Efficient Applications in User Transparent Parallel Image Processing

F.J. Seinstra, D. Koelma, J.M. Geusebroek, F.C. Verster and A.W.M. Smeulders
Intelligent Sensory Information Systems,
Faculty of Science, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
(fjseins, koelma, geusebroek, fransve, smeulders)@science.uva.nl

Abstract

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 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 architecture constitutes a powerful and user-friendly tool for obtaining high performance in many important image processing research areas.

1. Introduction

While available for decades, high performance computing architectures have not gained widespread acceptance in the general scientific community. Even in low level image processing — an area that is particularly suitable for the application of parallelism — high performance computing is not applied on a regular basis. Essentially, this is due to the fact that, in comparison with traditional sequential systems, parallel and distributed machines are much harder to program. Although several attempts have been made to alleviate the problem of software design for such systems, no tools have been made available that are truly satisfactory.

The ideal solution is to provide a compiler that can detect all parallelism automatically. Unfortunately, many user-defined imaging algorithms contain data dependencies that prevent efficient parallelization. Also, techniques for automatic dependency analysis and algorithm transformation are still in their infancy. Another solution is to provide a parallel programming language, either general purpose [14] or aimed at image processing specifically [3, 15]. However, such languages still require the programmer to identify the available parallelism, often at a level of detail that is beyond the expertise of most researchers in imaging.

A more practical approach is to design a software library containing parallel versions of operations commonly used in imaging, e.g. as in [6, 13]. Our research is closely related to these projects, as the core of our software architecture is library-based as well. The fundamental difference, however, is that in implementing our architecture we have taken a minimalistic approach to enhance code maintainability. Essentially, we strive to maximize sequential operation reusability and avoid code redundancy as much as possible [12]. An additional distinctive aspect of our work is that we make use of domain specific performance models, e.g. for application optimization across library calls [11].

This paper gives an assessment of the effectiveness of our architecture in providing significant performance gains. To that end, the implementation and automatic parallelization of three example imaging applications is described: (1) template matching, (2) multi-baseline stereo vision, and (3) line detection. Where available, results from the literature are compared with those obtained with our architecture.

This paper is organized as follows. Section 2 shortly introduces our software architecture. In Section 3 a short description is given of the parallel machine used for all evaluation purposes. Next, in each of the three Sections 4, 5, and 6, a different example application is described. Each presents parallelization and optimization details, in combination with obtained performance and speedup characteristics. Concluding remarks are given in Section 7.

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 **IEEE**

2. Software Architecture

The software architecture consists of eight logical components (see Figure 1), each of which is described in short.

C1. Sequential Image Processing Operations. The first component contains a large set of sequential operations typically used in image processing research. As recognized in, for example, Image Algebra [10], a small set of *operation classes* can be identified that covers the bulk of all commonly applied image operations. Each such operation class gives a generic description of a large set of operations with comparable behavior, and is implemented as a *generic algorithm* using the C++ *function template mechanism*. Currently, the following set of generic algorithms 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.

C2. Parallel Extensions. Three classes of routines are implemented (using MPI) that introduce the parallelism into the library: (1) data *partitioning* routines, to map data structures onto a logical grid of processing units of up to 3 dimensions, (2) data *distribution* and *redistribution* routines, and (3) routines for *overlap communication*, to exchange image borders in neighborhood operations.

C3. Parallel Image Processing Operations. To enhance library maintainability, the code for the sequential generic algorithms is reused in the implementation of their respective parallel counterparts. To that end, and as described extensively in [12], for each generic algorithm a so-called *parallelizable pattern* is defined. Each such pattern constitutes the maximum amount of work that can be performed both sequentially and in parallel - in the latter case without having to communicate to obtain non-local data.

C4. Single Uniform API. The image processing library is provided with an application programming interface identical to that of an existing sequential library (Horus [7]). As a result, all parallelism is fully transparent to the user.

C5. Annotated Performance Models. For each generic algorithm only one parallel counterpart is implemented. To ensure efficiency on all target platforms, the parallel algorithms are implemented such that they are capable of adapting to the performance characteristics of a specific machine. To identify these characteristics, each operation is annotated with a performance model, as described extensively in [11].

