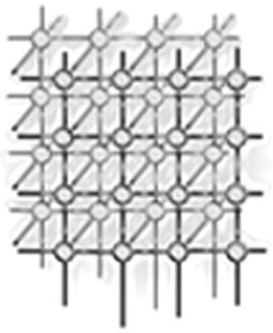
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:611–644 (DOI: 10.1002/cpe.765)

User transparency: a fully sequential programming model for efficient data parallel image processing



F. J. Seinstra*,† and D. Koelma

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

SUMMARY

Although many image processing applications are ideally suited for parallel implementation, most researchers in imaging do not benefit from high-performance computing on a daily basis. Essentially, this is due to the fact that no parallelization tools exist that truly match the image processing researcher's frame of reference. As it is unrealistic to expect imaging researchers to become experts in parallel computing, tools must be provided to allow them to develop high-performance applications in a highly familiar manner. In an attempt to provide such a tool, we have designed a software architecture that allows transparent (i.e. sequential) implementation of data parallel imaging applications for execution on homogeneous distributed memory MIMD-style multicomputers. This paper presents an extensive overview of the design rationale behind the software architecture, and gives an assessment of the architecture's effectiveness in providing significant performance gains. In particular, we describe the implementation and automatic parallelization of three well-known example applications that contain many fundamental imaging operations: (1) template matching; (2) multi-baseline stereo vision; and (3) line detection. Based on experimental results we conclude that our software architecture constitutes a powerful and user-friendly tool for obtaining high performance in many important image processing research areas. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: data parallel image processing; software architecture design; performance evaluation

1. INTRODUCTION

To satisfy the performance requirements of current and future applications in image and video processing, the imaging community at large exhibits an overwhelming desire to employ the speed

*Correspondence to: F. J. Seinstra, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands.
†E-mail: fjseins@science.uva.nl

Received 4 October 2002
Revised 30 January 2003
Accepted 10 March 2003

potential of high-performance computer systems [1]. Also, it has been recognized for years that the application of parallelism in image processing can be highly beneficial [2]. As a consequence, collaboration between the research communities of high-performance computing and image processing has been commonplace, and often resulted in hardware configurations capable of executing sets of domain-specific routines [3–5]. Yet, in spite of the significant achievements in this direction, the application of high-performance computing in imaging research is not widespread.

Primarily, we ascribe this problem to the high threshold associated with the use of high-performance computing architectures. One determinative factor for the existence of this threshold is the relatively high cost involved in using a specific parallel machine. More importantly, the threshold exists due to the single characteristic parallel and distributed computers have in common: they are much harder to program than traditional sequential systems. Although several attempts have been made to alleviate the problem of software design for parallel and distributed systems, as of yet no single solution has been provided that has found widespread acceptance.

As will be discussed extensively in this paper, the latter problem boils down to the fact that no efficient parallelization tool exists that is provided with a programming model that truly matches the image processing researcher's frame of reference. More specifically, most existing software development tools require the user to identify the available parallelism, often at a level of detail that is beyond the expertise of most image processing researchers. As it is unrealistic to expect researchers in imaging to become experts in parallel computing, it is essential to provide an alternative tool that offers a much more 'familiar' programming model. In this paper we argue that a parallelization tool for the image processing community is acceptable only if it hides all parallelism from the application programmer and still produces highly efficient code in most cases. Stated differently, we argue that the only programming model that will be considered 'familiar' by the imaging community is the model that offers complete *user transparent parallel image processing*.

In the literature several solutions have been described that—to a certain extent—allow user transparent implementation of high-performance image processing applications. In all cases, solutions are 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 problems that hinder a widespread acceptance. Most significantly, the efficiency of parallel execution of complete image processing applications often is far from optimal. In addition, in many cases the provided software library does not incorporate a sufficiently high level of *sustainability*, which dramatically reduces chances of long-term success.

Given these observations, the primary research issue addressed in this paper is formulated as follows: how can the image processing research community be provided with a fully sequential programming model that allows implementation of efficient parallel image processing applications in a way that excludes the user from acquiring any additional skills related to parallelism? In this paper we present a new and innovative software architecture for user transparent parallel image processing that is specifically designed to deal with these (and other) issues. We discuss the requirements put forward for such a programming tool, and provide an extensive overview of the separate components that constitute the software architecture's design. In addition, we present an extensive overview of measurement results, to assess the architecture's effectiveness in providing significant performance gains.

This paper is organized as follows. Section 2 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 3 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 is introduced in Section 4. In Section 5, we give an assessment of the software architecture's effectiveness in providing significant performance gains. In particular, we describe the implementation and automatic parallelization of three well-known example applications that contain many operations commonly applied in image processing research: (1) template matching; (2) multi-baseline stereo vision; and (3) line detection. Concluding remarks are given in Section 6.

2. HARDWARE ARCHITECTURES

A parallelization tool that is to be used 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. We have defined several requirements such a machine should adhere to. These are formulated as follows.

- *Wide availability*. To ensure that the image processing community at large can benefit from a specific parallelization tool, it is essential that the target platform is widely available and used. Less popular or experimental architectures tend to suffer from a lack of continuity, thus hindering the ever present desire for hardware upgrades.
- *Easy accessibility*. The target platform should be easily accessible to the imaging researcher. This refers to (1) the manner in which one logs on to the machine (if at all), (2) how programs are to be compiled and run, and (3) how easy it is to obtain a set of processing elements.
- *Silent upgradability*. It is essential that all software developed for the target platform still should be executable without any additional programming effort after each upgrade to the next generation of the same architecture type. 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 operations that are most commonly applied in image processing research. Clearly, if no significant performance improvements are to be expected, the whole 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 image processing software on the target platform should be kept to a minimum. As high-performance computing is not a goal in itself in image processing research, the maximum amount of money that may be spent on computing resources is small compared with the amount of money that flows to more fundamental research.

We are aware of the fact that different or 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 researchers (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 paper, 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.

As described in [6], many *general purpose* parallel architectures (ranging from the Cray family of vector machines, to popular distributed memory multicomputers such as the IBM SP-2, as well as the many shared memory multiprocessors such as the SGI Origin 2000) are potential candidates for high-speed execution of image processing applications. Also, many *special purpose* architectures (e.g., ASICs [7,8], FPGAs [9,10], DSPs [3,11]), and enhanced general purpose CPUs (see, e.g., [12–14]), have been developed to deliver even higher performance for specific imaging tasks [15].

Irrespective of the significance of many of these architectures, from the set of general-purpose platforms one architecture type stands out as particularly interesting for our purposes, i.e. the class of *Beowulf-type commodity clusters* [16,17]. Like most other general-purpose systems mentioned above, Beowulf clusters fully adhere to the stated list of requirements. An important additional advantage is that the bulk of all image processing research is currently being performed on machines similar to the constituent nodes of such clusters. The single characteristic that makes Beowulf clusters favorable, however, is that they deliver better 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 (see, e.g., [18–21]), we conclude that Beowulf clusters constitute the most appropriate target platforms for our specific needs.

3. SOFTWARE DEVELOPMENT TOOLS

Apart from its design and capabilities, the (commercial) success of any computer architecture significantly depends on the availability of tools that simplify 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, for many of the architectures referred to in Section 2, available high-level language compilers often have great difficulties in generating assembly code that makes use of the capabilities of a parallel machine effectively. To obtain truly *efficient* code 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 different programming tools have been developed that attempt to alleviate the problem of low-level software design for parallel and distributed computers. In all cases such tools are provided with a programming model that—to a certain extent—abstracts from the idiosyncrasies of the underlying parallel hardware. However, the small user base of parallel computing within the image processing research community indicates that (as of yet) no parallelization tool has been provided that has a truly 'familiar' programming model.

The ideal solution would be to have a parallelization tool that abstracts from the underlying hardware completely, and that allows 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 generally believed that no such tool probably will be developed ever at all. 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

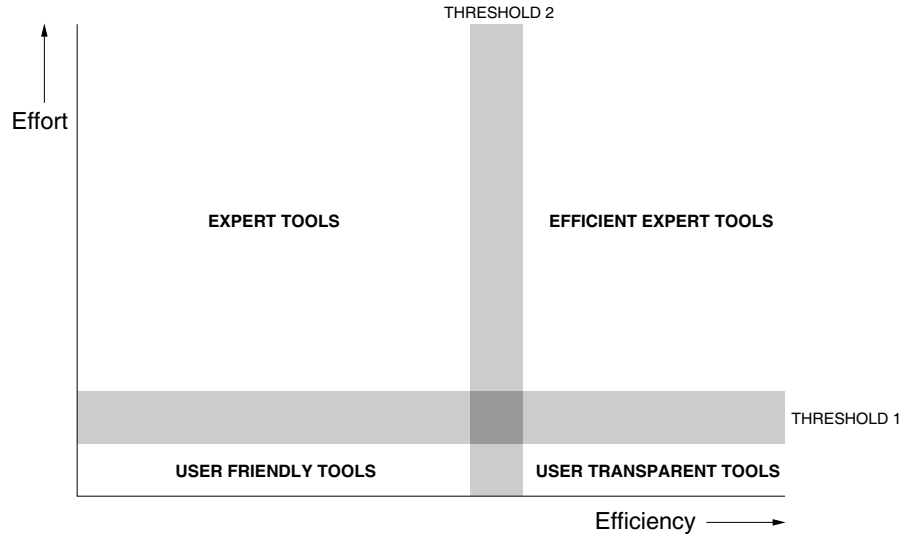


Figure 1. Parallelization tools: effort versus efficiency. User transparent tools constitute a subset of all tools in that these are both user friendly and highly efficient.

such a tool largely depends on the amount of *effort* that is required from the application programmer and the level of *efficiency* that is obtained in return.

