Chapter 7

Efficient Applications in User Transparent Parallel Image Processing*

"Thy will by my performance shall be serv'd:
So make the choice of thy own time, for I,
Thy resolv'd patient, on thee still rely."

William Shakespeare - *All's Well That Ends Well* (1623)

In the previous chapters we have described the essential and most innovative aspects of our software architecture for user transparent parallel image processing. First, in Chapter 2 we have discussed the need for the availability of such architecture, and we have presented a bird's eye view of all of the architecture's constituent components. In Chapter 3 we have presented some of the implementation details of the architecture's core — which is a sustainable software library consisting of an extensive set of operations commonly applied in state-of-the-art image processing research. In Chapter 4 we have introduced a performance model, derived from a high level abstract parallel image processing machine definition, which is used for obtaining accurate run time cost estimations for all operations available in our architecture. In addition, in Chapter 5 we have presented an extended model for accurate prediction of the cost of the basic point-to-point communication operations applied in the library implementations. As discussed in Chapter 6, performance estimations obtained from these models are essential for generating the fastest possible parallel version of any

*This chapter is based on our paper as appeared in *Proceedings of the 16th International Parallel & Distributed Processing Symposium (IPDPS 2002)* [150]. An extended version of this chapter is to appear in *Concurrency and Computation: Practice and Experience* [146].

sequential program implemented using our software architecture. In relation to this, in Chapter 6 we have also presented a finite state machine specification, which is used for the automatic conversion of a legal sequential image processing application into a legal, correct, and time-optimal parallel version of the same program.

For each of the research issues presented in the previous chapters, we have discussed the advantages and drawbacks of the solutions incorporated in our software architecture. Where possible, we have also presented results for each of the solutions applied in isolation. To validate *all* of the results of this research, however, the single remaining issue that has yet to be discussed in this thesis is the overall efficiency obtained in case the architecture components are applied in combination.

To this end, in this chapter 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. For all three applications we determine whether the performance obtained with the parallel versions generated by our software architecture indeed adheres to requirement I.2 put forward in Section 2.3 which states that the obtained efficiency generally should compare well to that of reasonable hand-coded parallel implementations.

This chapter is organized as follows. First, in Section 7.1 we give a short description of the hardware architecture that we have used for all evaluation purposes. Next, in each of the Sections 7.2, 7.3, and 7.4, one of the example applications is described and evaluated in extensive detail. Information regarding the parallelization and optimization issues of each application is presented, in combination with obtained performance results and speedup characteristics. Where available, measurement data presented in the literature are compared with performance results obtained with our software architecture. Finally, concluding remarks are given in Section 7.5.

7.1 Hardware Environment

All of the applications described in this chapter have been implemented and tested on the 128-node homogeneous Distributed ASCI Supercomputer (DAS) cluster located at the Vrije Universiteit in Amsterdam [7]. This is a typical example of a machine from the class of homogeneous commodity clusters as described in Section 2.1. All nodes in the cluster contain a 200 Mhz Pentium Pro with 128 MByte 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 [16], an implementation of MPI which is partially optimized for the DAS. The required set of benchmarking operations (see Section 4.4) 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.

7.2 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:

$$\varepsilon(i,j) = \Sigma_m \Sigma_n ((f(i+m, j+n) - g(m,n))^2 \cdot w(m,n)). \tag{7.1}$$

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

$$\varepsilon = f^2 \otimes w - 2 \cdot (f \otimes w \cdot g). \tag{7.2}$$

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.

7.2.1 Sequential Implementation

Listing 7.1 is a sequential pseudo code representation of Equation (7.2). The library calls are as described in Chapter 3. 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.