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

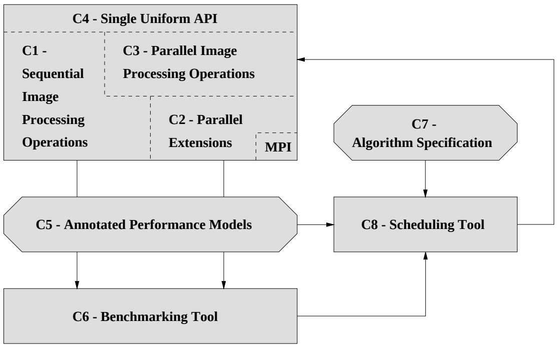

C4 - Single Uniform API
C1 - Sequential Image Processing Operations
C3 - Parallel Image Processing Operations
C2 - Parallel Extensions
MPI
C7 - Algorithm Specification
C5 - Annotated Performance Models
C8 - Scheduling Tool
C6 - Benchmarking Tool

Figure 1. Architecture overview.

C7. Algorithm Specification. Besides intra-operation optimization, optimization across library calls can be performed if information is available on the order in which library operations are applied in a given application. This information is obtainable from the original program code, but currently we assume that a specification is provided in addition to the program itself. Such specification closely resembles a concatenation of library calls, and does not require any parallelism to be introduced by the programmer.

C8. Scheduling Tool. For optimization of a given application a scheduling component is present. It is the task of the scheduler to remove redundant communication steps, and to make optimization decisions regarding: (1) the logical processor grid to map data structures onto, (2) the number of processing units, and (3) the routing pattern for the distribution of data. In the implementation of each parallel generic algorithm, requests for scheduling results are performed to correctly execute the optimizations decided upon. Whether scheduling results are static only (as they are now), or should be generated and updated dynamically is still an important ongoing research issue.

3. Hardware Environment

All three applications described in the remainder of this paper were implemented and tested on the 120-node homogeneous DAS-cluster [1] located at the Vrije Universiteit in Amsterdam. The 200 Mhz Pentium Pro nodes (with 128 MByte of EDO-RAM) are connected by a 1.2 Gbit/sec full-duplex Myrinet network, and run RedHat Linux 6.2. The software architecture was compiled using `gcc` 3.0 (at highest level of optimization) and linked with MPI-LFC [2], 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.

4. Template Matching

Template matching is one of the most fundamental tasks in many imaging 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 occured), 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 (theoretically having a better run-time complexity), this brute force method is fastest for our particular data set.

4.1. Sequential Implementation

Listing 1 is a sequential pseudo code representation of Equation (2). The library calls are as described in Section 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.

```
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
```

Listing 1: Pseudo code for template matching.

4.2. Parallel Execution

As all parallelization issues are shielded from the user, the pseudo code of Listing 1 directly constitutes a program that can be executed in parallel as well. Efficiency of parallel execution depends on the optimizations performed by the scheduling component. For this particular sequential implementation the generated schedule enforces only four different communication steps. First, each input image read from file is scattered throughout the parallel system (note: our architecture does not support parallel I/O). 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 data parallel manner. As such, the program executes in the same way as would have been the case for a hand-coded version.

4.3. 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 presented in Figure 2 show that this is indeed the case (note: in this figure the column title **5/10** (for example) represents results for 5 input images matched against 10 different templates). The graph shows that even for a large number of processing units, speedup is close to linear. Also, it is interesting to see that 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. Additional measurements have indicated that the ’10 template’ case is a representative upper bound. Even when up to 50 templates were used in the matching process, the speedup characteristics were found to be almost identical to this upper bound.

# CPUs	1 / 1	1 / 10	5 / 1	5 / 10
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

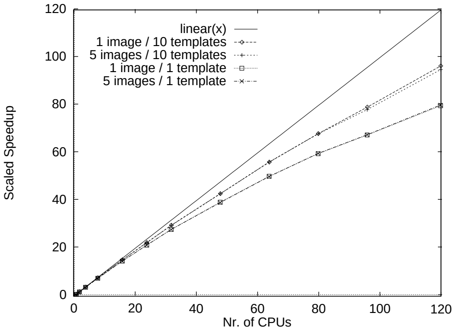


Figure 2. Performance (left) and speedup characteristics (right) for template matching using input images of 1093∗649 (4-byte) pixels and templates of size 41∗35. All times in seconds.

5. Multi-Baseline Stereo Vision

As indicated in [9], multi-baseline stereo vision is a more accurate approach for depth estimation than conventional stereo. Whereas in ordinary stereo depth 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. This approach significantly reduces 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, $d =,0 \cdots ,15$, 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.

```
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
```

Listing 2: Pseudo code for multi-baseline stereo vision.

5.1. Sequential Implementations

Our sequential implementation is based on a previous implementation written in a specialized parallel image processing language, called Adapt [16]. As shown in Listing 2, 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 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 Listing 2 shows the version that performs a full 2-dimensional convolution, which we refer to as ***VisSlow***. As explained in detail in [4], a faster 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 version of the algorithm as ***VisFast***.

