Journal of Supercomputing, 14, 233–255 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

An Efficient Parallel Algorithm for Computing the Gaussian Convolution of Multi-dimensional Image Data

HOI-MAN YIP, ISHFAQ AHMAD AND TING-CHUEN PONG {hmyip,iahmad,tcpong}@cs.ust.hk
Department of Computer Science, The Hong Kong University of Science and Technology, Hong Kong

(Received September 4, 1997; final version accepted July 29, 1998.)

Abstract. In this paper, we propose a parallel convolution algorithm for estimating the partial derivatives of 2D and 3D images on distributed-memory MIMD architectures. Exploiting the separable characteristics of the Gaussian filter, the proposed algorithm consists of multiple phases such that each phase corresponds to a separated filter. Furthermore, it exploits both the task and data parallelism, and reduces communication through data redistribution. We have implemented the proposed algorithm on the Intel Paragon and obtained a substantial speedup using more than 100 processors. The performance of the algorithm is also evaluated analytically. The analytical results confirming with the experimental results indicate that the proposed algorithm scales very well with the problem size and number of processors. We have also applied our algorithm to the design and implementation of an efficient parallel scheme for the 3D surface tracking process. Although our focus is on 3D image data, the algorithm is also applicable to 2D image data, and can be useful for a myriad of important applications including medical imaging, magnetic resonance imaging, ultrasonic imagery, scientific visualization, and image sequence analysis.

Keywords: image processing, Gaussian filter, parallel algorithms

1. Introduction

Image processing and computer vision technologies are required in a variety of applications in many areas of science and engineering. However, due to the limitation of the current state-of-the-art technology, conventional image processing and computer vision systems have predominantly focused on the processing of 2D images. The need for 3D images is now widely felt in fields such as medical imaging [4, 8], image sequence analysis [21], and scientific visualization [15]. In medical imaging field, for instance, a 3D image is a sequence of slices from tomographic techniques such as computed tomography (CT), magnetic resonance imaging (MRI), nuclear medicine imagery (NMI), and ultrasonic imagery. Other 3D images are also prevalent in seismic imaging, numerical simulation experiments, and atmospheric sciences.

Since the size and computational requirements of the image data, particularly 3D data, are usually huge for a single-processor system such as a general-purpose workstation, efforts have been directed towards using parallel or distributed architectures. Most previous efforts have concentrated on the parallelization of visualization aspects. However, in scientific visualization and other applications of empirical data interpretation, image analysis operators for the transformation and enhancement of the raw data are required. With these operators, the salient features embedded in the data can become discernible and quantifiable and thus provide useful information for decision making. For both 2D and 3D imaging systems, the partial derivatives of the image are essential elements in the image analysis process [21, 18].

The estimation of partial derivatives is very sensitive to noise which may usually be introduced into the raw images during the image formation process. Hence it is critical to filter out the noise of the raw images for the success of the whole system. Among many other noise suppression methods, the Gaussian based smoothing method, also called Laplacian of Gaussian (LoG), is the most commonly used method, because of its well localized property in both spatial and frequency domains.

The convolution operations are computationally intensive for a large filter on a fine resolution image. Dykes *et al.* [9] showed that a major problem with a sequential convolution is the heavy demand for memory accesses. Because of the fundamental nature of the convolution operation, much attention has been devoted to the development of efficient fine-grained multicomputer parallel algorithms [6, 23] as well as the development of special-purpose hardware [1, 17, 7, 16, 10, 14]. Alternatively, convolutions can also be computed with the aid of Fourier Transforms. Parallel algorithms based on this method have been proposed in [25]. However, the overhead of Fourier Transformation is too high and is not practical for large images.

