of general purpose MISD-, SIMD-, and MIMD-style parallel architectures (Flynn [49]) are potential candidates for high speed execution of image processing applications. Also, many special purpose architectures (e.g., ASICs [1, 98], FPGAs [24, 46], DSPs [47, 48]), as well as several enhanced general purpose CPUs [42, 116, 121], have been designed to obtain even higher performance for specific image processing tasks [33].

Irrespective of the significance of these systems, one architecture type stands out as particularly interesting for our purposes, i.e., the class of Beowulf-type commodity clusters [7, 157]. As one of the original designers of this type of architectures, Thomas Sterling, describes in a guest editorial on the clusters@top500 website [30]. Beowulf-type systems are particularly important because “it is quite possible that by the middle of this decade clusters in their myriad of forms will be the dominant high-end computing architecture.” Indeed, a strong trend in high performance computing is the growing use of commodity clusters, and many such systems are currently installed at research institutes and in commercial environments around the world.

Apart from being widely available, clusters often are made easily accessible to researchers from outside the computing community. Expected cooperation between multiple research disciplines often is the determinative factor in obtaining funding for such computer systems in the first place. In addition, the general-purpose nature of the constituent computing nodes fully adheres to the requirement of ‘unrestricted programmability’. In fact, the bulk of all image processing research is currently being performed on similar computing nodes traditionally employed in a stand-alone manner. Also, a major advantage of the use of personal computers as constituent components is a long term continuity combined with ‘ready upgradability’.

The single characteristic that makes a cluster favorable over other systems, however, is the emphasis on price-performance. As Sterling states in the same editorial: “for many application types, commodity clusters will deliver better, by even orders of magnitude in many cases, price-performance with respect to alternative systems”. From these properties, in combination with the fact that many references exist that show significant performance gains for a multitude of different image processing applications (e.g., see [75, 79, 93, 159]), we conclude that clusters constitute the most appropriate target platforms for our specific needs.
2.2 Software Development Tools

Apart from its design and capabilities, the (commercial) success of any computer architecture significantly depends on the availability of tools simplifying software development. As an example, for many users it is often desirable to be able to develop programs in a high-level language such as C or C++. Unfortunately, and in contrast with general-purpose sequential systems, for many of the hardware architectures referred to in Section 2.1 available high-level language compilers often have great difficulties in generating assembly code that makes use of the machine’s parallel capabilities effectively. As a result, for highest performance the programmer often must optimize the critical sections of a program by hand.

Whereas assembly coding or hand-optimization may be reasonable for a small group of experts, most users prefer to dedicate their time to describing what a computer should do rather than how it should do it. Consequently, many programming tools have been developed to alleviate the problem of low level software design for parallel and distributed systems. In all cases such tools are provided with a programming model that abstracts from the idiosyncrasies of the underlying parallel hardware. The small user base of parallel computing in the imaging community indicates, however, that no existing parallelization tool incorporates a level of abstraction that truly matches the image processing researcher’s frame of reference.

The ideal solution would be to have a parallelization tool that abstracts from the underlying hardware completely, allowing users to develop optimally efficient parallel programs in a manner that requires no additional effort in comparison to writing purely sequential software. Unfortunately, no such parallelization tool currently exists and due to the many intrinsic difficulties it is commonly believed that no such tool will be developed ever at all [17]. However, if the ideal of 'obtaining optimal efficiency without effort' is relaxed somewhat, it may still be possible to develop a parallelization tool that constitutes an acceptable solution for the image processing research community. The success of such a tool largely depends on the amount of effort requested from the application programmer and the level of efficiency obtained in return.