7.2.2 Parallel Execution

As all parallelization issues are shielded from the user, the pseudo code of Listing 7.1 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 (as described in Chapter 6) has generated a schedule that requires only

```
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 7.1: *Pseudo code for template matching.*

four different communication steps to be executed. First, each input image read from file is scattered throughout the parallel system (generally applying a logical CPU grid of $2 \times (P \div 2)$ or $4 \times (P \div 4)$, depending on the available number of nodes P). 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. As such, the program executes in exactly the same way as would have been the case for a hand-coded parallel version.

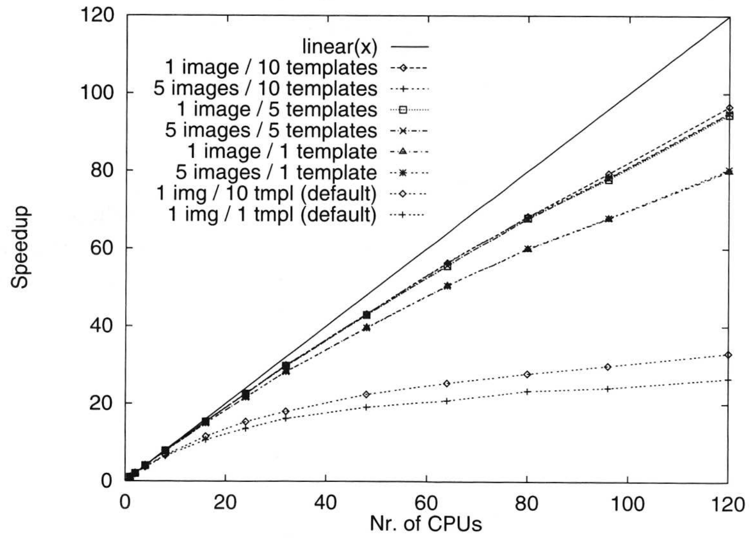
7.2.3 Performance Evaluation

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 the first six columns of Figure 7.1(a), show that this is indeed the case. For these results, the graph of Figure 7.1(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.

# CPUs	time- optimized parallel program						default parallel program	
	1 input image			5 input images			1 input image	
	1 template (s)	5 templates (s)	10 templates (s)	1 template (s)	5 templates (s)	10 templates (s)	1 template (s)	10 templates (s)
1	25.439	126.654	253.165	127.158	632.485	1265.425	25.526	253.627
2	12.774	63.410	126.694	63.819	316.921	633.083	13.466	133.443
4	6.449	31.895	63.707	32.237	159.497	318.559	7.126	69.924
8	3.287	16.138	32.212	16.435	80.655	161.303	3.972	37.975
16	1.703	8.254	16.459	8.519	41.263	82.259	2.399	21.960
24	1.176	5.618	11.207	5.876	28.078	55.838	1.876	16.539
32	0.902	4.261	8.473	4.508	21.318	42.414	1.581	14.128
48	0.642	2.956	5.875	3.218	14.751	29.367	1.337	11.330
64	0.503	2.280	4.493	2.523	11.353	22.409	1.224	9.998
80	0.424	1.865	3.708	2.115	9.340	18.546	1.093	9.119
96	0.375	1.627	3.189	1.871	8.088	16.146	1.056	8.493
120	0.317	1.340	2.619	1.581	6.659	13.299	0.960	7.668

(a)



(b)

Figure 7.1: *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. Results in first six columns obtained for optimized parallel version. Results in last two columns (gray) obtained for non-optimized parallel version generated by default algorithm expansion. (b) Speedup graph for all measurements. Four uppermost lines for optimized program calculating matches for 5 and 10 templates; two lower lines for matching with a single template. Bottom two lines for non-optimized (default) parallel program.*

The additional values in the gray columns of Figure 7.1(a) represent measurement results obtained for a non-optimized parallel version of the program (i.e., the parallel program which is obtained in the process of default algorithm expansion, without applying a redundancy avoidance strategy or any other optimization steps, see Section 6.1.2). These measurements, as well as the related speedup characteristics shown in Figure 7.1(b), clearly indicate the importance of the optimization process presented in Chapter 6. Most importantly, the dramatic results are due to the fact that the default parallel program executes many redundant communication steps. For evaluation of the efficiency of our software architecture, these non-optimized results simply should be ignored. In the remainder of this chapter we will therefore only present results for time-optimized parallel programs.

7.3 Multi-Baseline Stereo Vision

As indicated in [82, 110], 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. In part, this is due to the fundamental difficulty of the stereo correspondence problem: finding corresponding points between left and right images is locally ambiguous. 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 [117], 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 is estimated by calculating the error between two images, multi-baseline stereo requires more than two equally spaced

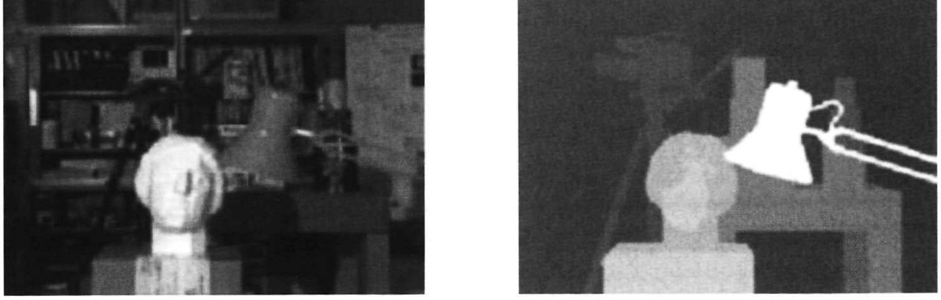

(a) (b)

Figure 7.2: *Example of typical input scene (a) and extracted depth map (b). Courtesy of Professor H. Yang, University of Alberta, Canada.*

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, $d = 0, \cdots, 15$, 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. A typical example of a depth map extracted in this manner is given in Figure 7.2.

7.3.1 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 [166] (see also Section 2.2.2). As shown in Listing 7.2, 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 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 Listing 7.2 shows the version that performs a full 2-dimensional convolution, which we refer to as ***VisSlow***. As explained in detail in [43], 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***.