The problem of surface reconstruction is to track the surface of objects in the 3D data volume by means of connected-component search and to generate triangular facets for rendering. Fuchs *et al.* [11] reduced the problem of surface reconstruction from planar contours to finding a minimum cost cycle in a directed toroidal graph. Talele *et al.* [28] developed a parallel algorithm based on the shortest-path on a BBN TC2000 parallel computer, a shared-memory machine. More recently, Lorensen and Cline [19] proposed an algorithm called *Marching Cubes* that generates polygonal representations of the anatomy directly from the segmented volume data. Hansen and Hinker [13] designed a SIMD version of Marching Cubes for fine-grain massively parallel surface extraction. The algorithm of Marching Cubes has also been parallelized by Yoo and Chen [31] on a special-purpose parallel multicomputer for computer graphics called Pixel-Planes 5 [12].

In most studies of parallel surface reconstruction [24, 28], a master-slave parallel computational model is employed where the master node initiates, controls and schedules the tasks on slave nodes. This model has a limited scalability because the master node becomes the bottleneck for large computation, and, therefore, cannot be implemented on a large number of processors.

Another important issue to be considered for parallel implementation is load balancing. Zhang and Deng [32] indicated that the straight-forward uniform data partitioning method is the most effective because the additional overheads introduced from the static and dynamic scheduling methods cost more than the savings from balancing the computation.

Our objective is to develop a parallel algorithm for the Gaussian convolution on distributed memory MIMD parallel architectures such as the Intel Paragon. The proposed algorithm, which exploits the separable characteristics of the Gaussian filters, is a multi-phase algorithm in which each phase corresponds to a particular separated filter. The communication overhead is minimized by the data redistribution processes between every two phases. The proposed convolution algorithm has been applied to design and implementation of an efficient two-level parallel scheme for the 3D surface tracking process using the 3D image. In our two-level parallel strategy, we use a combination of task and data parallelism. For the task parallelism, the tasks are distributed across various groups of processors. For the data parallelism, within each group, data is partitioned across sub-meshes of processors. Although our focus is on 3D image data, the parallel convolution algorithm is also applicable to 2D image data.

The rest of this paper is organized as follows. Section 2 shows how Gaussian filter and differential operator can be combined into a single operator. The parallel algorithm of the Gaussian convolution is presented in Section 3. In Section 4, we study the performance of the parallel algorithm analytically. The parallel surface building algorithm is presented in Section 5. The experimental performance of the algorithm is given in Section 6, and the last section provides concluding remarks.

2. Gaussian convolution

The estimation of partial derivatives of an image is sensitive to noise. Hence some kind of noise suppression operations is required to improve the quality of the estimation. Previous strategies on noise suppression include simple averaging, surface fitting, non-linear filtering and Gaussian smoothing. Among these methods, the Gaussian has a number of desirable properties which make it the most commonly used smoothing filter. For examples, Gaussian filter is the only low-pass filter that has good localization property in both spatial and frequency domains, and is decomposable and rotationally invariant. In fact, the two most popular edge detectors, Marr-Hildreth operator [20] and Canny operator [5], are designed based on Gaussian smoothing. Moreover, Gaussian filter is closely related to the technique of multi-resolution or multi-scale processing [30] because it can be employed to create images with resolution from coarse to fine. However, it should be noticed that Gaussian smoothing may cause displacement of the feature locations [3]. Fortunately, Ulupinar and Medioni [29] have suggested a solution for refining the feature locations.

Basically, the Gaussian filter smoothes the image by removing fine structures that are smaller than the window size of the filter. These fine structures are usually the noise on the image. The filter is a digital version of the following Gaussian function:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right). \qquad [1]$$

Here σ is the standard deviation. The 2D Gaussian filter is a digital version of the product of two 1D Gaussian:

$$G(x, y) = G(x)G(y) \tag{2}$$

$$= \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right). \tag{3}$$

The 3D filter can be determined similarly. The parameter, σ, determines the support of the filter. We can set the parameter σ to control the resolution of the smoothed image.

Theoretically, the support of the Gaussian filters is infinity. In discrete realization, Sotak and Boyer [27] show that a support of $4w$, where $w = 2\sqrt{2}\sigma$, is most appropriate. A support smaller than $4w$ may cause distortion of the filter in the frequency domain. A support larger than $4w$ results in increased computational burden with only negligible improvement in the filter performance.

The discussion below shows how the Gaussian filter and the differential operator can be combined. We will focus on 2D case in the following derivation. The derivation for 3D follows naturally.