The graph of Figure 2.1 depicts a general classification of parallelization tools based on the two dimensions of effort and efficiency. Here, the efficiency of a parallelization tool is loosely defined as the average ratio between the performance of any image processing application implemented using that particular tool and the performance of an optimal hand-coded version of the same application. Similarly, the required effort refers to (1) the amount of initial learning needed to start using a given parallelization tool, (2) the additional expense that goes into obtaining a parallel program that is correct, and (3) the amount of work required for obtaining a parallel program that is particularly efficient. In the graph, the maximum amount of effort the average image processing practitioner generally is willing to invest into the implementation of efficient parallel applications is represented by THRESHOLD 1. The minimum level of efficiency a user generally expects as a return on investment is depicted by THRESHOLD 2. To indicate that the two thresholds are not defined strictly, and may differ between groups of researchers, both are represented by somewhat fuzzy bars in the graph of Figure 2.1.
In this thesis, each tool that is considered both 'user friendly' and 'sufficiently efficient' is referred to as a tool that offers full user transparent parallel image processing. Apart from adhering to certain levels of requested effort and obtained efficiency, an important additional feature of any user transparent tool is that it does not require the user to fine-tune any application in order to obtain particularly efficient parallel code (although the tool may still allow the user to do so). Based on the above considerations, we conclude that a parallelization tool constitutes an acceptable solution for the image processing community only, if it can be considered fully user transparent.

One may argue that the thresholds in Figure 2.1 are not straight lines in each of the two dimensions, but are better combined in a single diagonal (or curved) line. This would be reasonable, as for a small amount of obtained efficiency the user is probably not prepared to invest as much effort as for a much higher level of efficiency. The presented classification is still valid, however, as we argue that it should not be required from the user to invest any additional effort to obtain higher efficiency.

### 2.2.1 General Purpose Parallelization Tools

The following gives an overview of the most significant development tools that (a.o.) can be used for implementing image processing applications on clusters. For each tool we discuss the level of abstraction incorporated in the programming model, and assess to what extent it adheres to the properties of full user transparency. The discussion starts with an overview of general-purpose parallelization tools, and is followed by an overview of tools designed specifically for developing high performance image processing applications.
Message Passing Libraries

Good examples of tools from the set of efficient programming aids for experts in parallel computing are the many software libraries providing message passing functionality [6]. Message passing is a programming paradigm based on the concept of processes that explicitly communicate data. It is mainly intended for programming distributed memory MIMD-style multicomputers, but the paradigm applies to shared memory machines as well. Many efficient and portable message passing systems have been described in the literature [102], but the sets of library routines provided by PVM (Parallel Virtual Machine [53]) and MPI (Message Passing Interface [61, 104, 105]) have become the most widely used [54, 67].

Parallel programming on the basis of message passing requires the programmer to personally manage the distribution and exchange of data, and to explicitly specify the parallel execution of code on different processors. Although this approach often produces highly efficient parallel programs, even for expert programmers it is difficult to do correctly [29]. This is due to the fact that message passing tools do not provide explicit support for the design and implementation of parallel data structures. Also, deadlocks are introduced easily, and debugging is hard under critical dependencies in the relative timing of events. Due to these problems, message passing is often referred to as the "assembly language of parallel computing", since it offers "a means for expressing parallel computation in an often painstaking, low-level, error-prone manner" [23]. Given these observations, we conclude that message passing is not the programming paradigm of choice for the average image processing researcher.

Shared Memory Specifications

As message passing was intended for client/server applications running across a network, PVM and MPI include costly semantics (e.g., the assumption of wholly separate memories) that are often not required on parallel systems with a globally addressable memory. To provide a simpler, yet efficient, and portable approach to implementing parallel programs, several shared memory specifications have been proposed, such as CRL [76] and Midway [15]. OpenMP [25, 119], which consists of a set of compiler directives, library routines, and environment variables to specify shared memory parallelism in Fortran and C/C++ programs, is the most commonly used.

Although a cluster does not fit in the class of shared-memory architectures, it is still relevant to include shared memory specifications in this evaluation. This is because shared memory specifications can be implemented on top of MPI, albeit at the cost of higher latencies [41]. Also, the provided programming paradigm is generally believed to be much simpler than MPI [57, 113].

One of the major advantages of shared memory specifications is that it is easy to incrementally parallelize sequential code. For non-expert programmers, however, it is still difficult to write efficient and scalable programs. In addition, the presence of both shared and private variables often causes confusion. As a result, the amount of effort requested from the average user still exceeds Threshold 1 in Figure 2.1. Therefore we conclude that shared memory specifications fall in the set of 'efficient expert tools' as well, and do not adhere to the requirements of full user transparency.
**Extended High-Level Languages**

An alternative to the library approach as followed by MPI and OpenMP is to provide a small set of modifications and/or extensions to an existing high-level programming language. Probably the most popular example of a language that has adopted this approach is HPF (High Performance Fortran [97]). A similar approach is followed in SPAR [128, 129], which is one of the many extended, parallel versions of Java. Also, many alternative extensions and modifications to C++ exist [171], of which Compositional C++ [26] and Mentat [60] are the most significant examples.

