Performing real-time image processing on distributed computer systems

RADU DOBRESCU, MATEI DOBRESCU, DAN POPESCU
"Politehnica" University of Bucharest, Faculty of Control and Computers,
313 Splaiul Independentei, Bucharest
ROMANIA
radud@isis.pub.ro

Abstract. The aim of the paper is to validate a software architecture that allows an image processing researcher to develop parallel applications. The challenge was to develop algorithms that perform real-time low level operations on digital images able to be executed on a cluster of desktop PCs. The experiments show how to use parallelizable patterns and how to optimize the load balancing between the workstations.
.

Keywords: real-time image processing, low level operations, parallel and distributed processing, tasks scheduling, lines detection, directional filtering.

1 Introduction

Considering the need for real-time image processing and how this need can be met by exploiting the inherent parallelism in an algorithm, it becomes important to discuss what exactly is meant by the term “real-time,” an elusive term that is often used to describe a wide variety of image processing systems and algorithms. From the literature, it can be derived that there are three main interpretations of the concept of “real-time”, namely real-time in the perceptual sense, real-time in the software engineering sense, and real-time in the signal processing sense.

Real-time in the perceptual sense is used mainly to describe the interaction between a human and a computer device for a near instantaneous response of the device to an input by a human user. For instance, Bovik defines the concept of “real-time” in the context of video processing, describing that “*the result of processing appears effectively* ‘*instantaneously*’ *(usually in a perceptual sense) once the input becomes available*”[1]. Note that “real-time” imposes a maximum tolerable delay based on human perception of delay, which is essentially some sort of application-dependent bounded response time.

Real-time in the software engineering sense is also based on the concept of a bounded response time as in the perceptual sense. Dougherty and Laplante [2] point out that a “*real-time system is one that must satisfy explicit bounded response time constraints to avoid failure*”. So, soft real-time refers to the case where missed real-time deadlines result in performance degradation rather than failure.

Real-time in the signal processing sense is based on the idea of completing processing in the time available between successive input samples [3]. An important item of note here is that one way to gauge the “real-time” status of an algorithm is to determine some measure of the amount of time it takes for the algorithm to complete all requisite transferring and processing of image data, and then making sure that it is less than the allotted time for processing.

In the following the discussion is focused on the possibility to perform software implementation on a parallel processing platform of some primary image processing algorithms, corresponding to real-time in the software engineering sense.

2 Software operations involved in real time image processing

2.1 Levels of image processing operations

The digital primary processing mainly consists of three stages: noise rejection, binary representation, and edge extraction. Due to the fact that the noise can introduce errors in other stages (like contour detection and feature extraction), the image noise rejection must be the first stage in any digital image processing application. For these algorithms it is recommend local operators which act in symmetrical neighborhoods of the considered pixels. They have the advantage of simplicity and they can be implemented easily implemented on dedicated hardware structures. This approach changes when considering software processing. Digital images are essentially multidimensional signals and are thus quite data intensive, requiring a significant amount of computation and memory resources for their processing. The key to cope with this issue is the concept of parallel processing who deals with computations on large data sets. In fact, much of what goes into implementing an efficient image/video processing system centers on how well the implementation, both hardware and software, exploits different forms of parallelism in an algorithm, which can be data level parallelism - DLP or/and instruction level parallelism – ILP [4]. DLP manifests itself in the application of the same operation on different sets of data, while ILP manifests itself in scheduling the simultaneous execution of multiple independent operations in a pipeline fashion.

Traditionally, image processing operations have been classified into three main levels, namely low, intermediate, and high, where each successive level differs in its input/output data relationship [5]. Low-level operators take an image as their input and produce an image as their output, while intermediate-level operators take an image as their input and generate image attributes as their output, and finally high-level operators take image attributes as their inputs and interpret the attributes, usually producing some kind of knowledge-based control at their output.

One can hope that with an adequate task scheduling and a well designed cluster of processors one can perform in real time low-level operations by software parallelization.