The smoothed version, $J(x, y)$, of the continuous image, $I(x, y)$, is given by

$$J(x, y) = G(x, y) \otimes I(x, y) \tag{4}$$

where $\otimes$ denotes convolution. Hence the partial derivative of J is

$$\frac{\partial^{p+q}}{\partial x^p \partial y^q} J(x, y) = \left[\frac{\partial^{p+q}}{\partial x^p \partial y^q} G(x, y)\right] \otimes I(x, y). \tag{5}$$

Therefore, in the discrete domain, any $I_{x^p y^q}$ can be estimated by convolving the image with the digital version of a corresponding Gaussian derivative, which can be derived analytically, as follows

$$J_{x^p y^q}(i, j) = G_{x^p y^q}(i, j) \otimes I(i, j) \tag{6}$$

$$= \sum_{a=-m/2}^{m/2} \sum_{b=-m/2}^{m/2} G_{x^p y^q}(a, b) I(i - a, j - b), \tag{7}$$

where m is the width of the 2D Gaussian filter.

Since the Gaussian filter is decomposable [22], that is

$$G(x, y) = G(x) \otimes G(y), \tag{8}$$

the partial derivative of Gaussian is also decomposable and is given by

$$\frac{\partial^{p+q}}{\partial x^p \partial y^q} G(x, y) = \frac{\partial^p}{\partial x^p} G(x) \otimes \frac{\partial^q}{\partial y^q} G(y). \tag{9}$$

Denote $(\partial^p/\partial x^p)G(x)$ by G_x^p. From Equation 5 we have

$$J_{x^p y^q} = \left(G_x^p \otimes G_y^q\right) \otimes I(x,y) \quad [10]$$

$$= G_x^p \otimes \left(G_y^q \otimes I(x,y)\right). \quad [11]$$

Although the results given by Equation 5 and Equation 11 are the same, there is a significant difference in the number of operations (multiplication) for convolving with the original and separated filters. Suppose the filter size is $m \times m$ and the image size is $n_x \times n_y$. The number of multiplications in Equation 5 is $m^2 n_x n_y$ while it is only $2mn_x n_y$ in Equation 11. The reduction in number of multiplications is by a factor of m which can be very significant for large σ.

The above derivation leads naturally to 3D image. The partial derivative of the smoothed 3D image is given by

$$\frac{\partial^{p+q+r}}{\partial x^p \partial y^q \partial z^r} J(x,y,z) = \left[\frac{\partial^{p+q+r}}{\partial x^p \partial y^q \partial z^r} G(x,y,z)\right] \otimes I(x,y,z). \quad [12]$$

If separated filters are applied, the above equation becomes

$$J_{x^p y^q z^r} = G_x^p \otimes \left(G_y^q \otimes \left(G_z^r \otimes I(x,y,z)\right)\right). \quad [13]$$

The saving in computation is even more for the 3D case. Suppose the filter size is $m \times m \times m$ and the image volume size is $n_x \times n_y \times n_z$. The number of multiplications in the separated convolution is reduced from $m^3 n_x n_y n_z$ to $3mn_x n_y n_z$.

The implementation of Equations 11 and Equation 13 on serial computers is not difficult. For Equation 11, we first perform 1D convolution of each column of $I(x,y)$ with Gaussian derivative, G_y^q, to create an intermediate image J_{y^q}; then we perform another 1D convolution of each row of J_{y^q} with G_x^p to give $J_{x^p y^q}$, i.e.

$$J_{y^q} = G_y^q \otimes I(x,y). \quad [14]$$

$$J_{x^p y^q} = G_x^p \otimes J_{y^q}. \quad [15]$$

Similarly, for Equation 13, we first perform 1D convolution in z-direction, then in y-direction and finally in x-direction, i.e.

$$J_{z^r} = G_z^r \otimes I(x,y,z). \quad [16]$$

$$J_{y^q z^r} = G_y^q \otimes J_{z^r}. \quad [17]$$

$$J_{x^p y^q z^r} = G_x^p \otimes J_{y^q z^r}. \quad [18]$$

3. The proposed algorithm

In this section, we describe the proposed parallel algorithm for computing the derivatives of the image. We first show why the simple strategy to parallelize the

2-D separated convolution by conventional block partitioning is not optimal for our purpose. Analytical results will be presented in next section.