Irrespective of language design and compilation issues, for users of such languages the most important problem is that it is often required to understand in what situations the compiler can produce efficient executable code. For example, HPF requires that the distribution of data is specified separately from the routines operating on that data. Consequently, a mismatch between data distribution and functionality is easily introduced, possibly resulting in reduced performance due to huge amounts of unnecessary communication. As state-of-the-art compilers are not capable of detecting all such non-optimal behavior automatically [8, 17], much of the efficiency of parallel execution is still in the hands of the application programmer. As a result, the amount of effort a non-expert user must invest into writing efficient parallel codes in an extended high-level language also exceeds **THRESHOLD 1** in Figure 2.1.

**Parallel Languages**

Rather than extending an existing sequential language, it is also possible to design an entirely new parallel programming language from scratch. Considering parallelism directly in the design phase of a concurrent language offers a better chance of obtaining a clean and unified parallel programming model. Also, this approach facilitates implementation of efficient compiler optimizations, and the development of effective debugging tools. For these reasons, many parallel languages have been described in the literature (e.g., Ada [13], Occam [77], Orca [8, 137, 138], and Parlog [58]).

Despite years of intensive research, no parallel language has truly found widespread acceptance, either in the imaging community or elsewhere. One reason is that it appears to be difficult to design language features that are both generally applicable and easy to use [120]. A more important reason is that most scientific programmers are reluctant to learn an entirely new program development philosophy, or unfamiliar language constructs. As the parallelism in a parallel language is always explicit, and fine-tuning is often an inherent part of the program development process, we conclude that the amount of effort required from the average user generally is too high.

**Fully Automatic Parallelizing Compilers and Parallelizing Pre-Compilers**

As opposed to the parallelization tools discussed so far, an efficient **automatic parallelizing compiler** would constitute an ideal solution. It would allow programmers to develop parallel software by using a sequential high-level language without having to learn additional parallel constructs or compiler directives [10]. However, a fundamental problem is that many user-defined algorithms contain data dependencies that
prevent efficient parallelization. This problem is particularly severe for languages supporting pointers [2]. In addition, techniques for automatic dependency analysis and algorithm transformation are still in their infancy. Although interesting solutions have been reported that require the user to be conservative in application development (e.g., to allow efficient parallelization of loop constructs [52]), fully automatic parallelizing compilers that can produce efficient parallel code for any type of application do not exist — and a real breakthrough is not expected in the near future [17].

As an alternative, effort is currently being put into semi-automatic tools (such as FORGE [5]) that require the programmer to help the compiler interactively in the parallelization process. Although, in principle, this approach could allow user transparent implementation of parallel imaging applications, it can not be considered an acceptable solution. This is because the approach does not eliminate the burden of specifying the available parallelism; it merely pushes the problem forward to a later stage in the program development process.

### 2.2.2 Tools for Parallel Image Processing

The regular evaluation patterns in many low level image processing operations often make it easy to determine how to parallelize such routines efficiently. Also, because many different image operations incorporate similar data access patterns, a small number of alternative parallelization strategies often need to be considered. These observations have led to the creation of software development tools that are specifically tailored to image processing applications. Such tools may provide higher abstraction levels to the user than general-purpose tools, and are potentially much more efficient as important domain-specific assumptions often can be incorporated.