The graph of Figure 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 that has been 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* that is 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 that is required to obtain a parallel program that is particularly *efficient*.

In Figure 1 the maximum amount of effort an image processing researcher is generally willing to invest into the implementation of efficient parallel applications is represented by `THRESHOLD 1`. This threshold divides the set of all parallelization tools into two subsets, one containing tools that can be considered 'user friendly', and the other containing 'expert tools' that require knowledge far beyond the expertise of most imaging researchers. The minimum level of efficiency a user generally expects as a return on investment is depicted by `THRESHOLD 2`. This threshold further divides each subset into two sets, each containing tools that are considered either sufficiently efficient or not efficient enough. To show 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 1.

In this paper, each tool that is considered both 'user friendly' and 'sufficiently efficient' by the image processing community is referred to as a tool that offers full *user transparent parallel image processing*.

Apart from adhering to certain levels of required 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). In general, the efficiency as obtained by a user transparent tool is considered sufficiently high—independent of the actual application that is being implemented. 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*.

In the following we give an overview of the most significant development tools that (a.o.) can be used for implementing parallel image processing applications on Beowulf-type commodity clusters. For each tool we discuss the level of abstraction that is incorporated in the programming model, and assess to what extent it adheres to the properties of full user transparency.

3.1. General purpose parallelization tools

This section presents an overview of tools that can be used for implementing high-performance image processing software, but that are not tailored to this application area specifically.

3.1.1. Message passing libraries

Good examples of tools from the set of *efficient* programming aids for *experts* in parallel computing are the many software libraries that provide *message passing* functionality. Many efficient and portable message passing systems have been described in the literature [21], but the sets of library routines provided by PVM [22] and MPI [23] have become the most widely used.

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, it is generally recognized as being difficult to do correctly [24]. This is mainly 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 easily introduced, and debugging is often hard due to potential race-conditions. Consequently, for the image processing community, even the minimum amount of effort that is required to write message passing programs far exceeds the maximum amount a user generally is willing to invest (i.e. `THRESHOLD 1` in Figure 1).

3.1.2. Shared memory specifications

As message passing was initially intended for client/server applications running across a network, the MPI standard includes costly semantics (e.g., the assumption of wholly separate memories) that are often not required on parallel systems with a globally addressable memory. In an attempt to provide a simpler, more efficient, and portable approach to running programs on shared memory architectures, OpenMP [25,26] has been proposed as a standard. OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.

Although a Beowulf-type commodity cluster as a whole does not fit the class of shared-memory architectures (its constituent nodes may do), it is still relevant to include OpenMP in this evaluation.

First, this is because it is possible to implement OpenMP for a distributed memory system on top of MPI, albeit at the cost of somewhat higher latencies (see, e.g., [27]). Second, the program development philosophy of OpenMP is generally believed to be simpler than that of MPI [28].

One of the major advantages of the OpenMP approach 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—and erroneous programs. As a result, the amount of effort that is generally required from the user to obtain a correct parallel program still exceeds `THRESHOLD 1` in Figure 1.

3.1.3. 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 such as C/C++, Fortran, or Java. Probably the most popular example of a language that has adopted this approach is HPF (High Performance Fortran [29]). Also, many alternative extensions and modifications to C++ exist [30], of which Compositional C++ [31] and Mentat [32] 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 (parallel) 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 generally are not capable of detecting such non-optimal behavior automatically [33,34], 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 1.

3.1.4. Parallel languages

Instead of 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. As a result, many parallel languages have been described in the literature (e.g., Ada [35], Occam [36], and Orca [33]).

Despite years of intensive research, no parallel language has truly found widespread acceptance (either in the imaging community or elsewhere). One important reason is that it was found to be extremely difficult to design language features that are both generally applicable and easy to use (see Pancake and Bergmark [37]). A much more important reason, however, is that most scientific programmers are generally reluctant to learn an entirely new program development philosophy, or unfamiliar language constructs. Given the fact that the parallelism in any parallel language is always explicit, and fine-tuning is often an inherent part of the program development process, the amount of effort required from the user again exceeds `THRESHOLD 1` in Figure 1.

3.1.5. *Fully automatic parallelizing compilers*

As opposed to the parallelization tools discussed so far, an efficient *automatic parallelizing compiler* indeed 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 [38]. However, a fundamental problem is that many user-defined algorithms contain data dependencies that prevent efficient parallelization. This problem is particularly severe for languages (such as C/C++) that support pointers [39]. In addition, techniques for automatic dependency analysis and algorithm transformation are still in their infancy, and often far from optimal. 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 [40]), 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 [34].

3.2. Tools for parallel image processing

The regular evaluation patterns in many 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 relatively 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 much higher-level abstractions to the user than general-purpose tools, and are potentially much more efficient as important domain-specific assumptions often can be incorporated.

3.2.1. *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 [41] 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, and smoothing. It is based on the observation that many image operations are of the form:

```
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. 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. To overcome this problem, in a different language (Adapt [42]) the basic principles of Apply were 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's programming model, however, was not ideal as the programmer is made responsible for data partitioning and merging, albeit at quite a high level.

An alternative approach is taken in a language called IAL (Image Algebra Language [43]). IAL is based on the abstractions of Image Algebra [44], 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 operator is implemented in IAL as a single statement:

```
OutputIm := (abs(InputIm ⊕ S_h) + abs(InputIm ⊕ S_v)) >= threshold;
```

where S_h and S_v are the horizontal and vertical Sobel masks, and $\oplus$ represents convolution. The language proved to be useful for a wide range of image processing tasks, but was limited in its flexibility and expressive power. Two extended versions of IAL, I-BOL [45] and Tulip [46], provide a more flexible and powerful notation. As well as providing the 'complete image' notation as a subset, these languages permit access to data at either the pixel or 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.

3.2.2. *Parallel image processing libraries*

An alternative to the language approach is to provide an extensive set of parallel image processing operations in library form. 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 briefly discuss the most important ones.

One particularly interesting data parallel library implementation is described by Taniguchi *et al*. [47]. This software platform is applicable to both SIMD- and MIMD-style architectures and incorporates a data structure abstraction known as DID (or, *distributed image data*). The DID-abstraction is intended to be an image data type declaration, without exposing the details of the actual distribution of data. For example, a DID structure for a binary image may be declared as:

```
Image2D_Binary bimage(Horizontal, "pict1.jpg");
```

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

The alternative library-based environment described by Jamieson *et al*. [18] does provide 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 (1) a dynamic scheduler for optimization of full applications, (2) an interactive environment for developing parallel algorithms, and (3) a graph matcher for mapping algorithms onto parallel architectures. Although this environment proved to be quite successful, its sustainability proved to be problematic. Partially, this is because it is required to provide *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*. [20]. 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 environments that follow a similar approach, but are both much more extensive in functionality, are presented in [19,48] respectively. 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 [44], that to a large extent adheres to the requirements of user transparency, is described in [49]. 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 its so-called *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. In case this is the first time the program is executed, 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 [50], is probably 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, which are decomposed in either a spatial or a temporal manner. All issues related to data decomposition, communication routing and scheduling are dealt with by using simple domain-specific 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.

Based on this overview we come to the conclusion that, although several library-based user transparent systems exist, none of these is truly satisfactory. As indicated above, this is because it is not sufficient to merely offer full user transparency. Issues relating to the *design* and *implementation* of a parallelization tool (such as maintainability, extensibility and portability of the provided software library) are important as well. All of these issues will be discussed in more detail in the remainder of this paper.

3.3. Discussion

In Figure 2 we have positioned all classes of parallelization tools presented in this section in a single effort-efficiency graph similar to that of Figure 1. It should be noted that this figure presents a generalized view only. Consequently, the relative distances between the classes of parallelization

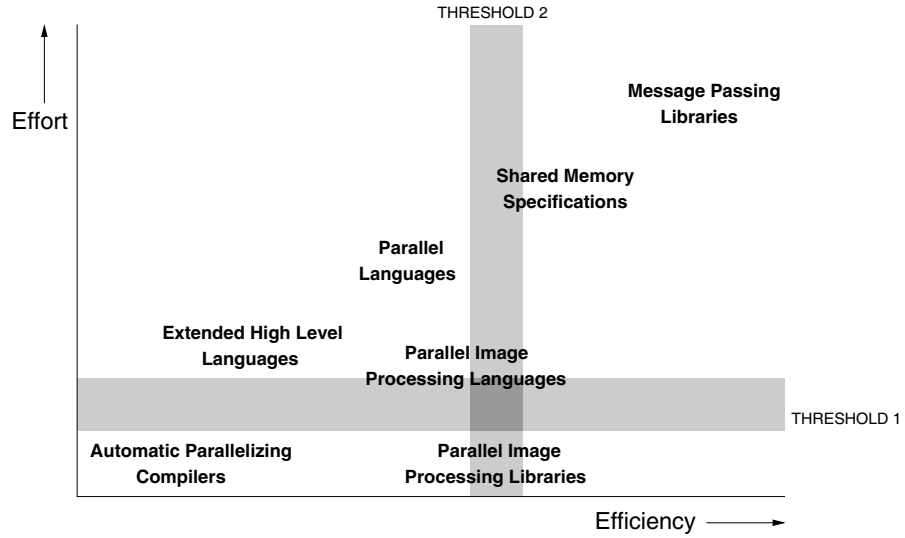


Figure 2. Generalized view of effort versus efficiency of existing parallelization tools. From the set of tools only a subset of all parallel image-processing libraries can be considered truly user transparent.

tools is merely an indication. Also, in reality there may be a certain amount of overlap between the different types of tools. The figure shows that the amount of effort required for using any type of *general-purpose* parallelization tool generally exceeds `THRESHOLD 1` (the exception being the class of automatic parallelizing compilers). Also, the higher the efficiency that potentially is provided by such a general-purpose tool, the higher the amount of effort that is required from the application programmer. Although the introduction of domain-specific knowledge reduces the required amount of user effort (due to the possibility of providing higher level abstractions), 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.* [20], Moore *et al.* [50], and Morrow *et al.* [49]), none of these has been widely accepted by the image processing research community. One may argue that this is due to the fact that the tools are still relatively new, and may need some more time to make a significant impact on the image processing community. We feel, however, that the tools still do not constitute a solution that is acceptable *in the long term*. This is because 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 1, perpendicular to the two dimensions of effort and efficiency we can add a third axis that represents a tool's level of *sustainability* (defined more specifically in the next section). As before, a critical threshold can be identified for the level of sustainability, below which no tool (not even one that provides full user transparency) is expected to survive in the long term. As implicitly stated before, we feel that none of the existing tools that do provide full user transparency 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.

4. A SUSTAINABLE SOFTWARE ARCHITECTURE FOR USER TRANSPARENT PARALLEL IMAGE PROCESSING

This section presents an overview of our library-based architecture for user transparent parallel image processing on homogeneous Beowulf-type commodity clusters. The overview starts with a detailed list of architecture requirements. This is followed by a description of each of the architecture's constituent components. For a much more detailed overview we refer to [51].

4.1. Architecture requirements

From the observations of the previous sections we conclude that a library-based parallelization aid for the image processing research community should adhere to the following requirements.

I. *User transparency*. As discussed in Section 3, user transparency refers to a combination of 'user friendliness' and 'high efficiency'. For a library-based parallelization tool, this terminology translates into 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 research community, it must contain an extensive set of commonly used data types and associated operations. The application programming interface should disclose as little as possible information about the library's parallel processing capabilities. Preferably, the API is to be made identical to that of an existing *sequential* image processing library that has gained substantial acceptance.
   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 applications. Also, it is required to obtain a level of efficiency that compares well with that of hand-coded parallel implementations. Efficiency, in this respect, refers to the execution of each library operation in isolation (i.e. *intra-operation efficiency*), as well as to multiple operations applied in sequence (i.e. *inter-operation efficiency*).

II. *Long term sustainability*. To ensure longevity, the architecture's design must be such that extensions are easily dealt with. This refers to the following requirements.

   3. *Architecture maintainability by code genericity*. To minimize the coding effort in case of changing demands and environments, care must be taken in the architecture's implementation to avoid unnecessary code redundancy, and to enhance operation reusability. To this end, it is essential to rely on *code genericity* in the implementation of each set of operations with comparable behavior.

4. *Architecture extensibility*. As no library can ever contain all functionality applied in imaging, it is required to allow the user to insert new operations and data types. In case an additional operation maps onto a generic operation present in the library, insertion should be straightforward and should not require any parallelization effort from the user.
5. *Applicability to Beowulf-type commodity clusters*. As we have identified the class of homogeneous Beowulf clusters to be the most appropriate hardware architecture for image processing research, the software architecture must *at least* be applicable to this type of machines. All general and distinctive properties of such machines (e.g., the homogeneity of its constituent components) 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 type of node that may serve as constituent component in a Beowulf-type cluster a high-quality C or C++ compiler is generally available—and upgrades are released frequently. Although the general properties of Beowulf-type commodity clusters can be incorporated explicitly in all implementations, care should be taken not to incorporate any assumptions about a specific interconnection network topology.

4.2. Architecture overview

This section presents an overview of each of our software architecture's constituent components (see Figure 3), and design choices are related to the aforementioned requirements.

Component 1: parallel image processing library

The core of our 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 4.1, the library's application programming interface is made identical to that of an existing sequential library: Horus [52]. 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 major 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 3.2, the sequential Horus implementation is based on the abstractions of Image Algebra [44], a mathematical notation for specifying image processing algorithms. Image Algebra is an important basis for the design of a 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* [53]. Each operation that maps onto the functionality as provided by such an algorithm is implemented by instantiating the generic algorithm with the proper parameters, including the function to be applied to the individual data elements. Currently, the following set of generic algorithms

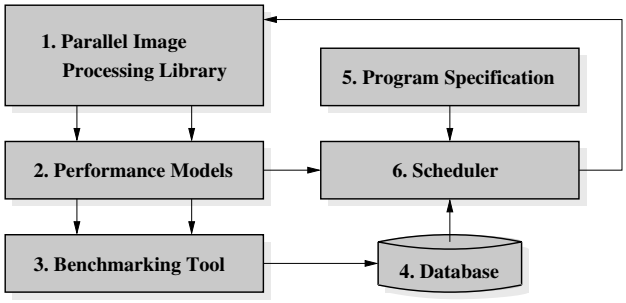


Figure 3. Simplified architecture overview.

is available: (1) *unary pixel operation*, e.g. negation, absolute value; (2) *binary pixel operation*, e.g. addition, threshold; (3) *global reduction*, e.g. sum, maximum; (4) *neighborhood operation*, e.g. percentile, median; (5) *generalized convolution*, e.g. erosion, gauss; and (6) *geometric (domain) operation*, e.g. rotation, scaling. In the future additional generic algorithms will be added, e.g. iterative and recursive neighborhood operations, and queue based algorithms.

As the properties of the sequential Horus library fully match requirements 1, 3, 4, and 6 of Section 4.1, in the implementation of the parallel extensions we can focus on adhering to the remaining requirements 2 and 5, whilst avoiding the introduction of violations to any of the other requirements. In this respect, to ensure applicability and portability of the library to homogeneous Beowulf-type commodity clusters, and also to have full control over the communication behavior (and thus: efficiency) of the library operations, the parallel extensions are implemented using MPI [23]. Also, to sustain a high maintainability level, each parallel image processing operation contained in the library is implemented simply by concatenating data communication routines with fully sequential code blocks that are separately available within 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 [51].

Essentially, the design of the parallel library ensures that our parallelization tool adheres to all requirements of Section 4.1, with the exception of requirement 2. To also guarantee efficiency of execution of operations that are applied in isolation, as well as complete applications, 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., [18]), our library contains not more than one parallel implementation for each operation. 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 to be decomposed at run time is not fixed in the implementations. Similarly, the optimal number of processing units to be applied in a calculation may vary.

To make a machine's performance characteristics explicit, each library operation is annotated with a domain specific performance model. Due to the intended portability of the library to Beowulf-type systems, the models must be *applicable* to all such machines as well. To this end, the models are based on an abstract machine definition (the *APIPM*, or Abstract Parallel Image Processing Machine), that captures the relevant hardware and software aspects of image processing operations executing on a Beowulf-type cluster. An overview of the APIPM, as well as a formal definition of the APIPM-based models for *sequential operation*, is presented in [51]. A description of the model that accurately models the additional costs of *communication* is given in [54].

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 successful in other research as well (e.g., see [50]). In this approach, essential but implicit cost factors (e.g., related to cache size) 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 important requirements of applicability and portability to Beowulf-type clusters.

The performance models and benchmarking results allow intra-operation optimization to be performed automatically, fully transparent to the user. This type of optimization is performed by the architecture's scheduling component, described below. A detailed description of the approach of semi-empirical modeling, and the applied benchmarking strategy is given in [6,51].

Component 4: database of benchmarking results

All benchmarking results are stored on a database of performance values. Although the design and implementation of such a 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 paper for extensive discussion.

Component 5: program specification

Apart from incorporating an intra-operation optimization strategy, to obtain high efficiency it is of great importance to perform the additional task of inter-operation optimization (or optimization across library calls) as well. As it is often possible to combine the communication steps of multiple operations applied in sequence, the cost of data transfer among the nodes generally can be reduced dramatically. Essentially, the information required to perform such optimization correctly can be obtained from the original program code. As implementation of a complete parser is not an essential part of this research, however, we currently assume that an algorithm specification is provided in addition to the program itself. Such a 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 deciding on all issues regarding: (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 [6], the scheduler's task of automatically converting any sequential image processing application into a legal, correct, and efficient parallel version of the same program, is performed on the basis of a simple finite state machine (fsm) definition. First, the fsm 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 fsm is used in the construction of an application state transition graph (*ASTG*), 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 ASTG. 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 ASTG.

5. ARCHITECTURE EVALUATION

In the remainder of this paper we give an assessment of the effectiveness of the presented software architecture in providing significant performance gains. To this end, we describe the implementation and automatic parallelization of three well-known example applications that contain many operations commonly applied in image processing research: (1) template matching; (2) multi-baseline stereo vision; and (3) line detection.

All of the applications described in this section have been implemented and tested on the 128-node homogeneous Distributed ASCI Supercomputer (DAS) cluster located at the Vrije University in Amsterdam [16]. This is a typical example of a machine from the class of homogeneous Beowulf-type commodity clusters as described in Section 2. All nodes in the cluster contain a 200 MHz Pentium Pro with 128 MB of EDO-RAM, and are connected by a 1.2 Gbit/sec full-duplex Myrinet SAN network. At the time of measurement, the nodes ran the RedHat Linux 6.2 operating system. The software architecture was compiled using `gcc` 3.0 (at highest level of optimization) and linked with MPI-LFC [55], an implementation of MPI which is partially optimized for the DAS. The required set of benchmarking operations was run on a total of three DAS nodes, under identical circumstances as the complete software architecture itself. At the time of measurement, 8 nodes in the DAS cluster were unusable due to a malfunction in the related network cards. As a consequence, performance results are presented for a system of up to 120 nodes only.

5.1. Template matching

Template matching is one of the most fundamental tasks in many image processing applications. It is a simple method for locating specific objects within an image, where the template (which is, in fact,

an image itself) contains the object one is searching for. For each possible position in the image the template is compared with the actual image data in order to find subimages that match the template. To reduce the impact of possible noise and distortion in the image, a similarity or error measure is used to determine how well the template compares with the image data. A match occurs when the error measure is below a certain predefined threshold.

In the example application described here, a large set of electrical engineering drawings is matched against a set of templates representing electrical components, such as transistors, diodes, etc. Although more post-processing tasks may be required for a truly realistic application (such as obtaining the actual positions where a match has occurred), we focus on the template matching task, as it is by far the most time-consuming. This is especially so because, in this example, for each input image f error image ε is obtained by using an additional *weight* template w to put more emphasis on the characteristic details of each 'symbol' template g:

$$\varepsilon(i, j) = \Sigma_m \Sigma_n ((f(i+m, j+n) - g(m, n))^2 \cdot w(m, n)) \tag{1}$$

When ignoring constant term $g^2 w$, this can be rewritten as:

$$\varepsilon = f^2 \otimes w - 2 \cdot (f \otimes w \cdot g) \tag{2}$$

with $\otimes$ the convolution operation. The error image is normalized such that an error of zero indicates a perfect match and an error of one a complete mismatch. Although the same result can be obtained using the fast Fourier transform (which has a better theoretical run time complexity, and also provides immediate localization of the best match and all of its resembling competitors), this brute force method is fastest for our particular data set.

5.1.1. Sequential implementation

Figure 4 is a sequential pseudo code representation of Equation (2). Essentially, each input image is read from file, squared (to obtain f^2), and matched against all symbol and weight templates, which are also obtained from file. In the inner loop the two convolution operations are performed and the error image is calculated and written out to file.

5.1.2. Parallel execution

As all parallelization issues are shielded from the user, the pseudo code of Figure 4 directly constitutes a program that can be executed in parallel as well. Efficiency of parallel execution depends on the optimizations performed by the architecture's scheduling component. For this particular sequential implementation, the optimization process has generated a schedule that requires only four different communication steps to be executed. First, each input image read from file is scattered throughout the parallel system. Next, in the inner loop all templates are broadcast to all processing units. Also, in order for the convolution operations to perform correctly, image borders (or *shadow regions*) are exchanged among neighboring nodes in the logical CPU grid. In all cases, the extent of the border in each dimension is half the size of the template minus one pixel. Finally, before each error image is written out to file it is gathered to a single processing unit. Apart from these communication operations all processing units can run independently, in a fully data parallel manner.