In a conventional approach, image is partitioned into regular blocks which are distributed to a group of processors. Each processor then performs local convolution on its own data. In order to compute the convolution of the boundary pixels, each processor has to access the data of neighboring processors. Communication overhead can be avoided by duplicating the boundary of neighboring processors into a processor's own memory space. This method is perhaps the best way for parallelizing the conventional one step convolution of non-separable filters. However, it is not practical for convolution with larger filters because it requires the duplication of a large amount of boundary data. Moreover, as shown in the previous section, our filters are separable where two 1D convolutions are performed for 2D image, and three 1D convolutions are performed for 3D image. In the case of 2D image, for example, the second convolution depends on the result of the first. So the first convolution has to be performed on all data including the duplicated boundaries.

Since the conventional strategy of fixed partition is not optimal for our filters, we propose to redistribute the data blocks among the tasks between the convolutions of two 1D Gaussian filters. The algorithm works as follows. First, the image is divided into regular blocks by a column-wise striped partitioning (see Figure 1(a)). The blocks are distributed to a group of processors connected as an array. Each processor convolves its own data with 1D filter, G_y^q, in y-direction. Next, all processors work cooperatively to exchange data blocks with each other. The arrow curves in Figure 1 indicate how the data blocks are redistributed. The blocks with a dot remain in the same processor without inter-processor redistribution. The result of this redistribution is that the image is divided into regular blocks with row-wise partitioning (Figure 1(b)). This process is similar to a parallel transposition of the whole image (see Figure 2). After redistribution, each processor performs another 1D convolution with G_x^p in x-direction. The final result is obtained by merging the data blocks of the processors together.

The following is the pseudo code of the algorithm where B_{yi} and B_{xi} are data blocks held by processor P_i before and after redistribution, respectively, and M is the number of processors.

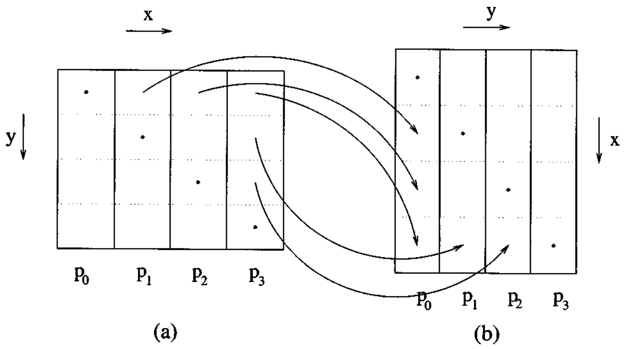


Figure 1. Redistribution of the image.

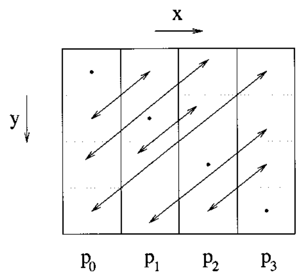


Figure 2. Transposition of the image.

```
1.  **function** $\partial I \leftarrow Gaussian2(I, p, q, M)$
2.    **for all** $P_i, i \leftarrow 0, \cdots, M-1$ {
3.      $B_{yi} \leftarrow Split(I, M)$;
4.      $Conv1D(B_{yi}, G_y^q)$;
5.      **for** $j \leftarrow 0, \cdots, M-1$ {
6.        $B_{xi}(j) \leftarrow B_{yj}(i)$;
7.      }
8.      $Conv1D(B_{xi}, G_x^p)$;
9.    }
10.   $\partial I \leftarrow Merge(B_{xi})$;
```

During the course of redistribution, the sub-images B_{yi} and B_{xi} are logically divided into M regular blocks which is represented by $B_{yi}(j)$ and $B_{xi}(j)$, respectively. $B_{yi}(j)$ and $B_{xi}(j)$ are the unit of data exchange among the processors.