#### Programming Languages for Parallel Image Processing

One approach to integrating domain-specific knowledge is to design a programming language for parallel image processing specifically. Apply [64, 164] was one of the first attempts in this direction. It is a simple, architecture-independent language restricted to *local* image operations, such as edge detection, smoothing, and point operations. It is based on the observation that many operations follow a stereotypical form:

```plaintext
for each row
  for each column
    produce an output pixel based on a window of pixels around
    the current row and column in the input image
```

Apply exploits this idea by requiring the programmer to write only the innermost 'per pixel' portion of the computation. The iteration is then implicit and can easily be made parallel. Apply's restricted programming model allows easy implementation of quite an extensive set of operations. The programmer simply has to describe the program in terms of the smallest meaningful unit — namely, a window taken around a pixel in an image. Because a program is specified in this way, the compiler needs
only to divide the images among processors and then iterate the Apply program over the image sections allocated to each processor. Despite the fact that the language was capable of providing significant speedups for many applications, the programming model proved to be too restricted for practical use.

In a different language, called Adapt [165], the basic principles of Apply are extended to incorporate _global_ operations as well. In such operations an output pixel can depend on many or all pixels in the input image. Adapt is based on the split-and-merge programming model, in which data structures are split according to data position, and separately computed adjacent results are then merged. The programmer has to describe both the operation to be performed at every pixel of the image (as in Apply), as well as a combining operation to merge two results produced independently at different processors. Although the language certainly allows for an efficient parallel implementation of many important image processing applications, the programming model is not ideal. This is because the programmer is personally responsible for data partitioning and merging, albeit at quite a high level. For this reason we categorize Apply as an 'efficient expert parallelization tool' as well. Yet, it may constitute an acceptable solution for quite a large group of users.

An alternative approach is taken in a language called IAL (Image Algebra Language [35, 37]). IAL is based on the abstractions of Image Algebra [131], a mathematical notation for specifying image processing algorithms. IAL provides operations at the complete image level, with no access to individual pixels. For example, the Sobel edge detector is implemented in IAL as a single statement:

\[
\text{OutputIm} := (\text{abs}(\text{InputIm} \odot S_h) + \text{abs}(\text{InputIm} \odot S_v)) \geq \text{threshold};
\]

where \(S_h\) and \(S_v\) are the horizontal and vertical Sobel masks, and \(\odot\) represents convolution. The language proved to be useful for a wide range of tasks, but was limited in its expressive power. Two extended versions of IAL, I-BOL [20] and Tulip [155] provide a more flexible and more powerful notation. The languages permit access to data at either the pixel level or at the neighborhood level, without being architecture-specific. Although the languages hide all parallelism from the user, a major disadvantage is that it proved to be difficult to incorporate a global application optimization scheme to ensure efficiency of complete programs at all times. Another disadvantage is that the syntax of the languages differs quite somewhat from C and C++ — arguably the most popular languages applied in the image processing community.

**Parallel Image Processing Libraries**

An alternative to the language approach is to provide an extensive set of parallel image processing operations in library form — possibly as part of a complete framework that deals with additional issues, such as global application optimization. In principle, this approach allows the programmer to write applications in a familiar sequential language, and make use of the abstractions as provided by the library. Due to the relative ease of implementation, many parallel image processing libraries have been described in the literature, and here we will shortly discuss the most important ones.
One particularly interesting data parallel library implementation is described by Taniguchi et al. [159]. This software platform is applicable to both SIMD- and MIMD-style architectures, and incorporates a data structure abstraction known as DID, for \textit{distributed image data}. The DID-abstraction is intended as an image data type declaration, without exposing the actual distribution of data. For example, a DID structure for binary image data may be declared as:

\begin{verbatim}
Image2D_Binary bimage(Horizontal, "pict1.jpg"); 
\end{verbatim}

to indicate that a binary image "pict1.jpg" is read into a horizontally distributed image data structure, which can be referred to through \textit{bimage}. Although a DID declaration is easy to understand for programmers unfamiliar to parallel computing, it has the disadvantage of making the user responsible for the type of data distribution.

Another library-based approach applicable to both SIMD- and MIMD-style architectures is developed by Olk et al. [118]. The library provides a fully sequential interface to the user, and incorporates data parallel data structure abstractions such as images, kernels, neighborhoods, queues, buckets, etcetera. The programmer addresses a data structure as a single entity, with no concern of the implementation and parallel execution of an operation. However, to obtain efficient executables the user needs to implement in Compositional C++ [26] (see Section 2.2.1). Clearly, this is approach is not ideal, as it still requires the programmer to personally identify part of the available parallelism.