```
FOR i=0:NrImages-1 DO
   InputIm = ReadFile(...);
   SqrdInputIm = BinPixOp(InputIm, "mul", InputIm);
   FOR j=0:NrSymbols-1 DO
      IF (i==0) THEN
         weights[j] = ReadFile(...);
         symbols[j] = ReadFile(...);
         symbols[j] = BinPixOp(symbols[j], "mul", weights[j]);
      FI
      FiltIm1 = GenConvOp(SqrdInputIm, "mult", "add", weights[j]);
      FiltIm2 = GenConvOp(InPutIm, "mult", "add", symbols[j]);
      FiltIm2 = BinPixOp(FiltIm2, "mult", 2);
      ErrorIm = UnPixOp(FiltIm1, "sub", FiltIm2);
      WriteFile(ErrorIm);
   OD
OD
```

Figure 4. Pseudo code for template matching.

5.1.3. Performance evaluation

Because template matching is such an important task in image processing, it is essential for our software architecture to perform well for this application. The results obtained for the automatically optimized parallel version of the program, presented in Figure 5(a), show that this is indeed the case. For these results, the graph of Figure 5(b) shows that even for a large number of processing units, speedup is close to linear. As was to be expected, the speedup characteristics are identical when the same number of templates is used in the matching process, irrespective of the number of input images.

It should be noted that the '1 template' case represents a lower bound on the obtainable speedup (which is slightly over 80 for 120 nodes). This is because in this situation the communication versus computation ratio is highest for the presented parallel system sizes. Additional measurements have indicated that the '10 template' case is a representative upper bound (with a speedup of more than 96 for 120 nodes). Even when up to 50 templates are being used in the matching process, the speedup characteristics were found to be almost identical to this upper bound.

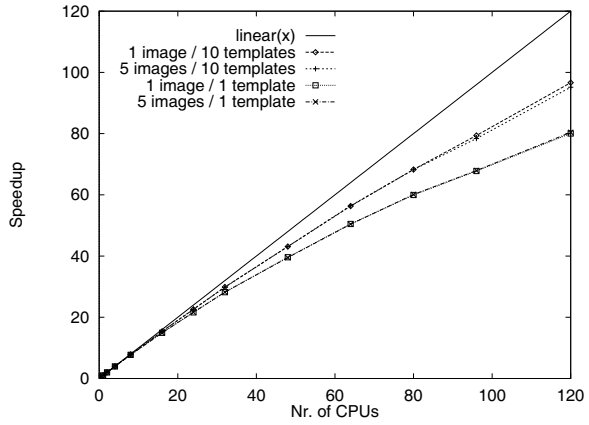
5.2. Multi-baseline stereo vision

As indicated in [56], depth maps obtained by conventional stereo ranging, which uses correspondences between images obtained from two cameras placed at a small distance from each other, are generally not very accurate. Several solutions to this problem have been proposed in the literature, ranging from a hierarchical smoothing or coarse-to-fine strategy, to a global optimization technique based on surface coherence assumptions. These techniques, however, tend to be heuristic or result in computationally expensive algorithms.

In [57], Okutomi and Kanade propose an efficient *multi-baseline stereo vision* method, which is more accurate for depth estimation than more conventional approaches. Whereas, in ordinary stereo, depth

# CPUs	1 input image		5 input images	
	1 template (s)	10 templates (s)	1 template (s)	10 templates (s)
1	25.439	253.165	127.158	1265.425
2	12.774	126.694	63.819	633.083
4	6.449	63.707	32.237	318.559
8	3.287	32.212	16.435	161.303
16	1.703	16.459	8.519	82.259
24	1.176	11.207	5.876	55.838
32	0.902	8.473	4.508	42.414
48	0.642	5.875	3.218	29.367
64	0.503	4.493	2.523	22.409
80	0.424	3.708	2.115	18.546
96	0.375	3.189	1.871	16.146
120	0.317	2.619	1.581	13.299

(a)



(b)

Figure 5. Performance and speedup characteristics for template matching using input images of 1093×649 (4-byte) pixels and templates of size 41 × 35. (a) Execution times in seconds for multiple combinations of templates and images. (b) Speedup graph for all measurements.

(a) (b)

Figure 6. Example of typical input scene (a) and extracted depth map (b). Courtesy of Professor H. Yang, University of Alberta, Canada.

is estimated by calculating the error between two images, multi-baseline stereo requires more than two equally spaced cameras along a single *baseline* to obtain redundant information. In comparison with two-camera methods, multi-baseline stereo was shown to significantly reduce the number of false matches, thus making depth estimation much more robust.

In the algorithm discussed here, input consists of images acquired from three cameras. One image is the *reference* image, the other two are *match* images. For each of 16 disparities, $d = 0, \ldots, 15$, the first match image is shifted by d pixels, the second image is shifted by $2d$ pixels. First, a *difference* image is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error* image is formed by replacing each pixel with the sum of the pixels in a surrounding 13×13 window. The resulting *disparity* image is then formed by finding, for each pixel, the disparity that minimizes the error. The depth of each pixel then can be displayed as a simple function of its disparity. A typical example of a depth map extracted in this manner is given in Figure 6.

5.2.1. Sequential implementations

The sequential implementation used in this evaluation is based on a previous implementation written in a specialized parallel image processing language, called Adapt [58] (see also Section 3.2.1). As shown in Figure 7, for each displacement two disparity images are obtained by first shifting the two match images, and calculating the squared difference with the reference image. Next, the two disparity images are added to form the difference image. Finally, in the example code, the result image is obtained by performing a convolution with a 13×13 uniform filter and minimizing over results obtained previously.

With our software architecture we have implemented two versions of the algorithm that differ only in the manner in which the pixels in the 13×13 window are summed. The pseudo code of Figure 7 shows the version that performs a full two-dimensional convolution, which we refer to as *VisSlow*.