5.2. Parallel Execution

The generated optimal schedule for either version of the program of Section 5.1 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 possible 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 about half the size of the kernel (i.e., six pixels in total). Finally, at the end of the last loop iteration the result image (`ErrorIm`) is gathered to one processing unit.

# CPUs	VisFast	VisSlow	SPAR
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	

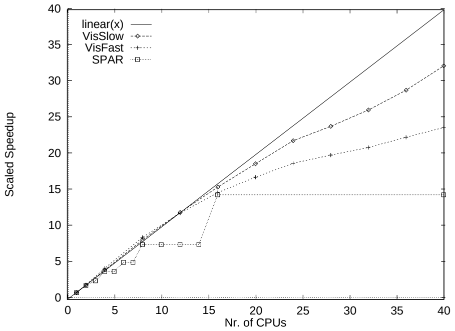


Figure 3. Performance (left) and speedup characteristics (right) for multibaseline stereo vision using input images of 240∗256 (4-byte) pixels. All times in seconds.

# CPUs	VisFast	VisSlow
1	8.770	24.375
2	4.515	12.343
4	2.396	6.300
8	1.250	3.218
16	0.670	1.641
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

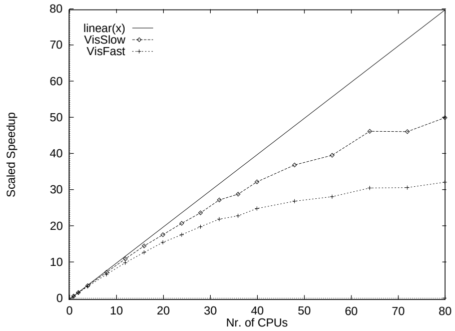


Figure 4. Results for input images of 512∗528 (4-byte) pixels.

5.3. 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 3. 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. As expected, performance of the ***VisFast*** version of the algorithm is significantly better than that of ***VisSlow***. As the schedule generated for this program is identical to what an expert would have implemented by hand, we feel there is no straightforward way of improving these results even further. Figure 4 shows similar results for input images of size $512 * 528$ pixels.

In Figure 3 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 [14]), and executed on the same parallel machine. In this implementation each loop iteration is designated as an independent task, thus reducing the number of processing units that can be used effectively to 16. For this comparison we have made sure that the code generated by the SPAR front-end was compiled identically to our software architecture. Although the communication characteristics of the SPAR implementation are significantly different, the timing results obtained on a single DAS node indicate that the overhead resulting from our software architecture is much smaller than that of the SPAR runtime system. We still feel, however, that SPAR does a pretty good job for this particular application as well.

Based on these results we conclude that our architecture behaves well for this application. For a medium number of nodes (up to 32) speedup is close to linear. When more than 32 nodes are used, the speedup graphs flatten out due to the relatively short execution times. It should be noted that these results are comparable to those reported by Webb [16]. However, a true comparison is difficult, as these results were obtained on a significantly different machine (i.e., 64 iWarp processors), and for an implementation that was optimized for 2^x nodes. Much more interestingly, our results are far better than those reported recently in [8], which were obtained on exactly the same parallel machine, and with a software environment similar to ours. This is all the more remarkable when taking into account the fact that, in contrast with this particular environment, our software architecture shields *all* parallelism from the application programmer.

6. Detection of Linear Structures

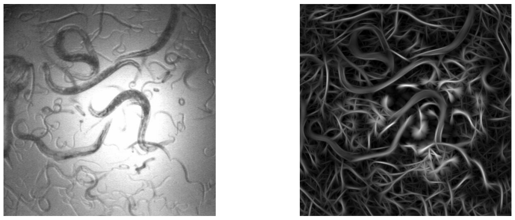
As discussed in [5], the important problem of detecting lines and linear structures in images is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_u in the line direction, and differentiation scale σ_v perpendicular to the line, given by

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

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:

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

Figure 5(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 5(b).