The library-based environment described by Jamieson et al. [73, 74, 75, 168] also provides a fully sequential interface to the user. At the heart of the environment is a set of algorithm libraries, along with abstract information about the performance characteristics of each library routine. In addition, the environment contains a dynamic scheduler for optimization of full applications, an interactive environment for developing parallel algorithms, and a graph matcher for mapping algorithms onto parallel hardware. Although this environment proved to be quite successful, its sustainability proved to be problematic. Partially, this is because it is required to provide \textit{multiple} implementations for an algorithm, one for each target parallel machine.

One data parallel environment that indeed can be considered fully user transparent is developed by Lee et al. [93]. An interesting aspect of this work is that it incorporates simple performance models to ensure efficiency of execution of complete applications. However, the environment is too limited in functionality to constitute a true solution, as it supports point operations and a small set of window operations only. Two similar environments, presented in [79, 80, 81] and [86, 87] respectively, are much more extensive in functionality. However, in both cases the performance models as designed in relation with the library operations are not used as a basis for optimization of complete programs, but serve as an indication to library users only.

An interesting environment based on the abstractions of Image Algebra [131], that to a large extent adheres to the requirements of user transparency, is described in [109]. It is targeted towards homogeneous MIMD-style multicomputers, and is implemented in a combination of C++ and MPI. One of the important features of this environment is the so-called \textit{self-optimizing class library}, which is extended automatically with optimized parallel operations. During program execution, a syntax
graph is constructed for each statement in the program, and evaluated only when an assignment operator is met. At first execution of a program, each syntax graph is traversed, and an instruction stream is generated and executed. In addition, any syntax graph for combinations of primitive instructions (i.e., those incorporated as a single routine within the library) is written out for later consideration by an off-line optimizer. On subsequent runs of the program a check is made to decide if an optimized routine is available for a given sequence of library calls. An important drawback of this approach, however, is that it may guarantee optimal performance of sequences of library routines, but not necessarily of complete programs.

The MIRTIS system, described in [108], is the most efficient and extensive library-based environment for user transparent parallel image processing designed to date. MIRTIS is targeted towards homogeneous MIMD-style architectures, and provides operations at the complete image level. Programs are parallelized automatically by partitioning sequential data flows into computational blocks, to be decomposed in either a spatial or a temporal manner. Issues related to data decomposition, communication routing, and scheduling are dealt with by using simple performance models. In the modeling of the execution time of a certain application, MIRTIS relies on empirically gathered benchmarks. Although, from a programmer’s perspective, MIRTIS constitutes an ideal solution, its implementation suffers from poor maintainability and extensibility. Also, the provided MIRTIS implementation suffers from reduced portability as the applied communication kernels are too architecture specific.

From this overview we conclude that, although several library-based user transparent systems exist, none of these is truly satisfactory. As indicated in the discussion, this is because it is not sufficient to offer user transparency as is. Issues relating to the design and implementation of a parallelization tool, such as maintainability, extensibility, and portability of the provided software library, play an important role as well. A discussion of these issues follows in the remainder of this chapter.

2.2.3 Discussion

In Figure 2.2 we have positioned all classes of parallelization tools presented in this section in a single effort-efficiency graph similar to that of Figure 2.1. The figure shows that the amount of effort required for using any type of general-purpose parallelization tool generally exceeds THRESHOLD 1 (the class of automatic parallelizing compilers being the only exception). Also, the higher the efficiency provided by such general-purpose tool, the higher the amount of effort required from the application programmer. Although the introduction of domain-specific knowledge reduces the required amount of user effort, parallel image processing languages are generally still too specialized for widespread acceptance. From the two classes of tools that are considered 'user-friendly' by the image processing community (i.e., automatic parallelizing compilers and parallel image processing libraries), only a small subset of all library-based tools provides a sufficiently high level of efficiency as well.

Despite the fact that some of the library-based systems adhere to all requirements of user transparency (especially those described by Lee et al. [93], Moore et al. [108], and Morrow et al. [109]), none of these has found widespread acceptance. One may
argue that this is due to the fact that they are still relatively new, and may need some more time to make a significant impact on the imaging community. We feel, however, that the tools still do not constitute a solution that is acceptable on the long term.