```
ErrorIm = UnPixOp(ErrorIm, "set", MAXVAL);
FOR all displacements $d$ DO
    DisparityIm1 = BinPixOp(MatchIm1, "horshift", $d$);
    DisparityIm2 = BinPixOp(MatchIm2, "horshift", $2 \times d$);
    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 7.2: *Pseudo code for multi-baseline stereo vision.*

7.3.2 Parallel Execution

The generated optimal schedule for either version of the program of Section 7.3.1 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 — i.e., using a $1 \times P$ logical CPU grid mapping) 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. As in the example of Section 7.2, the optimized parallel programs obtained with our software architecture execute in exactly the same way as would have been the case for reasonable hand-coded implementations.

7.3.3 Performance Evaluation

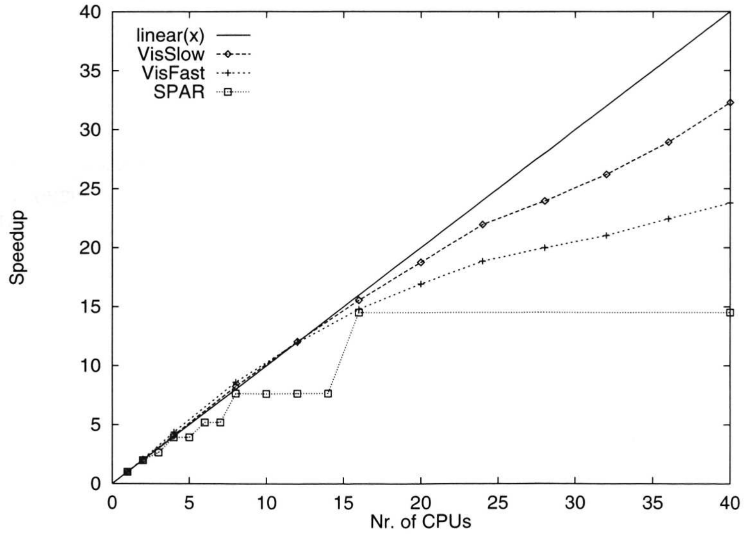
Results obtained for the two implementations, given input images of size 240×256 pixels (as used most often in the literature) are shown in Figure 7.3(a). Given the fact that we only allow border exchange among neighboring nodes in a logical CPU grid, the maximum number of nodes that can be used for such image size is 40. In case more CPUs are being used, several nodes will have partial image structures with an extent of less than 6 pixels in one dimension (due to the one-dimensional partitioning of the input images). As the size of the shadow region for a 13×13 kernel is 6 pixels in both dimensions, nodes would have to obtain data from its neighbor's neighbors as well — or even further away. The communication pattern for this behavior is costly (i.e., the communication versus computation ratio is high), and therefore we have not incorporated it in our architecture.

As expected, Figure 7.3(a) shows that the performance of the ***VisFast*** version of the algorithm is significantly better than that of ***VisSlow***. Also, the graph of Figure 7.3(b) shows that the speedup obtained for both applications is close to linear up to 24 CPUs. When more than 24 nodes are being used, the speedup graphs flatten out due to the relatively short execution times. Because the generated schedule for this program is identical to what an expert programmer would have implemented by hand, this is to be considered optimal. This also can be derived from the fact that superlinear speedups are obtained for up to 12 processing units. Figure 7.4 shows similar speedup characteristics obtained for a system of up to 80 nodes, and using input images of size 512×528 pixels. For up to 40 nodes these results are almost identical to Figure 7.3, indicating a similar impact of communication on overall performance.