```
ErrorIm = UnPixOp(ErrorIm, "set", MAXVAL);
FOR all displacements d DO
    DisparityIm1 = BinPixOp(MatchIm1, "horshift", d);
    DisparityIm2 = BinPixOp(MatchIm2, "horshift", 2 × d);
    DisparityIm1 = BinPixOp(DisparityIm1, "sub", ReferenceIm);
    DisparityIm2 = BinPixOp(DisparityIm2, "sub", ReferenceIm);
    DisparityIm1 = BinPixOp(DisparityIm1, "pow", 2);
    DisparityIm2 = BinPixOp(DisparityIm2, "pow", 2);
    DifferenceIm = BinPixOp(DisparityIm1, "add", DisparityIm2);
    DifferenceIm = GenConvOp(DifferenceIm, "mult", "add", unitKer);
    ErrorIm = BinPixOp(ErrorIm, "min", DifferenceIm);
OD
```

Figure 7. Pseudo code for multi-baseline stereo vision.

As explained in detail in [59], a faster sequential implementation is obtained when partial sums in the image's y-direction are buffered while sliding the window over the image. We refer to this optimized version of the algorithm as *VisFast*.

5.2.2. *Parallel execution*

The generated optimal schedule for either version of the program of Section 5.2.1 requires not more than five communication steps. In the first loop iteration—and only then—the three input images `MatchIm1`, `MatchIm2`, and `ReferenceIm` are scattered to all processing units. The decompositions of these images are all identical (and performed in a row-wise fashion only) to avoid a domain mismatch and unnecessary communication. Also, in each loop iteration border communication is performed in either version of the program. Again, the extent of the border in each dimension is half the size of the kernel minus one pixel (i.e. six pixels in total). Finally, at the end of the last loop iteration the result image (`ErrorIm`) is gathered to one processor.

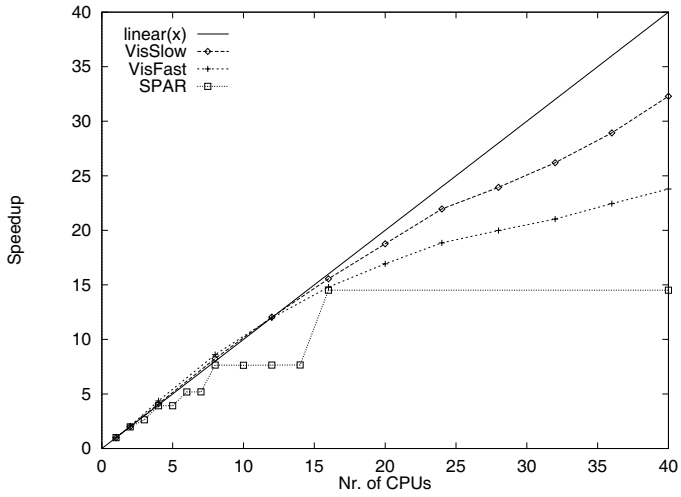
5.2.3. *Performance evaluation*

Results obtained for the two implementations, given input images of size 240×256 pixels (as used most often in the literature) are shown in Figure 8(a). Given the fact that we only allow border exchange among neighboring nodes in a logical CPU grid, the maximum number of nodes that can be used for such image size is 40. In case more CPUs are used, several nodes will have partial image structures with an extent of less than six pixels in one dimension. As the size of the shadow region for a 13×13 kernel is six pixels in both dimensions, nodes would have to obtain data from its neighbor's neighbors as well. The required communication pattern for this behavior generally has a negative impact on performance, and therefore we have not incorporated it in our architecture.

As expected, Figure 8(a) shows that the performance of the *VisFast* version of the algorithm is significantly better than that of *VisSlow*. Also, the graph of Figure 8(b) shows that the speedup

# CPUs	Software architecture VisFast (s)	Software architecture VisSlow (s)	SPAR VisTask (s)
1	1.998	5.554	*8.680*
2	0.969	2.759	*4.372*
4	0.458	1.354	*2.214*
8	0.232	0.674	*1.135*
12	0.167	0.461	*1.135*
16	0.135	0.357	*0.598*
20	0.118	0.296	
24	0.106	0.253	
28	0.100	0.232	
32	0.095	0.212	
36	0.089	0.192	
40	0.084	0.172	

(a)

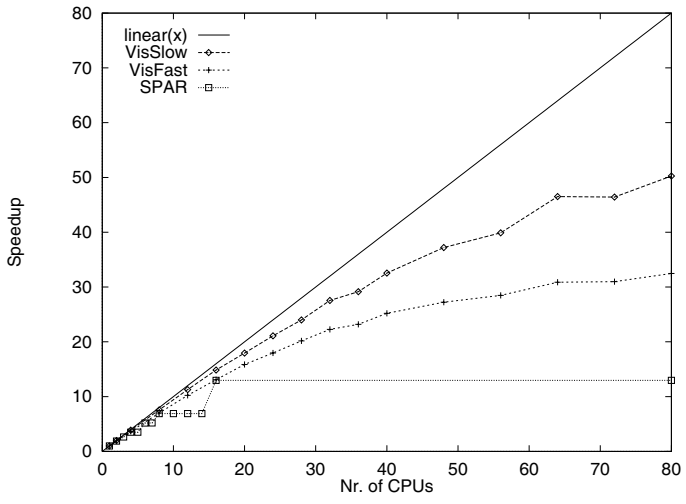


(b)

Figure 8. Performance and speedup characteristics for multi-baseline stereo vision using input images of 240×256 (4-byte) pixels. (a) Execution times in seconds for parallel programs obtained with our architecture for both algorithms. Results in italic obtained for the task parallel implementation in the SPAR parallel programming language. (b) Speedup graph for all measurements.

# CPUs	Software architecture VisFast (s)	Software architecture VisSlow (s)	SPAR VisTask (s)
1	8.770	24.375	*42.993*
2	4.515	12.343	*22.776*
4	2.396	6.300	*12.283*
8	1.250	3.218	*6.219*
16	0.670	1.641	*3.312*
24	0.488	1.156	
32	0.394	0.885	
40	0.348	0.749	
48	0.322	0.655	
56	0.308	0.611	
64	0.284	0.524	
80	0.270	0.485	

(a)



(b)

Figure 9. Performance and speedup characteristics for multi-baseline stereo vision using input images of 512×528 (4-byte) pixels. (a) Execution times in seconds for parallel programs obtained with our architecture for both algorithms. Results in italic obtained for the task parallel implementation in the SPAR parallel programming language. (b) Speedup graph for all measurements.

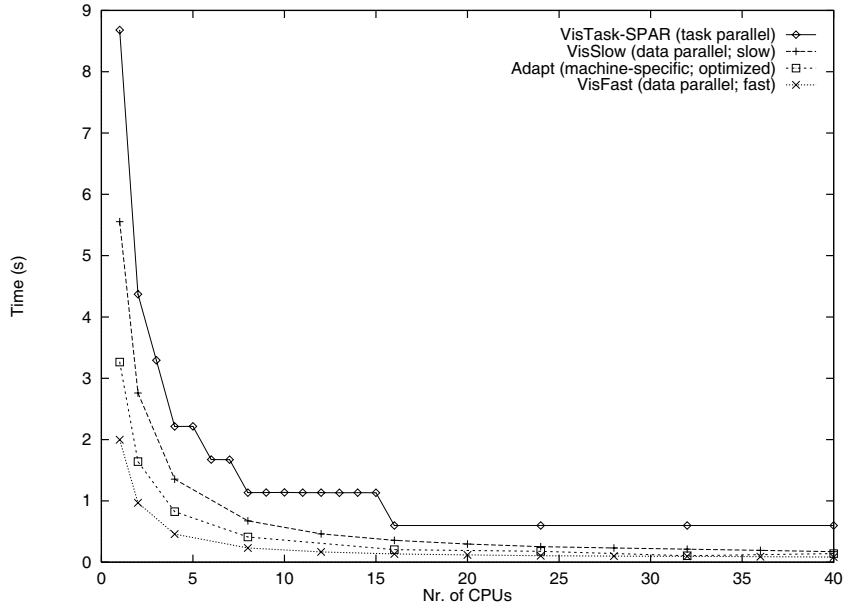


Figure 10. Comparison of execution times for the *VisSlow* and *VisFast* programs implemented with our software architecture, the *VisTask* program implemented using SPAR, and the results obtained for the Adapt implementation reported in [58] (all for 240 × 256 (4-byte) input images).

obtained for both applications is close to linear for up to 24 CPUs. When more than 24 nodes are used, the speedup graphs flatten out due to the relatively short execution times. The schedule for this program generated by our software architecture is to be considered optimal. This can be derived from the fact that our implementations even obtain superlinear speedups for up to 12 processing units. Figure 9 shows similar results obtained for a system of up to 80 nodes, and using input images of size 512 × 528 pixels.

In Figures 8 and 9 we have also made a comparison with results obtained for the same application—implemented in a task parallel manner—written in a specialized parallel programming language (SPAR [60]), and executed on the same parallel machine. In this implementation, referred to as *VisTask*, each loop iteration is designated as an independent task, thus reducing the number of nodes that can be used effectively to 16. The code generated by the SPAR front-end, as well as our own architecture, both were compiled in an identical manner. Although the communication characteristics of the SPAR implementation are significantly different, measurements on a single DAS node indicate that the overhead resulting from our software architecture is much smaller than that of the SPAR runtime system. However, the speedup characteristics obtained for the *VisTask* implementation indicate that SPAR does a good job for this particular application as well.

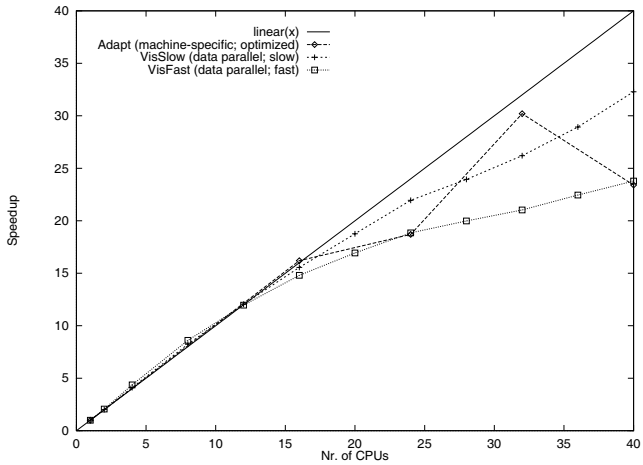


Figure 11. Comparison of speedup for the *VisSlow* and *VisFast* programs implemented with our software architecture, and the results obtained for the Adapt implementation reported in [58] (all for 240 × 256 (4-byte) input images).