As a byproduct of the mechanism, redistribution maintains the locality of sub-image in each processor which minimizes the cache miss rate, and hence improves the performance of the algorithm [26]. Although cache miss may be negligible for convolution of 2D images with small size, it may be one of the major sources of performance degradation for 3D case because the size of 3D images is usually huge. follows. Before the 1(a)) to store column major because the 1D performed for each column of the sub-image. the sub-image (Figure convolution is

Similarly, the same philosophy of redistributing the data block among processors is generalized to parallelize the estimation of partial derivatives of 3D spatiotemporal volume. The algorithm works as follows. The image volume is first partitioned into smaller volumes along the z-direction as shown in Figure 3(a). Each processor convolves its own volume with G_t^r in t-direction. The whole image volume is then redistributed among the processor mesh. As a result, the volume is rotated above x-axis (see Figure 3(a) and (b)) and the volume is partitioned along the y-direction into smaller volumes. Each processor performs the second convolution on its own volume with G_y^q in y-direction. Next, the volume is once again redistributed such that it is rotated above z-axis (see Figure 3(b) and (c)) and the volume is partitioned along the x-direction.

Finally, the results of the third convolution with G_x^p in x-direction are merged together. Totally, we need to redistributed the volume among the mesh-connected

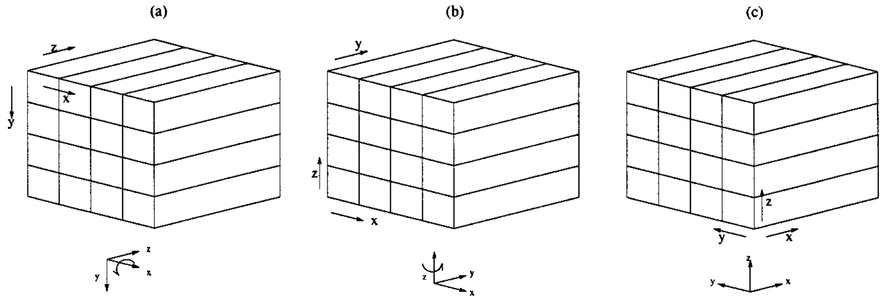


Figure 3. Rotation of image volume.

processors twice (see Figure 4). Throughout the process, good locality of data blocks is maintained to reduce the cache missing rate.

The following is the pseudo code of the algorithm. In the pseudo code, B_{tij}, B_{yij} and B_{xij} are data blocks held by P_{ij} at three time steps separated by the redistribution operations. During the algorithm, they are divided into N sub-blocks which are represented by $B_{tij}(k)$, $B_{yij}(k)$ and $B_{xij}(k)$. These sub-blocks are exchanged among the processors during the redistribution processes.