In Figures 7.3 and 7.4 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 [129]), and executed on the same parallel machine. In this implementation, referred to as ***VisTask***, each iteration is designated as an independent task, thus exploiting 16 processing units at maximum. For this comparison, the code generated by the SPAR front-end was compiled in a iden-

# CPUs	Software Architecture		SPAR
	VisFast (s)	VisSlow (s)	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)

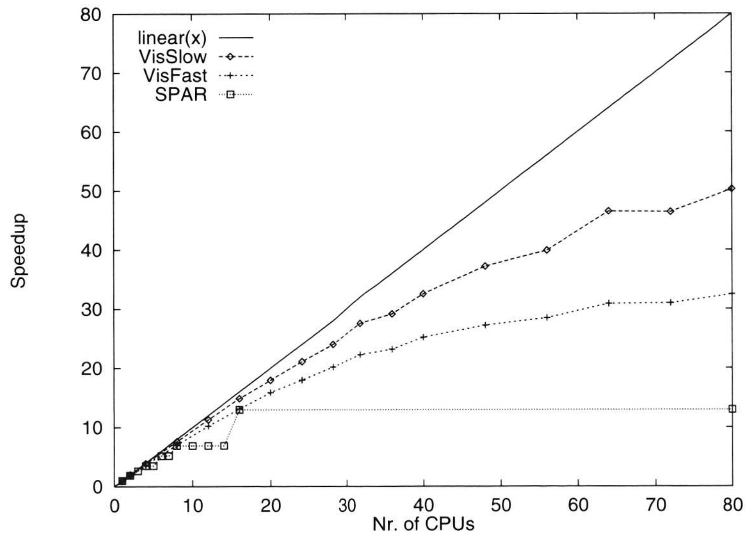


(b)

Figure 7.3: *Performance and speedup characteristics for multi-baseline stereo vision using input images of 240×256 (4-byte) pixels. (a) Execution times in seconds for the optimized parallel programs obtained with our architecture for both algorithms. Results in gray obtained for the task parallel implementation in the SPAR parallel programming language. (b) Speedup graph for all measurements.*

# CPUs	Software Architecture		SPAR
	VisFast (s)	VisSlow (s)	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 7.4: *Performance and speedup characteristics for multi-baseline stereo vision using input images of 512×528 (4-byte) pixels. (a) Execution times in seconds for the optimized parallel programs obtained with our architecture for both algorithms. Results in gray obtained for the task parallel implementation in the SPAR parallel programming language. (b) Speedup graph for all measurements.*

tical manner to the previous case. Although the communication characteristics of the SPAR implementation are significantly different, measurements on a single node indicate that the overhead by our software architecture is much smaller than that of the SPAR runtime system. Nevertheless, the speedup obtained for the ***VisTask*** implementation indicates that SPAR successfully exploits all available parallelism for this particular application. From this comparison we conclude that our software architecture provides fast sequential code, as well as high parallelization efficiency.

Interestingly, our results are comparable to the performance obtained for a ***VisFast***-like implementation in the Adapt parallel image processing language reported by Webb [166] (see Figure 7.5). A comparison is difficult, however, as 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 optimized for 2^x nodes. Comparison with the speedup characteristics of the Adapt implementation is even more difficult, as the results in Figure 7.6 indicate that they fluctuate substantially. Yet, our results on the DAS (which was installed less than 5 years later) make a strong case for our general purpose approach.