Based on the presented results we can conclude that our software architecture behaves well for this application. In this respect, it is interesting to note that our results are comparable to the performance obtained for a *VisFast*-like implementation in the Adapt parallel image processing language reported by Webb [58] (see Figure 10). A true comparison with this work is difficult, however, as the results were obtained on a significantly different machine (i.e. a collection of iWarp processors, with a better potential for obtaining high speedup than the DAS cluster), and for an implementation that was optimized for 2^x nodes. Comparison with the speedup characteristics of the Adapt implementation is even more difficult, as the results in Figure 11 indicate that these fluctuate quite substantially.

More interesting is a comparison with *Easy-PIPE* [61], a library-based software environment for parallel image processing similar to ours. The most distinctive feature of this particular architecture, is that it incorporates a mechanism for combining data and task parallelism. Also, in contrast to our architecture, *Easy-PIPE* does not shield *all* parallelism from the application programmer. As a consequence of these two essential differences, *Easy-PIPE* has the potential of outperforming our architecture, which is fully user transparent and strictly data parallel. However, performance and speedup characteristics for the multi-baseline stereo application obtained on the very same DAS cluster (see Figure 12), indicate that our implementations better exploit the available parallelism than the two presented *Easy-PIPE* versions of the program. Part of the difference is accounted for by the fact that the *Easy-PIPE* architecture does not incorporate an inter-operation optimization mechanism for removal of redundant communication overhead. This, however, cannot fully explain why the speedup graph for the strictly data parallel *Easy-PIPE* version of the program is not comparable to the graphs for both of our data parallel implementations. Apparently, the parallelization overhead of the *Easy-PIPE* implementations is much higher than that of our software architecture.

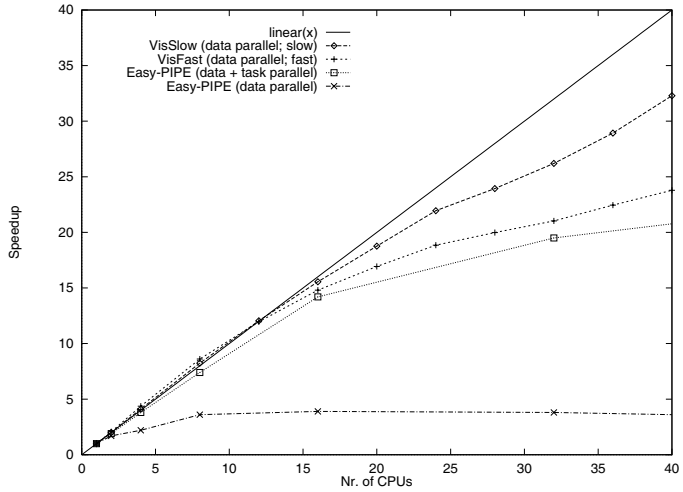


Figure 12. Comparison of speedup for the *VisSlow* and *VisFast* programs implemented with our software architecture, and the results obtained for the two Easy-PIPE implementations reported in [61] (all for 240 × 256 (4-byte) input images).

5.3. Detection of linear structures

As discussed in [62], the important problem of detecting lines and linear structures in images is solved by considering the second-order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_u in the line direction, and differentiation scale σ_v perpendicular to the line, given by

$$r''(x, y, \sigma_u, \sigma_v, \theta) = \sigma_u \sigma_v |f_{vv}^{\sigma_u, \sigma_v, \theta}| \frac{1}{b^{\sigma_u, \sigma_v, \theta}} \tag{3}$$

with b the line brightness. When the filter is correctly aligned with a line in the image, and σ_u, σ_v are optimally tuned to capture the line, filter response is maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection:

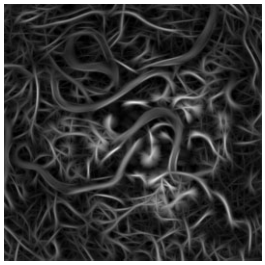
$$R(x, y) = \arg \max_{\sigma_u, \sigma_v, \theta} r''(x, y, \sigma_u, \sigma_v, \theta) \tag{4}$$

Figure 13(a) gives a typical example of an image used as input to this algorithm. Results obtained for a reasonably large subspace of $(\sigma_u, \sigma_v, \theta)$ are shown in Figure 13(b).

5.3.1. *Sequential implementations*

The anisotropic Gaussian filtering problem can be implemented sequentially in many different ways. In the remainder of this section we will consider three possible approaches. First, for each orientation θ

(a)



(b)

Figure 13. Detection of *C. Elegans* worms (courtesy of Janssen Pharmaceuticals, Beerse, Belgium).

it is possible to create a new filter based on σ_u and σ_v. In effect, this yields a rotation of the filters, while the orientation of the input image remains fixed. Hence, a sequential implementation based on this approach (which we refer to as *Conv2D*) implies full two-dimensional convolution for each filter.

The second approach (referred to as *ConvUV*) is to decompose the anisotropic Gaussian filter along the perpendicular axes u, v, and use bilinear interpolation to approximate the image intensity at the filter coordinates. Although comparable to the *Conv2D* approach, *ConvUV* is expected to be faster due to a reduced number of accesses to the image pixels. A third possibility (called *ConvRot*) is to keep the orientation of the filters fixed and to rotate the input image instead. The filtering now proceeds in a two-stage separable Gaussian, applied along the x- and y-direction.

Pseudo code for the *ConvRot* algorithm is given in Figure 14. The program starts by rotating the input image for a given orientation θ. In addition, for all (σ_u, σ_v) combinations the filtering is performed by xy-separable Gaussian filters. For each orientation step the maximum response is combined in a single contrast image structure. Finally, the temporary contrast image is rotated back to match the orientation of the input image, and the maximum response image is obtained.

For the *Conv2D* and *ConvUV* algorithms, the pseudo code is identical and given in Figure 15. Filtering is performed in the inner loop by either a full two-dimensional convolution (*Conv2D*) or by a separable filter in the principle axes directions (*ConvUV*). On a state-of-the-art sequential machine either program may take from a few minutes up to several hours to complete, depending on the size of the input image and the extent of the chosen parameter subspace. Consequently, for the directional filtering problem parallel execution is highly desired.

5.3.2. Parallel execution

Automatic optimization of the *ConvRot* program has resulted in an optimal schedule, as described in more detail in [51]. In this schedule, the full `OriginalIm` structure is broadcast to all nodes before each calculates its respective partial `RotatedIm` structure. This broadcast needs to be performed only once, as `OriginalIm` is not updated in any operation. Subsequently, all operations in the innermost