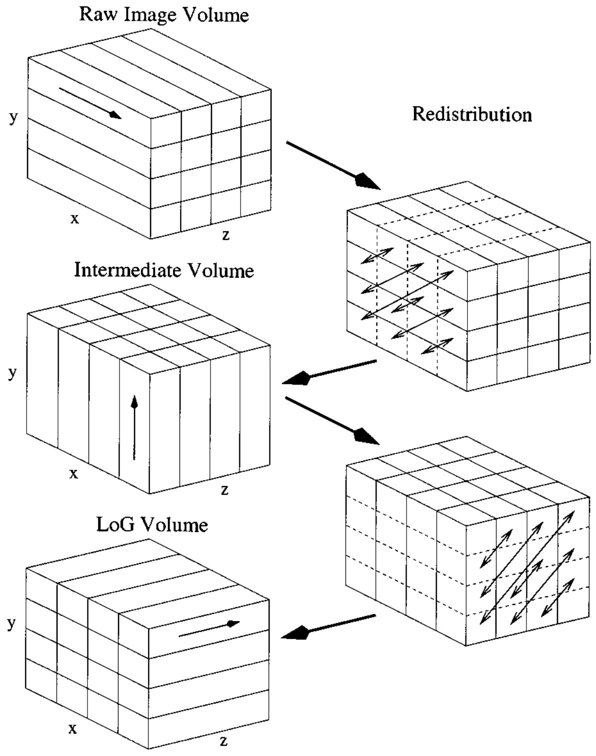


Figure 4. Redistribution of image volume.

```
1.  **function** $\partial I \leftarrow Gaussian3(I, p, q, r, M, N)$
2.    **for all** $P_{ij}$, $i \leftarrow 0, \ldots, M-1$, $j \leftarrow 0, \ldots, N-1$ {
3.      $B_{zij} \leftarrow Split(I, M, N)$;
4.      $Conv1D(B_{zij}, G_z^r)$;
5.      **for** $k \leftarrow 0, \ldots, M-1$ {
6.        $B_{yij}(k) \leftarrow B_{zkj}(i)$;
7.      }
8.      $Conv1D(B_{yij}, G_y^q)$;
9.      **for** $k \leftarrow 0, \cdots, N-1$ {
10.       $B_{xij}(k) \leftarrow B_{ykj}(i)$;
11.     }
12.     $Conv1D(B_{xij}, G_x^p)$;
13.   }
14.   $\partial I \leftarrow Merge(B_{xij})$;
```

4. Analysis

In this section, we analyze the performance of the proposed algorithm. The analysis is performed separately on 2D image and 3D spatiotemporal volume. In general, the overhead due to the redistribution scheme is the internal transposition of the partition inside each processor as well as the redistribution of image data among the processors. The computation time of our redistribution system can then be modeled as

$$T = \frac{T_1}{N_p} + \alpha O_{comm} + \beta O_{int}, \quad [19]$$

where T_1 is the computation time by single processors, O_{int} is the number of steps required for internal redistribution and O_{comm} is the number of steps for communication (redistribution) among processors. The parameters α and β are the computation times required for one step of the intra- and inter-processor redistribution, respectively. The last two terms of Equation 19 are overheads of the redistribution algorithm. However, the possibility of cache misses is not captured by this model. Moreover, our analysis will not consider the communication delay caused by network congestion which depends much on the network configuration.

4.1. 2D image

Suppose the image size is $n_x \times n_y$, the filter size is m and the number of processors is M. For the redistribution based algorithm, the communication overhead for each processor is

$$O_{comm} = \frac{n_x n_y}{M}\left(1 - \frac{1}{M}\right),$$

and the intra-processor overhead is

$$O_{int} = \frac{n_x n_y}{M^2}.$$

Therefore, the total overhead for each processor is

$$\delta_{redist} = \frac{n_x n_y}{M},$$

if we assume both α and β to be one. Notice that the overhead is independent of the filter size, since the amount of communication depends only on the size of the image. This shows that substantial speedup can be achieved for large number of processors.

For the simple algorithm of checkerboard partitioning (fixed partition without redistribution), suppose the image is partitioned into $M_x \times M_y$ blocks where $M_x M_y = M$. Then the overhead per processor of duplicating boundary points is given by

$$\delta_{checker} = \frac{m}{2}\left(\frac{2n_x}{M_x} + \frac{2n_y}{M_y}\right) \quad [20]$$

$$= m\left(\frac{n_x}{M_x} + \frac{n_y}{M_y}\right). \quad [21]$$

Unlike the redistribution based algorithm, this overhead depends on the filter size and is not upper bounded.

The ratio of the two overhead terms is

$$R = \frac{\delta_{checker}}{\delta_{redist}} = \frac{m(n_x M_y + n_y M_x)}{n_x n_y}. \quad [22]$$

Suppose $M_x = M_y = \sqrt{M}$ and $n_x = n_y = n$, then the ratio becomes

$$R = \frac{m(2n\sqrt{M})}{n^2} = \frac{2m\sqrt{M}}{n}. \quad [23]$$

We can see that the redistribution based algorithm performs better if $R > 1$, i.e. $2m\sqrt{M} > n$. In other words, the performance of the redistribution based algorithm is optimized when the filter size and number of processors is large.