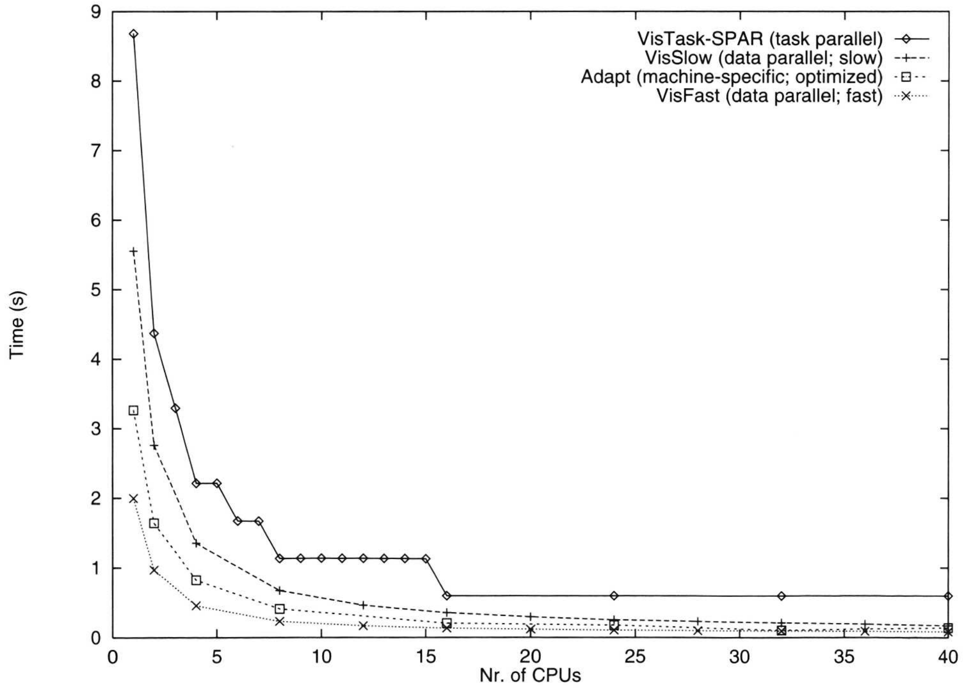


Figure 7.5: *Comparison of execution times for the* ***VisSlow*** *and* ***VisFast*** *programs implemented with our software architecture, the* ***VisTask*** *program implemented using the SPAR parallel language, and the results obtained for the Adapt implementation reported in [166] (all for 240×256 (4-byte) input images).*

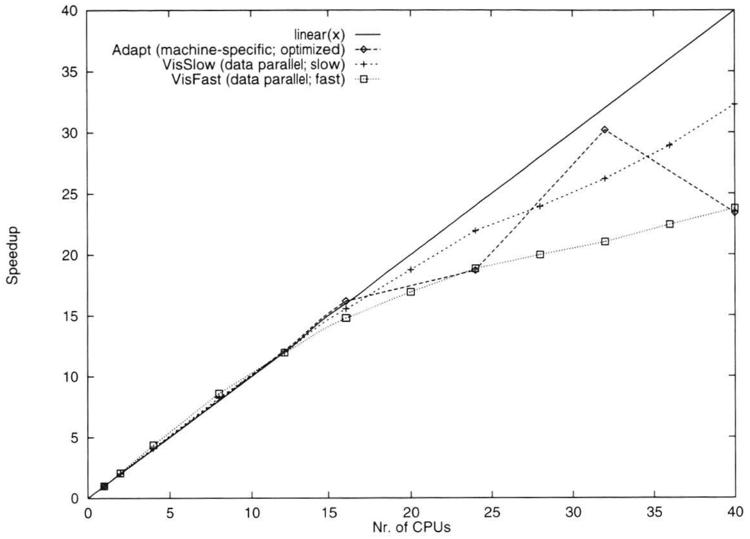


Figure 7.6: *Comparison of speedup for the* ***VisSlow*** *and* ***VisFast*** *programs implemented with our software architecture, and the Adapt implementation reported in [166] (all for 240×256 (4-byte) input images).*

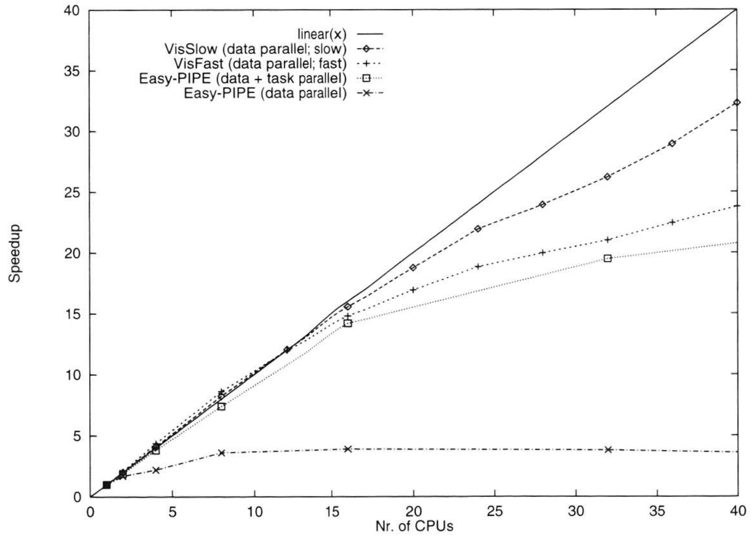


Figure 7.7: *Comparison of speedup for the* ***VisSlow*** *and* ***VisFast*** *programs implemented with our software architecture, and the two Easy-PIPE implementations reported in [111] (all for 240×256 (4-byte) input images).*