(a) (b)

Figure 5. Detection of *C. Elegans* **worms (courtesy of Janssen Pharmaceuticals, Belgium).**

6.1. 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 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 2-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 less costly 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.

```
FOR all orientations $\theta$ DO
  RotatedIm = GeometricOp(OriginalIm, ”rotate”, $\theta$);
  ContrastIm = UnPixOp(ContrastIm, ”set”, 0);
  FOR all smoothing scales $\sigma_u$ DO
    FOR all differentiation scales $\sigma_v$ DO
      FiltIm1 = GenConvOp(RotatedIm, ”gaussXY”, $\sigma_u$, $\sigma_v$ , 2, 0);
      FiltIm2 = GenConvOp(RotatedIm, ”gaussXY”, $\sigma_u$, $\sigma_v$ , 0, 0);
      DetectedIm = BinPixOp(FiltIm1, ”absdiv”, FiltIm2);
      DetectedIm = BinPixOp(DetectedIm, ”mul”, $\sigma_u * \sigma_v$);
      ContrastIm = BinPixOp(ContrastIm, ”max”, DetectedIm);
    OD
  OD
  BackRotatedIm = GeometricOp(ContrastIm, ”rotate”, $-\theta$);
  ResultIm = BinPixOp(ResultIm, ”max”, BackRotatedIm);
OD
```

Listing 3: Pseudo code for the *ConvRot* algorithm.

Pseudo code for the ***ConvRot*** algorithm is given in Listing 3. The program starts by rotating the original 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 Listing 4. 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.

```
FOR all orientations $\theta$ DO
  FOR all smoothing scales $\sigma_u$ DO
    FOR all differentiation scales $\sigma_v$ DO
      FiltIm1 = GenConvOp(OriginalIm, ”func”, $\sigma_u$, $\sigma_v$ , 2, 0);
      FiltIm2 = GenConvOp(OriginalIm, ”func”, $\sigma_u$, $\sigma_v$ , 0, 0);
      ContrastIm = BinPixOp(FiltIm1, ”absdiv”, FiltIm2);
      ContrastIm = BinPixOp(ContrastIm, ”mul”, $\sigma_u * \sigma_v$);
      ResultIm = BinPixOp(ResultIm, ”max”, ContrastIm);
```

Listing 4: Pseudo code for the *Conv2D* and *ConvUV* algorithms, with `"func"` either `"gauss2D"` or `"gaussUV"`.

6.2. Parallel Execution

Automatic optimization of the ***ConvRot*** program has resulted in a schedule that is optimal for this application, as described in detail in [11]. 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 loop are executed locally on partial image data structures. The only need for communication is in the exchange of image borders (shadow regions) in the two Gaussian convolution operations.

The two final operations in the outermost loop are excuted in a data parallel manner as well. As this requires the distributed image ContrastIm to be available in full at each node [11], 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 to be executed 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 template matching application of Section 4. 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 border exchange communication required in the convolution operations only. Finally, before termination of the program ResultIm is gathered to a single node.

6.3. Performance Evaluation

From the description above it is clear that the ***ConvRot*** algorithm is most difficult to parallelize efficiently. Note that 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 that are not as good as those of the other two algorithms. Furthermore, ***Conv2D*** is expected to be the slowest sequential implementation, due to the excessive accessing of image pixels in the 2-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 6 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 7 presents similar results for a minimal parameter subspace, thus indicating a lower bound on the obtainable speedup.

The generated schedules for both the ***Conv2D*** program and the ***ConvUV*** program are identical to what an expert would have implemented by hand. Speedup values obtained on 120 nodes for a typical parameter subspace (Figure 6) are 104.2 and 90.9 for ***Conv2D*** and ***ConvUV*** respectively. As a result we can conclude that our software architecture behaves well for these implementations. In contrast, the usage of generic algorithms (see Section 2 and also [12]) 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, mainly for reasons of software maintainability [11]. As a result, we expect a hand-coded and hand-optimized version of the same algorithm to be faster, but only marginally so.

7. Conclusions and Future Work

In this paper we have described a software architecture that allows researchers in image processing to develop parallel applications in a fully transparent (i.e., sequential) manner. Based on a description of the sequential implementation and parallel execution of three different example applications we have given an assessment of the effectiveness of this architecture in providing significant performance gains. These example applications are highly relevant because all are well-known from the literature, and all contain fundamental operations required in many other image processing research areas as well.

The results presented in this paper have shown our architecture to serve well in obtaining efficient parallel applications. In almost all situations hand-coded programs would not have produced significantly better results. However, as indicated in Section 6.3, in certain situations we have decided that code maintainability is more important than highest performance, thus resulting in applications that could be executed more efficiently (but often only marginally so). Therefore, based on the presented results, and given the fact that all parallelism is shielded from the application programmer, we conclude that our architecture constitutes a powerful and user-friendly tool for obtaining high performance in many important image processing research areas.

In the near future we will focus our attention on an extension of the set of (sequential) generic algorithms as described in Section 2. Also, we will continue implementing example applications to investigate the implication of parallelization of typical image processing problems, especially in the area of real-time image processing.