The computation time of the parallel program is

$$T_{2D}\left(n_x, n_y, m, M\right) = \frac{2mn_x n_y}{M} + \frac{n_x n_y}{M}\left(1 - \frac{1}{M}\right), \quad [24]$$

which is the sum of computation and overhead. The speedup of our algorithm is given by

$$\text{Speedup} = \frac{2mn_x n_y}{T_{2D}(n_x, n_y, m, M)} \quad [25]$$

$$= \frac{M}{1 + \frac{1}{2m}\left(1 - \frac{1}{M}\right)} \quad [26]$$

The efficiency of the algorithm is

$$\text{Efficiency} = \frac{1}{1+\frac{1}{2m}\left(1-\frac{1}{M}\right)} \qquad [27]$$

$$\geq \frac{1}{1+\frac{1}{2m}} \qquad [28]$$

which is lower bounded. If the filter size increases, this lower bound will be closer to one, the theoretical limit.

4.2. 3D volume

Suppose the size of the image sequence is $n_x \times n_y \times n_z$, the filter size is m and the number of processors is $M \times N = N_p$. The communication overhead for each processor is the sum of the two redistribution processes

$$O_{comm} = \frac{n_x n_y n_z}{MN}\left(1-\frac{1}{M}\right) + \frac{n_x n_y n_z}{MN}\left(1-\frac{1}{N}\right) \qquad [29]$$

$$= \frac{n_x n_y n_z}{N_p}\left(2-\frac{1}{M}-\frac{1}{N}\right). \qquad [30]$$

and the intra-processor overhead is

$$O_{int} = \frac{n_x n_y n_z}{N_p}\left(\frac{1}{M}+\frac{1}{N}\right). \qquad [31]$$

The total overhead is given by

$$\delta = O_{comm}\alpha + O_{int}\beta \qquad [32]$$

$$= \frac{n_x n_y n_z}{N_p}\left[\left(2-\frac{1}{M}-\frac{1}{N}\right)\alpha + \left(\frac{1}{M}+\frac{1}{N}\right)\beta\right]. \qquad [33]$$

If either M or N equals one (let's call it stripy partition), the total overhead becomes

$$\delta_1 = \frac{n_x n_y n_z}{N_p}\left[\left(1-\frac{1}{N_p}\right)\alpha + \left(1+\frac{1}{N_p}\right)\beta\right].$$

If $M = N = \sqrt{N_p}$ (let's call it checkerboard partition), the total overhead becomes

$$\delta_2 = 2\frac{n_x n_y n_z}{N_p}\left[\left(1-\frac{1}{\sqrt{N_p}}\right)\alpha + \left(\frac{1}{\sqrt{N_p}}\right)\beta\right].$$

Suppose α equals β, then both δ_1 and δ_2 will also be equal to

$$2\frac{n_x n_y n_z}{N_p}\alpha.$$

Suppose $\alpha = 1$ and $\beta = 0$, the ratio of these two overhead terms is

$$R = \frac{\delta_2}{\delta_1} \quad [34]$$

$$= \frac{2(1 - \frac{1}{\sqrt{N_p}})}{\left(1 - \frac{1}{N_p}\right)} \quad [35]$$

$$= \frac{2}{1 + \frac{1}{\sqrt{N_p}}} \quad [36]$$

Obviously, $1 \le R < 2$ for $N_p \ge 1$. Hence dividing the image volume using stripy partitioning is always preferable to checkerboard partitioning. This can be explained by the fact that only one redistribution requires transfer of data among processors and data is only exchanged within a processor for the other redistribution. Of course this is true only if $\beta = 0$, that is when the cost of intra-processor transposition is negligible.