```
FOR all orientations θ DO
    RotatedIm = GeometricOp(OriginalIm, "rotate", θ);
    ContrastIm = UnPixOp(ContrastIm, "set", 0);
    FOR all smoothing scales σ_u DO
        FOR all differentiation scales σ_v DO
            FiltIm1 = GenConvOp(RotatedIm, "gaussXY", σ_u, σ_v , 2, 0);
            FiltIm2 = GenConvOp(RotatedIm, "gaussXY", σ_u, σ_v , 0, 0);
            DetectedIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
            DetectedIm = BinPixOp(DetectedIm, "mul", σ_u × σ_v);
            ContrastIm = BinPixOp(ContrastIm, "max", DetectedIm);
        OD
    OD
    BackRotatedIm = GeometricOp(ContrastIm, "rotate", −θ);
    ResultIm = BinPixOp(ResultIm, "max", BackRotatedIm);
OD
```

Figure 14. Pseudo code for the *ConvRot* algorithm.

```
FOR all orientations θ DO
    FOR all smoothing scales σ_u DO
        FOR all differentiation scales σ_v DO
            FiltIm1 = GenConvOp(OriginalIm, "func", σ_u, σ_v , 2, 0);
            FiltIm2 = GenConvOp(OriginalIm, "func", σ_u, σ_v , 0, 0);
            ContrastIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
            ContrastIm = BinPixOp(ContrastIm, "mul", σ_u × σ_v);
            ResultIm = BinPixOp(ResultIm, "max", ContrastIm);
```

Figure 15. Pseudo code for the *Conv2D* and *ConvUV* algorithms, with `"func"` either `"gauss2D"` or `"gaussUV"`.

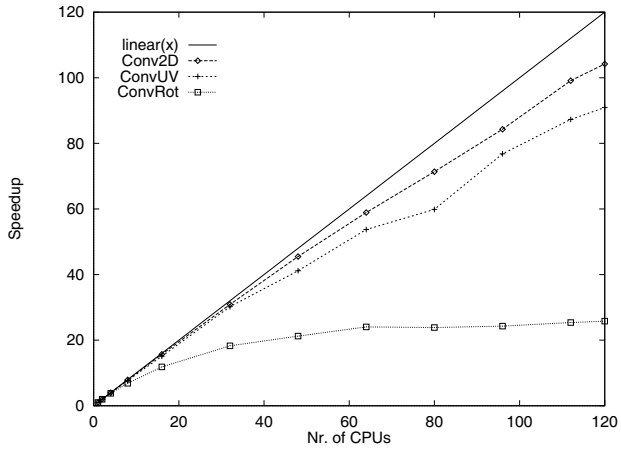
loop are executed locally on partial image data structures. The only need for communication is in the exchange of image borders in the two Gaussian convolution operations.

The two final operations in the outermost loop are executed in a data parallel manner as well. As this requires the distributed image `ContrastIm` to be available in full at each node, a gather-to-all operation is performed. Finally, a partial maximum response image `ResultIm` is calculated on each node, which requires a final gather operation just before termination of the program.

The schedule generated for either the *Conv2D* program or the *ConvUV* program is straightforward and similar to that of the application of Section 5.1. First, the `OriginalIm` structure is scattered such that each node obtains an equal-sized non-overlapping slice of the image's domain. Next, all operations are performed in parallel, with a border exchange required in the convolution operations. Finally, before termination of the program `ResultIm` is gathered to a single node.

# CPUs	ConvRot (s)	Conv2D (s)	ConvUV (s)
1	666.720	2085.985	437.641
2	337.877	1046.115	220.532
4	176.194	525.856	113.526
8	97.162	264.051	56.774
16	56.320	132.872	28.966
32	36.497	67.524	14.494
48	31.399	45.849	10.631
64	27.745	35.415	8.147
80	27.950	29.234	7.310
96	27.449	24.741	5.697
112	26.284	21.046	5.014
120	25.837	20.017	4.813

(a)

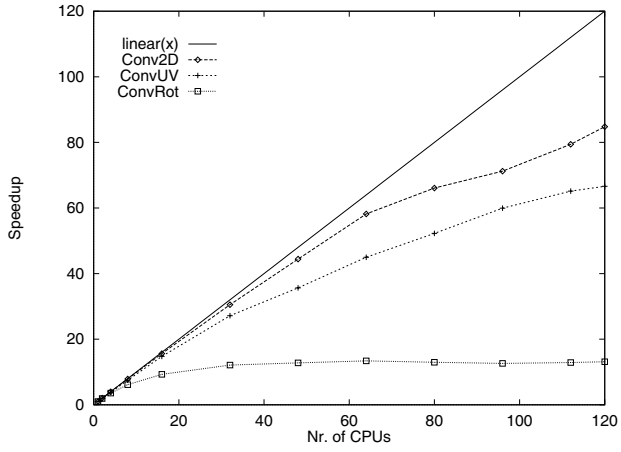


(b)

Figure 16. (a) Performance and (b) speedup characteristics for computing a typical orientation scale-space at 5° angular resolution (i.e. 36 orientations) and eight (σ_u, σ_v) combinations. Scales computed are $\sigma_u \in \{3, 5, 7\}$ and $\sigma_v \in \{1, 2, 3\}$, ignoring the isotropic case $\sigma_{u,v} = \{3, 3\}$. Image size is 512×512 (4-byte) pixels.

# CPUs	ConvRot (s)	Conv2D (s)	ConvUV (s)
1	110.127	325.259	56.229
2	56.993	162.913	28.512
4	30.783	82.092	14.623
8	17.969	41.318	7.510
16	11.874	20.842	3.809
32	9.102	10.660	2.071
48	8.617	7.323	1.578
64	8.222	5.589	1.250
80	8.487	4.922	1.076
96	8.729	4.567	0.938
112	8.551	4.096	0.863
120	8.391	3.836	0.844

(a)



(b)

Figure 17. (a) Performance and (b) speedup characteristics for computing a minimal orientation scale-space at 15° angular resolution (i.e. 12 orientations) and two (σ_u, σ_v) combinations. Scales computed are $\sigma_{u,v} = \{1, 3\}$ and $\sigma_{u,v} = \{3, 7\}$.

5.3.3. Performance evaluation

From the description, it is clear that the *ConvRot* algorithm is most difficult to parallelize efficiently. This is due to the data dependencies present in the algorithm (i.e. the repeated image rotations), and not in any way related to the capabilities of our software architecture. In other words, even when implemented by hand the *ConvRot* algorithm is expected to have speedup characteristics inferior to those of the other two algorithms. Also, *Conv2D* is expected to be the slowest sequential implementation, due to the excessive accessing of image pixels in the two-dimensional convolution operations. In general, *ConvUV* and *ConvRot* will be competing for the best sequential performance, depending on the amount of filtering performed for each orientation.

Figure 16 shows that these expectations are indeed correct. On one processor *ConvUV* is about 1.5 times faster than *ConvRot*, and about 4.8 times faster than *Conv2D*. For 120 nodes these factors have become 5.4 and 4.1, respectively. Because of the relatively poor speedup characteristics, *ConvRot* even becomes slower than *Conv2D* when the number of nodes becomes large. Although *Conv2D* has better speedup characteristics, the *ConvUV* implementation always is fastest, either sequentially or in parallel. Figure 17 presents similar results for a minimal parameter subspace, thus indicating a lower bound on the obtainable speedup.

Speedup values obtained on 120 nodes for a typical parameter subspace (Figure 16) are 104.2 and 90.9 for *Conv2D* and *ConvUV*, respectively. As a result we can conclude that our architecture behaves well for these implementations. In contrast, the usage of generic algorithms (see Section 4.2) has caused the sequential implementation of image rotation to be non-optimal for certain special cases. As an example, rotation over 90° can be implemented much more efficiently than rotation over any arbitrary angle. In our architecture we have decided not to do so, for reasons of software maintainability. As a result, an optimized version of the same algorithm is expected to be faster, but only marginally so.

6. CONCLUSIONS

In this paper, 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-type commodity 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 that indicate significant performance gains for typical image processing applications executing on Beowulf 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 Beowulf-type commodity 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’. Unfortunately, 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 observations we have designed a new library-based software architecture for parallel image processing on homogeneous Beowulf-type commodity clusters. We have presented a list of requirements such a tool must adhere to for it to serve as an acceptable long-term solution for the image processing community at large. 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.

In addition, we have given an assessment of the effectiveness of our software architecture in providing significant performance gains. To this end, we have described the sequential implementation, as well as the automatic parallelization, of three different example applications. The applications are relevant for this evaluation, as all are well-known from the literature, and all contain many fundamental operations required in many other image processing research areas as well.

The results presented in this paper have shown our software architecture to serve well in obtaining highly efficient parallel applications. However, it is important to note that, although all parallelism is hidden inside the architecture itself, part of the efficiency of parallel execution is still in the hands of the application programmer. As we have shown in Section 5.3.3, if a sequential implementation is provided that requires costly communication steps when executed in parallel, program efficiency may be disappointing. Thus, for highest performance the application programmer still should be aware of the fact that usage of such operations is expensive, and should be avoided whenever possible. Any programmer knows that this requirement is not new, however, as a similar requirement holds for sequential execution as well. In other words, this is not a drawback that results from any of the design choices incorporated in our software architecture. The problem cannot be avoided, as it stems directly from the fact that all parallelization and optimization issues are shielded from the application programmer entirely.

In conclusion, although we are aware of the fact that a much more extensive evaluation is required to obtain more insight into the specific strengths and weaknesses of our architecture, the presented results clearly indicate that our architecture constitutes a powerful and user-friendly tool for obtaining high performance in many important image processing research areas. As a result, 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.