# CPUs	ConvRot	Conv2D	ConvUV
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

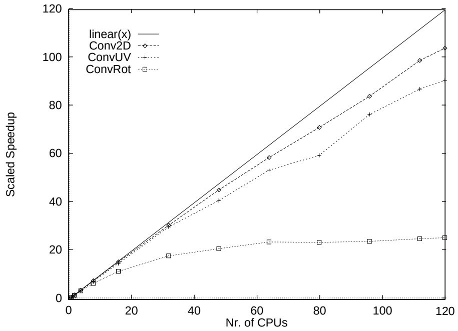


Figure 6. Performance (left) and speedup characteristics (right) for computing a typical orientation scale-space at 5° angular resolution (i.e., 36 orientations) and 8 (σ_u, σ_u) combinations. Scales computed are $\sigma_u \in \{3, 5, 7\}$ and $\sigma_v \in \{1, 2, 3\}$, ignoring the isotropic case $\sigma_{u,v} = \{3, 3\}$. Image size is 512$*$512 (4-byte) pixels. All times in seconds.

# CPUs	ConvRot	Conv2D	ConvUV
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

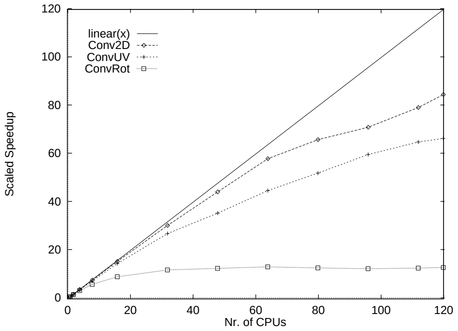


Figure 7. Results for computing a minimal orientation scale-space at 15° angular resolution (i.e., 12 orientations) and 2 (σ_u, σ_u) combinations. Scales computed are $\sigma_{u,v} = \{1, 3\}$ and $\sigma_{u,v} = \{3, 7\}$.

References

[1] H. Bal et al. The Distributed ASCI Supercomputer Project. *Operating Systems Review*, 34(4):76–96, Oct. 2000.

[2] R. Bhoedjang, T. Rühl, and H. Bal. LFC: A Communication Substrate for Myrinet. In *ASCI'98*, pages 31–37, June 1998.

[3] J. Brown et al. A High Level Language for Parallel Image Processing. *Im. Vis. Comp.*, 12(2):67–79, Mar. 1994.

[4] P. Dinda et al. The CMU Task Parallel Program Suite. Technical Report CMU-CS-94-131, 1994.

[5] J. Geusebroek, A. Smeulders, and H. Geerts. A Minimum Cost Approach for Segmenting Networks of Lines. *International Journal of Computer Vision*, 43(2):99–111, July 2001.

[6] L. Jamieson et al. A Software Environment for Parallel Computer Vision. *IEEE Computer*, 25(2):73–75, Feb. 1992.

[7] D. Koelma et al. Horus (Release 0.9.2). Technical report, ISIS, Fac. Science, Univ. Amsterdam, Feb. 2000.

[8] C. Nicolescu et al. EASY-PIPE - An Easy to Use Parallel Image Processing Environment Based on Algorithmic Skeletons. In *PDIVM 2001*, San Francisco, USA, Apr. 2001.

[9] M. Okutomi and T. Kanade. A Multiple-Baseline Stereo. *IEEE Trans. PAMI*, 15(4):353–363, Apr. 19923.

[10] G. Ritter and J. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Inc, 1996.

[11] F. Seinstra et al. A Software Architecture for User Transparent Parallel Image Processing on MIMD Computers. In *EuroPar 2001*, pages 653–662, Manchester, UK, Aug. 2001.

[12] F. Seinstra and D. Koelma. The Lazy Programmer's Approach to Building a Parallel Image Processing Library. In *PDIVM 2001*, San Francisco, USA, Apr. 2001.

[13] R. Taniguchi et al. Software Platform for Parallel Image Processing and Computer Vision. In *Proc. Parallel and Distributed Methods for Image Processing*, pages 2–10, 1997.

[14] K. van Reeuwijk et al. Spar: A Programming Language for Semi-Automatic Compilation of Parallel Programs. *Concurrency: Pract. and Exp.*, 9(11):1193–1205, Nov. 1997.

[15] J. Webb. Steps Toward Architecture-Independent Image Processing. *IEEE Computer*, 25(2):21–31, Feb. 1992.

[16] J. Webb. Implementation and Performance of Fast Parallel Multi-Baseline Stereo Vision. In *Proc. 1993 DARPA Image Understanding Workshop*, pages 1005–1010, Apr. 1993.