Low-level operations transform image data to image data. This means that such operators deal directly with image matrix data at the pixel level. Examples of such operations include color transformations, gamma correction, linear or nonlinear filtering, noise reduction, sharpness enhancement, frequency domain transformations, etc. The ultimate goal of such operations is to either enhance image data, possibly to emphasize certain key features, preparing them for viewing by humans, or extract features for processing at the intermediate-level. These operations can be further classified into point, neighborhood (local), and global operations [6]. Point operations are the simplest of the low-level operations since a given input pixel is transformed into an output pixel, where the transformation does not depend on any of the pixels surrounding the input pixel. Such operations include arithmetic operations, logical operations, table lookups, threshold operations, etc. The inherent DLP in such operations is obvious, as depicted in Fig. 1 (a), where the point operation on the pixel shown in black needs to be performed across all the pixels in the input image. Local neighborhood operations are more complex than point operations in that the transformation from an input pixel to an output pixel depends on a neighborhood of the input pixel. Such operations include two-dimensional spatial convolution and filtering, smoothing, sharpening, image enhancement, etc. Since each output pixel is some function of the input pixel and its neighbors, these operations require a large amount of computations. The inherent parallelism in such operations is illustrated in Fig. 1 (b), where the local neighborhood operation on the pixel shown in black needs to be performed across all the pixels in the input image. Finally, global operations build upon neighborhood operations in which a single output pixel depends on every pixel in the input image (see Fig. 1 (c)).

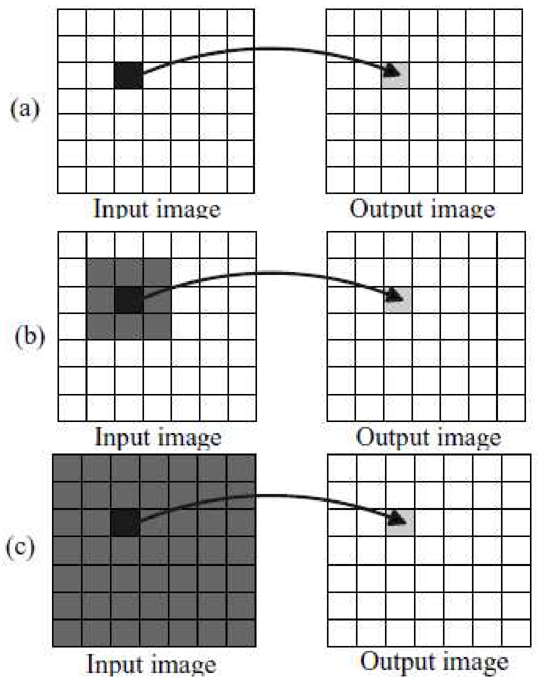

Fig.1.Parallelism in low-level image processing:
a) point b) neighborhood c) global

All low-level operations involve nested looping through all the pixels in an input image with the innermost loop applying a point, neighborhood, or global operator to obtain the pixels forming an output image. For this reason low-level operations are excellent candidates for exploiting DLP.

The higher degree operations are difficult to implement for real time execution. Intermediate-level operations transform image data to a slightly more abstract form of information by extracting certain attributes or features of interest from an image. This means that such operations also deal with the image at the pixel level, but a key difference is that the transformations involved cause a reduction in the amount of data from input to output. The goal by carrying out these operations (which include segmenting an image into regions/objects of interest, extracting edges, lines, contours, or other image attributes of interest such as statistical features) is to reduce the amount of data to form a set of features suitable for further high-level processing. Some intermediate-level operations are also data intensive with a regular processing structure, thus making them suitable candidates for exploiting DLP.

High-level operations interpret the abstract data from the intermediate-level, performing high level knowledge-based scene analysis on a reduced amount of data. These types of operations (for example recognition of objects) are usually characterized by control or branch-intensive operations. Thus, they are less data intensive and more inherently sequential rather than parallel.