As we have discussed extensively in the previous sections, user transparency in itself is the decisive property that matches a tool’s programming model to the image processing researcher’s frame of reference. In this respect, any tool that adheres to the requirements of user transparency is acceptable in that it can always be used immediately, without much effort from the application programmer. However, a parallelization tool is not a static product. It is essential for such tool to be able to deal with new hardware developments and additional user requirements. If the design of a parallelization tool makes it ever more difficult or even impossible for its developers to respond to changing demands quickly and elegantly, users will loose interest in the product almost immediately.

If we refer back to the graph of Figure 2.1, perpendicular to the two dimensions of effort and efficiency we can add a third axis that represents a tool’s level of sustainability. This term incorporates all issues relating to the extensibility, maintainability, applicability, and portability of a given parallelization tool, and indicates how easily a tool can be adapted to changing demands and environments. As before, a critical threshold can be identified for the level of sustainability, below which no tool is expected to survive on the long term. We feel that none of the existing user transparent tools incorporates an acceptable sustainability level as well. For this reason we have designed a new parallelization tool that, apart from adhering to the requirements of full user transparency, also offers a sufficiently high level of sustainability.
2.3 A Sustainable Software Architecture for User Transparent Parallel Image Processing

The discussion of the applicability of existing hardware and software architectures in the field of image processing research has led to several important conclusions. First, the most appropriate class of hardware architectures to be applied in image processing research is that of Beowulf clusters — most importantly due to its emphasis on price-performance. Second, software development tools based on a library of pre-parallelized routines offer a solution that is most likely to be acceptable to the image processing community — especially because it has shown to be possible to provide such tool with a programming model that offers full user transparency. Finally, no user transparent tool currently exists that indeed provides an acceptable long term solution, as none incorporates a sufficiently high level of sustainability.

In this section we present an overview of our new library-based architecture for user transparent parallel image processing on homogeneous clusters. Due to its innovative design we expect the architecture to constitute an acceptable solution for the image processing community on the long term.

2.3.1 Architecture Requirements

We argue that a library-based software architecture, which is to serve as a parallelization aid for the image processing research community, must adhere to the following list of requirements:

1. *User transparency.* As discussed in Section 2.2, user transparency refers to a combination of 'user friendliness' and 'high efficiency'. For a library-based parallelization tool, this terminology translates to the following two requirements:

   1. **Availability of an extensive sequential API.** To ensure that the parallel library is of great value to the image processing community, it must contain an extensive set of data types and associated operations commonly applied in image processing research. The application programming interface (API) should disclose as little as possible information about the library’s parallel processing capabilities. Preferably, the API is made identical to that of an existing sequential image processing library.

   2. **Combined intra-operation efficiency and inter-operation efficiency.** It is essential for the software architecture to provide significant performance gains for a wide range of image processing application types. For this reason it is required to obtain a level of efficiency that generally compares well to that of 'reasonable' hand-coded parallel implementations. Efficiency, in this respect, refers to the execution of each library operation in isolation (*intra-operation efficiency*), as well as to the execution of multiple operations applied in sequence (*inter-operation efficiency*).
II. *Long term sustainability.* To ensure longevity, the design and implementation of the software architecture must be such that extensions are easily dealt with. In this respect, long term sustainability refers to the following four requirements:

3. *Architecture maintainability.* To minimize the coding effort in case of changing demands and environments, care must be taken in the architecture’s design to avoid unnecessary code redundancy, to enhance operation reusability. In this respect, it is preferable to implement any set of operations with similar behavior as a single generic routine, to be instantiated at will to obtain the desired functionality. Also, to avoid implementing operations for all data types generic implementations are preferred.

4. *Architecture extensibility.* As no library can contain all functionality applied in image processing research, it is required to allow the user to insert new operations. In case an additional operation maps onto a generic operation present in the library, insertion should be straightforward, not requiring any parallelization effort from the user.

5. *Applicability to homogeneous Beowulf clusters.* As we have identified clusters as the most appropriate type of hardware architecture for image processing research (see Section 2.1), the complete software architecture must be applicable to this type of machines. All general and distinctive properties of such machines can therefore explicitly be incorporated in the implementation of the software architecture. Optimized functionality for any other machine type should not be incorporated.