REFERENCES

1. Siegel HJ, Armstrong JB, Watson DW. Mapping computer vision-related tasks onto reconfigurable parallel-processing systems. *IEEE Computer* 1992; **25**(2):54–63.
2. Unger SH. A computer oriented towards spatial problems. *Proceedings of the Institute of Radio Engineers* 1958; **46**:1744–1750.
3. van Eijndhoven JTJ *et al*. TriMedia CPU64 architecture. *Proceedings of ICCD'99*, Austin, TX, October 1999; 586–592.
4. Hammerstrom DW, Lulich DP. Image processing using one-dimensional processor arrays. *Proceedings IEEE* 1996; **84**(7):1005–1018.
5. Saoudi A, Nivat M. Optimal parallel algorithms for multidimensional template matching and pattern matching. *Proceedings of ICPIA'92*, Ube, Japan, December 1992; 240–246.
6. Seinstra FJ. User transparent parallel image processing. *PhD Thesis*, Faculty of Science, University of Amsterdam, The Netherlands, May 2003.
7. Andreadis I, Gasteratos A, Tsalides P. An ASIC for fast grey-scale dilation. *Microprocessors and Microsystems* 1996; **20**(2):89–95.

8. Luck L, Chakrabaty C. A digit-serial architecture for gray-scale morphological filtering. *IEEE Transactions on Image Processing* 1995; **4**(3):387–391.
9. Chan SC, Ngai HO, Ho KL. A programmable image processing system using FPGAs. *International Journal of Electronics* 1993; **75**(4):725–730.
10. Draper BA *et al*. Implementing image applications on FPGAs. *Proceedings of the 16th ICPR*, Quebec City, Canada, August 2002; 1381–1384.
11. Eyre J, Bier J. The evolution of DSP processors: From early architecture to the latest developments. *IEEE Signal Processing Magazine* 2000; **17**(2):44–55.
12. Diefendorff K, Dubey PK, Hochsprung R, Scales H. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro* 2000; **20**(2):85–95.
13. Oberman S, Favor G, Weber F. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro* 1999; **19**(2):37–48.
14. Peleg A *et al*. Intel MMX for multimedia PCs. *Communications of the ACM* 1997; **40**(1):24–38.
15. Crookes D. Architectures for high performance image processing: The future. *Journal of Systems Architecture* 1999; **45**:739–748.
16. Bal HE *et al.* The distributed ASCI supercomputer project. *Operating Systems Review* 2000; **34**(4):76–96.
17. Sterling T *et al*. BEOWULF: A parallel workstation for scientific computation. *Proceedings of 24th ICPP*, Oconomowoc, WI, August 1995; 11–14.
18. Jamieson LH, Delp EJ, Wang C-C, Li J, Weil FJ. A software environment for parallel computer vision. *IEEE Computer* 1992; **25**(2):73–75.
19. Juhasz Z. An analytical method for predicting the performance of parallel image processing operations. *The Journal of Supercomputing* 1998; **12**(1/2):157–174.
20. Lee C, Hamdi M. Parallel image processing applications on a network of workstations. *Parallel Computing* 1995; **21**(1):137–160.
21. McBryan OA. An overview of message passing environments. *Parallel Computing* 1994; **20**(4):417–444.
22. Geist A *et al*. PVM: Parallel virtual machine—a users' guide and tutorial for networked parallel computing. *Scientific and Engineering Computation Series*. The MIT Press: Cambridge, MA, 1994.
23. Message Passing Interface Forum. MPI: A message-passing interface standard (version 1.1). *Technical Report*, University of Tennessee, Knoxville, TN, June 1995.
24. Chu-Carroll M, Pollock L. Design and implementation of a general purpose parallel programming system. *Proceedings of HPCN Europe'96*, Brussels, Belgium, April 1996; 499–507.
25. Chandra R *et al. Parallel Programming in OpenMP*. Morgan Kaufmann: San Francisco, CA, 2000.
26. OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface. *Technical Report*. http://www.openmp.org [October 1998].
27. Dent D *et al*. Implementation and performance of OpenMP in ECMWF's IFS code. *Proceedings of Fifth European SGI/Cray MPP Workshop*, Bologna, Italy, September 1999.
28. Goozée RJ, Jacobs PA. Distributed and shared memory parallelism with a smoothed particle hydrodynamics code. *Proceedings of CTAC 2001*, Brisbane, Australia, July 2001.
29. Loveman D. High Performance Fortran. *IEEE Parallel and Distributed Technology* 1993; **1**:25–42.
30. Wilson GV, Lu P. *Parallel Programming Using C++*. The MIT Press: Cambridge, MA, 1996.
31. Chandy KM, Kesselman C. Compositional C++: Compositional parallel programming. *Technical Report TR-92-13*, California Institute of Technology, 1992.
32. Grimshaw AS. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer* 1993; **26**(5):39–51.
33. Bal HE, Kaashoek MF, Tanenbaum AS. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering* 1992; **18**(3):190–205.
34. Blume W *et al.* Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel and Distributed Technology* 1994; **2**(3):37–47.
35. Barnes J. *Programming in Ada 95* (1st edn). Addison-Wesley: Reading, MA, 1995.
36. Jones G, Goldsmith M. *Programming in Occam 2*. Prentice-Hall: Englewood Cliffs, NJ, 1988.
37. Pancake CM, Bergmark D. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer* 1990; **23**(12):13–23.
38. Banerjee U, Eigenmann R, Nicolau A, Padua DA. Automatic program parallelization. *Proceedings IEEE* 1993; **81**(2):211–243.
39. Allen R, Johnson S. Compiling C for vectorization, parallelization, and inline expansion. *Proceedings of PLDI 1988*, Atlanta, GA, June 1988; 241–249.
40. Gabber E, Averbuch A, Yehudai A. Portable, parallelizing Pascal compiler. *IEEE Software* 1993; **10**(2):71–81.
41. Hamey LGC *et al*. An architecture independent programming language for low level vision. *Computer Vision, Graphics and Image Processing* 1989; **48**(2):246–264.

42. Webb JA. Steps toward architecture-independent image processing. *IEEE Computer* 1992; **25**(2):21–31.
43. Crookes D, Morrow PJ, McParland PJ. IAL: A parallel image processing programming language. *IEE Proceedings, Part I* 1990; **137**(3):176–182.
44. Ritter GX, Wilson JN. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press: Boca Raton, FL, 1996.
45. Brown J, Crookes D. A high level language for parallel image processing. *Image and Vision Computing* 1994; **12**(2):67–79.
46. Steele JA. An abstract machine approach to environments for image interpretation on transputers. *PhD Thesis*, Faculty of Science, Queen's University of Belfast, Northern Ireland, May 1994.
47. Taniguchi R *et al*. Software platform for parallel image processing and computer vision. *Proceedings of Parallel and Distributed Methods for Image Processing*, 1997; 2–10.
48. Koelma D *et al*. A software architecture for application driven high performance image processing. *Proceedings of Parallel Distributed Methods for Image Processing*, 1997; 340–351.
49. Morrow PJ *et al*. Efficient implementation of a portable parallel programming model for image processing. *Concurrency: Practice and Experience* 1999; **11**:671–685.
50. Moore MS *et al*. A model-integrated program synthesis environment for parallel/real-time image processing. *Proceedings of Parallel Distributed Methods for Image Processing*, 1997; 31–45.
51. Seinstra FJ, Koelma D, Geusebroek JM. A software architecture for user transparent parallel image processing. *Parallel Computing* 2002; **28**(7–8):967–993.
52. Koelma D *et al*. Horus C++ reference, 1.1. *Technical Report*, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, January 2002.
53. Stroustrup B. *The C++ Programming Language* (3rd edn). Addison-Wesley: Reading, MA, 1997.
54. Seinstra FJ, Koelma D. P-3PC: A point-to-point communication model for automatic and optimal decomposition of regular domain problems. *IEEE Transactions on Parallel and Distributed Systems* 2002; **13**(7):758–768.
55. Bhoedjang RAF, Rühl T, Bal HE. LFC: A communication substrate for Myrinet. *Proceedings of Fourth ASCI Conference*, Lommel, Belgium, June 1998; 31–37.
56. Kanade T. Development of a video-rate stereo machine. *Proceedings of the 1994 DARPA Image Understanding Workshop*, November 1994; 549–558.
57. Okutomi M, Kanade T. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1993; **15**(4):353–363.
58. Webb JA. Implementation and performance of fast parallel multi-baseline stereo vision. *Proceedings of 1993 DARPA Image Understanding Workshop*, April 1993; 1005–1010.
59. Dinda P *et al*. The CMU task parallel program suite. *Technical Report CMU-CS-94-131*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.
60. van Reeuwijk C *et al*. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience* 1997; **9**(11):1193–1205.
61. Nicolescu C, Jonker P. EASY-PIPE—an easy to use parallel image processing environment based on algorithmic skeletons. *Proceedings of 15th IPDPS*, San Francisco, CA, 2001.
62. Geusebroek JM, Smeulders AWM, Geerts H. A minimum cost approach for segmenting networks of lines. *International Journal of Computer Vision* 2001; **43**(2):99–111.