2.2.Software Architecture Design

While translating a source code from a research development environment to a real-time environment is an involved task, it would be beneficial if the entire software system is well thought out ahead of time. Considering that real-time image processing systems usually consist of thousands of lines of code, proper design principles should be practiced from the start in order to ensure maintainability, extensibility, and flexibility in response to changes in the hardware or the algorithm [7]. One key method of dealing with this problem is to make the software design modular from the start, which involves abstracting out algorithmic details and creating standard interfaces or application programming interfaces (APIs) to provide easy switching among different specific implementations of an algorithm. Also beneficial is to create a hierarchical, layered architecture where standard interfaces exist between the upper layers and the hardware layer to allow ease in switching out different types of hardware so that if a hardware component is changed, only minor modifications to the upper layers will be needed.

In addition, because in real-time image processing system, certain tasks or procedures have strict real time deadlines, while other tasks have firm or soft real-time deadlines, it is useful to utilize a real time operating system in order to be able to manage the deadlines and ensure a smoothly running system. Real-time operating systems allow the assignment of different levels of priorities to different tasks. With such an assignment capability, it becomes possible to assign higher priorities to hard real-time deadline tasks and lower priorities to other firm or soft real-time tasks [8].

3. Performing real time image processing on a distributed platform

3.1 Parallel platform model and scheduling principles

Our system model consists of P processor units. Each processor p_i has capacity $c_i > 0$, $i = 1,2,\ldots, P$. The capacity of a processor is defined as its speed relative to a reference processor with unit-capacity. We assume for the general case that $c_1 \leq c_2 \leq \ldots \leq c_P$. The *total capacity* C of the system is defined as $C = \sum_{i=1}^{P} c_i$. A system is called *homogeneous* when $c_1=c_2\ldots=c_P$. The platform is conceived as a distributed system [9]. Each machine is equipped with a single processor. In other words, we do not consider interconnections of multiprocessors. The main difference with multiprocessor systems is that in a distributed system, information about the system state is spread across the different processors. In many cases, migrating a job from one processor to another is very costly in terms of network bandwidth and service delay [10], and that the reason that we have considered for the beginning only the case of data parallelism for a homogenous system. The intention was to test the general case of image processing with both data and task parallelism, by developing a scheduling policy with two components [11]. The *global* scheduling policy decides to which processor an arriving job must be sent, and when to migrate some jobs. At each processor, the *local* scheduling policy decides when the processor serves which of the jobs present in its queue.

Jobs arrive at the system according to one or more interarrival-time processes. These processes determine the time between the arrivals of two consecutive jobs. The *arrival time* of job j is denoted by A_j. Once a job j is completed, it leaves the system at its *departure time* D_j. The *response time* R_j of job j is defined as $R_j = D_j - A_j$. The

service time S_j of job j is its response time on a unit-capacity processor serving no other jobs; by definition, the response time of a job with service time s on a processor with capacity c' is s/c'. We define the *job set* $J(t)$ *at time t* as the set of jobs present in the system at time t:
For each job $j\in J(t)$, we define the *remaining work* $W_j^r(t)$ *at time t* as the time it would take to serve the job to completion on a unit-capacity processor. The *service rate* $\sigma_j^r(t)$ *of job j at time t* ($A_j\leq t<D_j$) is defined as: $\sigma_j^r(t)=\lim_{\tau\to t}\frac{dW_j^r(\tau)}{d\tau}$. The *obtained share* $\omega_j^s(t)$ *of job j at time t* ($A_j\leq t<D_j$) is defined as: $\omega_j^s(t)=\sigma_j^r(t)/C$. So, $\omega_j^s(t)$ is the fraction of the total system capacity C used to serve job j, but only if we assume that $W_j^r(t)$ is always a piecewise-linear, continuous function of t. Considering $W_j^r(A_j)=S_j$ and $W_j^r(D_j)=0$ we have $\int_{A_j}^{D_j}\omega_j^s(t)dt=\int_{A_j}^{D_j}\sigma_j^r(t)dt=S_j/C\cdot$
One can define an upper bound on the sum of the obtained job shares of any set of jobs {1,…,J}as:
$\omega_{\max}(t)=C^{-1}\sum_{i=1}^{\min(J,P)}c_i$.