More relevant is a comparison with *Easy-PIPE* [111, 112], a library-based software environment for parallel image processing similar to ours. *Easy-PIPE* mainly differs from our architecture in that it incorporates a mechanism for combining data and task parallelism. Also, *Easy-PIPE* does not shield *all* parallelism from the application programmer. As a consequence, *Easy-PIPE* has the potential of outperforming our architecture, which is fully user transparent, and strictly data parallel. Results for the multi-baseline stereo application obtained on the same DAS cluster (see Figure 7.7) indicate that our architecture performs better nonetheless. Part of the difference is accounted to the fact that the two *Easy-PIPE* implementations do not fully exploit all parallelism available in the program. Also, in contrast to our library implementations, the communication routines applied in *Easy-PIPE* rely on the costly creation of separate send and receive buffers in user-space. The bulk of the difference, however, is due to the absence in the *Easy-PIPE* architecture of an inter-operation optimization mechanism for removal of redundant communication overhead, such as our lazy parallelization approach of Chapter 6. As a result, the parallelization overhead of the *Easy-PIPE* implementations is much higher than that of our software architecture.

7.4 Detection of Linear Structures

As discussed in [55], 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

$$r''(x, y, \sigma_u, \sigma_v, \theta) = \sigma_u \sigma_v \left| f_{vv}^{\sigma_u, \sigma_v, \theta} \right| \frac{1}{b^{\sigma_u, \sigma_v, \theta}}, \tag{7.3}$$

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:

$$R(x, y) = \arg \max_{\sigma_u, \sigma_v, \theta} r''(x, y, \sigma_u, \sigma_v, \theta). \tag{7.4}$$

Figure 7.8(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 7.8(b).

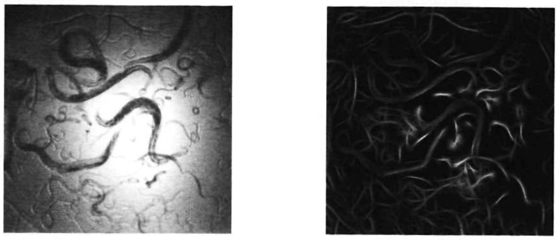

(a) (b)

Figure 7.8: *Detection of* C. Elegans *worms (Janssen Pharmaceuticals, Belgium).*

```
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", $\sigma_u \times \sigma_v$);
      ContrastIm = BinPixOp(ContrastIm, "max", DetectedIm);
    OD
  OD
  BackRotatedIm = GeometricOp(ContrastIm, "rotate", $-\theta$);
  ResultIm = BinPixOp(ResultIm, "max", BackRotatedIm);
OD
```