6. *Architecture portability.* To ensure portability to all target machines it is essential to implement the software architecture in a high-level language such as C or C++. For any constituent component in a cluster a high quality C or C++ compiler is generally available — and upgrades are released frequently. Although the properties of Beowulfs can be incorporated in all implementations, care should be taken not to incorporate any assumptions about a specific interconnection network topology.

### 2.3.2 Architecture Overview

The complete software architecture consists of six components (see Figure 2.3). This section presents a general overview of each of the components, and design choices are related to the requirements of Section 2.3.1.

**Component 1: Parallel Image Processing Library**

The core of our software architecture consists of an extensive software library of data types and associated operations commonly applied in image processing research. In accordance with the first requirement of Section 2.3.1, the library’s application programming interface is made identical to that of an existing sequential image processing library: Horus [84]. More specifically, rather than implementing a completely new
library from scratch, the parallel functionality is integrated with the Horus implementation in such a manner that all existing sequential code remains intact. Apart from reducing the required parallel implementation effort, this approach has the advantage that the important properties of the Horus library (i.e., maintainability, extensibility, and portability) to a large extent transfer to the parallel version of the library as well.

Similar to other libraries discussed in Section 2.2.2, the sequential Horus implementation is based on abstractions of Image Algebra [131], a mathematical notation for specifying image processing algorithms. Image Algebra is an important basis for the design of an extensive, maintainable, and extensible image processing library, as it recognizes that a small set of operation classes can be identified that covers the bulk of all commonly applied image processing functionality. Within the Horus library each such operation class is implemented as a generic algorithm, using the C++ function template mechanism [158]. 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. From this, it follows that the desired architectural properties of maintainability, extensibility, and portability, constitute an integral aspect the Horus design. As will be discussed in more detail in Chapter 3, the Horus library also covers a large majority of all common image processing operations. As a result, Horus fully adheres to requirements 1, 3, 4, and 6 of Section 2.3.1.

In extending the Horus library for parallel operation we have focused on adhering to the remaining requirements 2 and 5: i.e., the architecture’s efficiency and its applicability to Beowulfs. To this end, and also to have full control over the communication behavior of the library operations, the parallel extensions are implemented using MPI [104]. Also, to sustain a high maintainability level, each parallel image processing operation is implemented by concatenating data communication routines with sequential code blocks from the Horus library. In this manner, the source code for each sequential generic algorithm is fully reused in the implementation of its parallel counterpart, thus avoiding unnecessary code redundancy as much as possible. For a more detailed description of the library implementation, we refer to Chapter 3.

![Figure 2.3: Simplified architecture overview.](image-url)
The design and implementation of the parallel library ensures that our parallelization tool adheres to all requirements of Section 2.3.1, with the exception of requirement 2. To also guarantee efficiency of execution of (1) operations that are applied in isolation, and (2) applications or algorithms that contain sequences of library operations, five additional architectural components are designed and implemented in close connection with the software library itself. These additional components are described in the remainder of this section.

**Component 2: Performance Models**

In contrast to other library-based environments (e.g., [75]), our library contains not more than one parallel implementation for each generic algorithm. To still guarantee intra-operation efficiency on all target platforms, the parallel generic algorithms are implemented such that they are capable of adapting to the performance characteristics of the parallel machine at hand. As an example, the manner in which data structures are decomposed at run time is not fixed in the implementations, as the efficiency of each decomposition type may differ for each specific target machine. Also, the optimal number of processing units may vary.

To make a machine’s performance characteristics explicit, each library operation is annotated with a domain specific performance model. For applicability to clusters, the models are based on an abstract machine definition (the APIM, or: Abstract Parallel Image Processing Machine) that captures the hardware and software aspects of image processing operations executing on such a system. An overview of the APIPM, as well as a formal definition of the APIPM-based models for sequential operation, is presented in Chapter 4. A detailed description of the model that captures the additional communication aspects of parallel execution is given in Chapter 5.