3.2 A case study: lines detection

3.2.1 Theoretical background

Usually the 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 [12]. Theoretically, in two-dimensions, line points are detected by considering the second order directional derivative in the gradient direction. For a line point, the second order directional derivative perpendicular to the line is a measure of line contrast, given by $\lambda=f_{ww}(x,y)$ where *f(x,y)* is the grey-value function and the indices *w* denote differentiation in the gradient direction. Bright lines are observed when $\lambda<0$ and dark lines when $\lambda>0$. In practice, one can only measure differential expressions at a certain observation scale. By considering Gaussian weighted differential quotients in the gradient direction, $f_{ww}^{\sigma}=G_{ww}(\sigma)*f(x,y)$, a measure of line contrast is given by $r(x,y,\sigma)=\sigma^2\left|f_{ww}^{\sigma}\right|\frac{1}{b^{\sigma}}$ where σ,, the Gaussian standard deviation, denotes the scale for observing the line structure, and where line brightness b is given by

$$b^{\sigma}=\{\begin{array}{l} f^{\sigma}if\ldots f_{ww}^{\sigma}\leq 0\\ W-f^{\sigma}otherwise\end{array}$$

Line brightness is measured relative to black for bright lines, and relative to white level W (255 for an 8-bit camera) for dark lines [13].
The response of the second order directional derivate λ does not only depend on the image data, but it is also affected by the Gaussian smoothing scale σ. Because a line has a large spatial extent along the line direction, and only a small spatial extent (i.e., the line width) perpendicular to the line, the Gaussian filter should be tuned to optimally accumulate line evidence. For directional filtering anisotropic Gaussian filters may be used of scale σ_v and σ_w, for longest and shortest axis, respectively. Line contrast is given by:

$$r'(x,y,\sigma_v,\sigma_w)=\sigma_v\sigma_w\left|f_{ww}^{\sigma_v,\sigma_w}\right|\frac{1}{b^{\sigma_v,\sigma_w}}$$

The optimal filter orientation may be different for each position in the image plane, depending on line evidence at the particular image point under consideration. The final line detection filter, parameterized by orientation θ, smoothing scale σ_v in the line direction, and differentiation scale σ_w perpendicular to the line, is given by

$$r''(x,y,\sigma_v,\sigma_w,\theta)=\sigma_v\sigma_w\left|f_{ww}^{\sigma_v,\sigma_w,\theta}\right|\frac{1}{b^{\sigma_v,\sigma_w,\theta}}$$

where $f_{ww}^{\sigma_v,\sigma_w,\theta}=G_{ww}(\sigma_v,\sigma_w,\theta)*f(x,y)$
When the filter is correctly aligned with the line, and σ_v, σ_w are optimally tuned to capture the line, filter response is maximal. Hence, the maximum per pixel line contrast over the filter parameters yields line detection:

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

The final result is obtained by considering the maximum response per pixel over all filter results. This yields the optimal orientation θ, an estimate of line thickness σ_w, the best smoothing size σ_v, and the line contrast *R(x,y).*

3.2.2 Software implementation of the directional filtering algorithm

There are many different ways to implement a directional filtering algorithm. For example, one can create for each orientation a new filter based on σ_v and σ_w. This yields a rotation of the filters, while the orientation of the input image remains fixed. Another possibility is to keep the orientation of the filters fixed, and to rotate the input image instead. Yet another solution is to integrate the notion of orientation in the filter operation itself. In this case image pixels are accessed not only according to the size of the neighborhood of the filter, but also on the basis of the given orientation [14]. From these solutions, the second, who consists in applying fixed filters to rotated image data, seems to be more suitable for parallelization. In order to stress the possibility to execute parallel operations, let consider first the main steps of a sequential implementation.