Listing 7.3: *Pseudo code for the **ConvRot** algorithm.*

7.4.1 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 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 Listing 7.3. 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 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", $\sigma_u \times \sigma_v$);
      ResultIm = BinPixOp(ResultIm, "max", ContrastIm);
```

Listing 7.4: *Pseudo code for the **Conv2D** and **ConvUV** algorithms, with* `"func"` *either* `"gauss2D"` *or* `"gaussUV"`.

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 7.4. 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.

7.4.2 Parallel Execution

Automatic optimization of the ***ConvRot*** program has resulted in an optimal schedule, as described in more detail Section 4.5.2. 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 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 (see Section 4.5.2), 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.

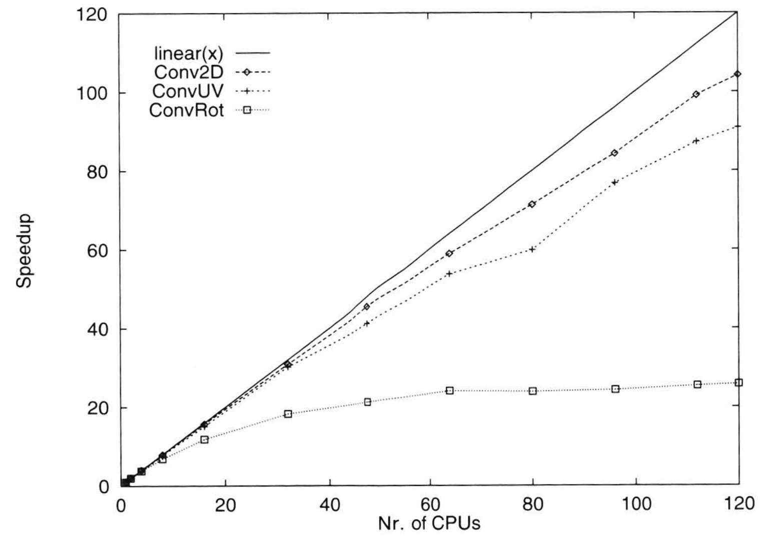
The schedule generated for either the ***Conv2D*** program or the ***ConvUV*** program is straightforward, and similar to that of the template matching application of Section 7.2. First, the `OriginalIm` structure is scattered such that each node obtains an equal-sized non-overlapping slice of the image's 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.

7.4.3 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.

# 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)



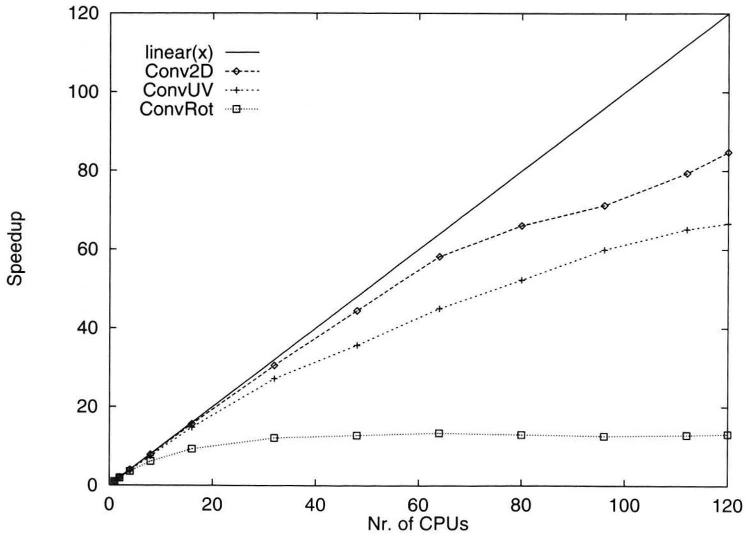
(b)

Figure 7.9: *(a) Performance and (b) speedup characteristics for computing a typical orientation scale-space at* 5° *angular resolution (i.e., 36 orientations) and 8* (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.*

Figure 7.9 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 7.10 presents similar results for a minimal parameter subspace, thus indicating a lower bound on the obtainable speedup.

# 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 7.10: *(a) Performance and (b) speedup characteristics for computing a minimal orientation scale-space at* $15°$ *angular resolution (i.e., 12 orientations) and 2* (σ_u, σ_v) *combinations. Scales computed are* $\sigma_{u,v} = \{1, 3\}$ *and* $\sigma_{u,v} = \{3, 7\}$.

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 7.9) 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 algorithmic patterns (see Chapter 3) 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 (see Chapter 2). As a result, we expect a hand-coded and hand-optimized version of the same algorithm to be faster, but only marginally so.

7.5 Conclusions and Future Work

In this chapter 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 chapter have shown our software architecture to serve well in obtaining highly efficient parallel applications. Moreover, in almost all situations handcrafted code would not have produced significantly better results. As such, we have shown that our architecture adheres to requirement I.2 put forward in Section 2.3 -- which states that the obtained efficiency generally should compare well to that of reasonable hand-coded parallel implementations. As indicated in Section 7.4.3, however, for certain specific operations we have decided that code maintainability is more important than highest performance. Consequently, in comparison with optimal handcrafted parallel code, any application that makes extensive use of such operations may suffer from reduced efficiency (but often only marginally so).

As an important note we should state that, although all parallelism is hidden inside the architecture itself, much of the efficiency of parallel execution is still in the hands of the application programmer. As we have shown in Section 7.4.3, 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 can not 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 in 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. For future evaluation, 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.