**Component 3: Benchmarking Tool**

Performance values for the model parameters are obtained by running a set of benchmarking operations that is contained in a separate architectural component. The combination of the high-level APIPM-based performance models and the specialized set of benchmarking routines allows us to follow a semi-empirical modeling approach, that has proven to be highly successful in other research as well (e.g., see [108, 172]). In this approach, essential but implicit cost factors are incorporated by performing actual experiments on a small set of sample data. Apart from its relative simplicity, the main advantage of the semi-empirical modeling approach is that it fully complies with the requirements of applicability and portability to clusters. The performance models and benchmarking results allow intra-operation optimization to be performed automatically, fully transparent to the user. This optimization is performed by the architecture’s scheduling component, described below.

Chapter 4 gives a thorough description of the approach of semi-empirical modeling, as well as an overview of the benchmarking strategy applied for the measurement of sequential operations. An overview of the measurement strategy relating to the communication aspects of parallel execution is given in Chapter 5.
Component 4: Database of Benchmarking Results

All benchmarking results are stored in a database of performance values. Although the design and implementation of such database is of significant importance (especially in case it must be accessed frequently at run time), this topic is too far outside the scope of this thesis for extensive discussion.

Component 5: Program Specification

Apart from incorporating an intra-operation optimization strategy, to obtain high efficiency it is essential to perform inter-operation optimization (or: optimization across library calls) as well. As it is often possible to combine the communication steps of multiple library operations applied in sequence, the cost of data transfer among the nodes in a parallel machine generally can be reduced considerably. Our architecture performs inter-operation optimization in case global 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, however, we currently assume that a complete algorithm specification is provided in addition to the program itself. Such specification closely resembles a concatenation of library calls, and does not require any parallelism to be introduced by the application programmer.

Component 6: Scheduler

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. The scheduler performs the tasks of intra-operation optimization and inter-operation optimization by removing all redundant communication steps, and by choosing: (1) the logical processor grid to map data structures onto (i.e., the actual domain decomposition), (2) the routing pattern for the distribution of data, (3) the number of processing units, and (4) the type of data distribution (e.g., broadcast instead of scatter).

As described in detail in Chapter 6, the scheduler's task of automatically converting any sequential image processing application into a correct and efficient parallel version, is performed on the basis of a simple finite state machine definition. First, the finite state machine allows for a straightforward and cheap run time method (called lazy parallelization) for communication cost minimization. If desired, the scheduler can be instructed to perform further optimization at compile-time. In this case, the finite state machine is used in the construction of an application state transition graph, that fully characterizes an application's run time behavior, and incorporates all possible parallelization and optimization decisions. As each decision is annotated with a run time cost estimation obtained from the APIPM-based performance models, the fastest version of the program is represented by the cheapest branch in the application state transition graph. In the library implementation of each parallel generic algorithm, requests for scheduling results are performed in order to correctly execute the optimizations prescribed by the application state transition graph.
2.4 Conclusions

In this chapter, we have investigated the applicability of existing hardware and software architectures in the field of image processing research. Based on a set of architecture requirements we have indicated that homogeneous Beowulf clusters constitute the most appropriate class of target platforms for application in image processing research. Apart from the fact that many references exist in the literature indicating significant performance gains for typical image processing applications executing on clusters, the foremost reason for favoring such architectures over other appropriate systems was found to be the fact that these deliver the best combination of price and performance.

Our investigation of software tools for implementing image processing applications on clusters has shown that library-based parallelization tools offer a solution that is most likely to be acceptable to the image processing research community. First, this is because such tools allow the programmer to write applications in a familiar programming language, and make use of the high level abstractions as provided by the library. More importantly, this is because library-based environments are most easily provided with a programming model that offers full user transparency — or, in other words: sufficiently high levels of 'user friendliness' and 'efficiency of execution'. Due to insufficient sustainability levels, no existing user transparent tool was found to provide an acceptable long term solution as well.

On the basis of these considerations we have proposed a new library-based software architecture for parallel image processing on clusters. We have presented a list of requirements such tool must adhere to for it to serve as an acceptable long term solution. In addition, we have given an overview of each of the architecture's constituent components, and we have touched upon the most prominent design issues for each of these. The architecture's innovative design and implementation ensures that it fully adheres to the requirements of user transparency and long term sustainability. Consequently, we believe our architecture for user transparent parallel image processing to constitute an acceptable long term solution for the image processing research community at large.