The first step consists in rotating the original input image for a given orientation *θ*. This operation is made by a dedicated routine *Rotate_Image*. Then, for all combinations *(σ_v, σ_w)* the filtering is performed by six operations executed in sequence by six dedicated routines, as follows: 1) *Filter 1* to compute $f_{ww}^{\sigma_v,\sigma_w,\theta}$; 2) *Filter 2* to compute $b^{\sigma_v,\sigma_w,\theta}$ (both filtering operations are generalized Gaussian convolutions performed by applying two 1-dimensional filters; 3) *Binary_Op1*, a binary pixel operation having an image as argument; 4) *Binary_Op2*, a binary pixel operation having an constant value as argument; 5) *Back_Rotate_Image* to match the orientation of the original input image; 6) *Contrast* to obtain the maximum response.

It is to note that on a state-of-the-art sequential machine the program may take from tens of seconds up to minutes to complete, depending on the size of the input image and the extent of the chosen parameter subspace. Consequently, for the directional filtering program parallel execution is highly desired.

The above described program may be processed in parallel in two different schedules. In the first schedule all dedicated routines are forced to run in parallel, using all available processing units. The second schedule differs from the first in that the last two operations in the innermost loop of the program are run on one node only. In both schedules the *Original_Image* structure must be broadcast to all nodes. This is because the structure is applied in the initial rotation operation. In addition, in both schedules the first four operations in the innermost loop can be 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 convolutions.

In the first schedule the last two operations in the innermost loop are run in parallel as well. This requires the distributed image *Binary_Op1* to be available in full at each node, because it has an access pattern of type 'other' in the back-rotation operation. This can be achieved by executing a gather-to-all operation, which is logically equivalent to a gather operation followed by a broadcast. Finally, a partial maximum response image *Contrast* is calculated on each node, which requires a final gather operation to be executed just before termination of the program. In the second schedule the last two operations are not executed in parallel. As a result, the intermediate result image after *Binary_Op2* that produces both the back-rotated image needs to be gathered to the single node, as well as the complete maximum response image.

3.2.3 Experimental results

A test image was processed first on a single processing unit, then on a test network configured as a cluster with 2, 4 or 8 nodes, each node being a processor unit working at 1 GHz with 128 MByte RAM. For each instruction utilized in the directional filtering algorithm two measurements were executed, for images having 200^2 or 1000^2 elements. Table 1 offers the measured results for the processing times of an image with 1024x1024 pixels (see fig. 2: a) original, b) after processing with 12 orientations and 4 combinations (σ_v, σ_w).

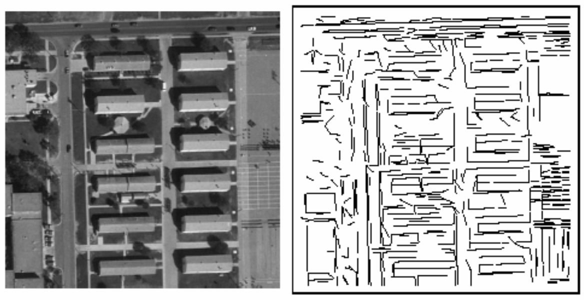
a) b)

Fig. 2. Test image for line detection

Table 1. Comparison of processing times for different cluster dimensions

Number of processors	Measured duration [s]	
	Schedule 1	Schedule 2
1	5.56	5.56
2	2.90	4.01
4	1.60	3.22
6	0.97	2.82
8	1.21	2.95

A schedule is preferred if the set of operations unique to that schedule is faster than the set of operations unique to another schedule (i.e., not in the set of operations common to both schedules). Hence, for the directional filtering program the schedule in which *all* operations are run in parallel is preferred if:

$$\theta_\sigma(P_{rotate}(size/N)+P_{max}(size/N)+P_{bcast}(size/N)+P_{gather}(size/N)) < \theta_\sigma(P_{rotate}(size)+P_{max}(size))$$

where N denotes the number of processing units and θ_σ denotes the size of the parameter subspace. For the first schedule the large number of broadcast operations is expected to have the most significant impact on performance. For the second schedule, on the other hand, the many rotations of non-partitioned image data are expected to be costly. Another difference between the two schedules is the fact that the total duration decrease proportional with the number of nodes only for schedule 2. For the schedule 1 there is an optimal structure with 6 nodes, then when the number of nodes is grater the processing duration begins to rise again.

4 Conclusions

The experiments show how to use parallelizable patterns, obtained for typical low level image processing operations. In our study case the performance model is highly accurate for parallel processing using convolution functions. Given the results we are confident in that the proposed software architecture forms a powerful basis for automatic parallelization and optimization of a wide range of image processing applications.

Regarding the potential of the parallel platform for image processing, in the near future we will focus our attention on the improvement of the scheduling component, by using processor units with different processing capacities and also other service policy for the queue of jobs. We will continue implementing example programs to investigate the implication of parallelization of typical applications in the area of real-time image processing, trying to improve the performances by supporting the execution of a sequence of algorithms on the same block and by dynamical reconstruction of the post processed image.

ACKNOWLEDGEMENTS
This work was partially supported by the Romanian Ministry of Education and Research under PN2 Grants 61-031/2007 and 11-031/2007.

References:

[1] A. Bovik, Introduction to Digital Image and Video Processing, in *Handbook of Image & Video Processing*, A. C. Bovik, Ed., Elsevier Academic Press, 2005.

[2] E. Dougherty and P. Laplante, *Introduction to Real-time Imaging*. SPIE Press/IEEE Press, 1995.

[3] N. Kehtarnavaz, *Real-Time Digital Signal Processing Based on the TMS320C6000.* Amsterdam, Elsevier, 2004.

[4] H. Hunter and J. Moreno, A New Look at Exploiting Data Parallelism in Embedded Systems, *Proc. of the Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems*, 2003, pp. 159–169.

[5] S. Kyo, S. Okazaki, and T. Arai, An Integrated Memory Array Processor Architecture for Embedded Image Recognition Systems, *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005, pp. 134–145.

[6] C. Soviany, *Embedding Data and Task Parallelism in Image Processing Applications*, Ph.D. Dissertation, Delft University of Technology, The Netherlands, 2003.

[7] R. Sangwan, R. Ludwig, P. Laplante and C. Neill, Performance Tuning of Imaging Applications Through Pattern Based Code Transformation, *Proc. of SPIE-IS&T Electronic Imaging Conf.on Real-Time Imaging*, SPIE Vol. 5671, 2005, pp. 1–7.

[8] R. Dobrescu, D. Popescu, M. Nicolae, H. Humaila, Real time dependable communication infrastructure for a collaborative groupware system, Proc. of the 1st Int. Conf. WSEAS MEQAPS'09, vol.1, p.207-212

[9] D.H.J. Epema and J.F.C.M. de Jongh. Proportional Share-Scheduling in Single-Server and Multiple-Server Computing Systems. *Performance Evaluation Review*,27(3):7–10, 1999.

[10] G.Agosta, S. Crespi Reghizzi, G. Falauto, M. Sykora, JIST: Just-in-Time Scheduling Translation for Parallel Processors, *Third Int. Symp. on Parallel and Distributed Computing (ISPDC/HeteroPar'04)*, 2004, pp. 122-132

[11] J.M. Geusebroek, A.W.M. Smeulders, and H. Geerts. A Minimum Cost Approach for Segmenting Networks of Lines. *International Journal of Computer Vision,* 43(2):99-lll, 2001.

[12] M. Dobrescu. *Distributed Image Processing Techniques for Multimedia Applications*, Ph.D. Thesis, Politehnica Univ. of Bucharest, 2005.

[13] F.J. Seinstra, D. Koelma, and J.M. Geusebroek. A Software Architecture for User Transparent Parallel Image Processing. *Parallel Computing,* 28 (7-8), 2002, pp. 967-993.

[14] E. Davies, *Machine Vision: Theory, Algorithms, Practicalities*. San Francisco,CA: Morgan Kauffmann Publishers, 2005.