The difference of δ_1 and δ_2 is

$$\delta_1 - \delta_2 = \frac{n_x n_y n_z}{N_p}\left[\left(2\frac{1}{\sqrt{N_p}} - 1 - \frac{1}{N_p}\right)\alpha + \left(1 + \frac{1}{N_p} - 2\frac{1}{\sqrt{N_p}}\right)\beta\right] \quad [37]$$

$$= \frac{n_x n_y n_z}{N_p}\left(1 + \frac{1}{N_p} - 2\frac{1}{\sqrt{N_p}}\right)(\beta - \alpha). \quad [38]$$

Note that for all $N_p \ge 1$, we have

$$\left(1 + \frac{1}{N_p} - 2\frac{1}{\sqrt{N_p}}\right) \ge 0.$$

Therefore, if $\beta > \alpha$, then $\delta_1 > \delta_2$, i.e. the overhead of the strip partitioning is larger than that of the checkerboard partitioning. On the contrary, if $\beta < \alpha$, then $\delta_1 < \delta_2$. Hence checkerboard partitioning is preferable to the strip partitioning if $\beta > \alpha$ and vice versa if $\beta < \alpha$.

The computation time of the parallel program with the communication overhead is

$$T_{3D}(n_x, n_y, n_z, m, M, N)$$

$$= \frac{3mn_x n_y n_z}{MN} + \frac{n_x n_y n_z}{MN}\left(2 - \frac{1}{M} - \frac{1}{N}\right)\alpha + \frac{n_x n_y n_z}{MN}\left(\frac{1}{M} + \frac{1}{N}\right)\beta. \quad [39]$$

The speedup of our algorithm is given by

$$\text{Speedup} = \frac{3mn_x n_y n_z}{T_{3D}(n_x, n_y, n_z, m, M, N)} \qquad [40]$$

$$= \frac{MN}{1 + \frac{1}{3m}(2 - \frac{1}{M} - \frac{1}{N})\alpha + \frac{1}{3m}(\frac{1}{M} + \frac{1}{N})\beta}. \qquad [41]$$

If $\alpha \approx \beta$, the efficiency of the algorithm is approximatively

$$\frac{1}{1 + \frac{2}{3m}\alpha}.$$

As in the 2D case, the efficiency for the 3D convolution is a function of the filter size.

5. Parallel surface reconstruction

In this section, we describe a parallel algorithm to detect the surface of the 3D image volume using LoG filter.

Our boundary detection algorithm is based on a surface building technique, named the *Weaving Wall* [2], which is designed for the analysis of image sequences. This process operates over images as they arrive from a sensor, knitting together, along a parallel frontier, connected descriptions of images as they evolve over time. The original Weaving Wall algorithm essentially consists of the following two phases:

Preprocessing. Firstly, we blur the image volume using the 3-D Gaussian filter. The purpose of this step is to filter out noise and unnecessary detail. Next, we compute the Laplacian of the blurred 3-D image. The Laplacian and the Gaussian operations can be combined by convolving the image volume with a Laplacian of Gaussian filter.

In three dimensional space, the LoG is defined as

$$LoG(x, y, z) = \nabla^2 G(x, y, z) \qquad [42]$$

$$= \frac{\partial^2}{\partial x^2}G(x, y, z) + \frac{\partial^2}{\partial y^2}G(x, y, z) + \frac{\partial^2}{\partial z^2}G(x, y, z). \qquad [43]$$

Hence the transformed image is given by

$$LoG \otimes I = \partial_x^2 G \otimes I + \partial_y^2 G \otimes I + \partial_z^2 G \otimes I. \qquad [44]$$

The processors are divided into three groups and each of them compute respectively the following partial derivatives using the parallel algorithm described in previous section:

$$\partial_x^2 G \otimes I = G''_x \otimes G_y \otimes G_z \otimes I. \quad [45]$$

$$\partial_y^2 G \otimes I = G_x \otimes G''_y \otimes G_z \otimes I. \quad [46]$$

$$\partial_z^2 G \otimes I = G_x \otimes G_y \otimes G''_z \otimes I. \quad [47]$$

Notice that two levels of parallelism are achieved. Task partitioning is applied on the three group to processors and data partitioning is applied within each group of processors.

After finishing convolutions in all three directions, each group of processor mesh will hold the partial results. The result of the convolution can be obtained by summing up the corresponding pixel values of the convoluted image volume obtained by the three groups of processors.

In our algorithm, the Summation operation is also carried out in parallel as follows:

- Group 2 and Group 3 send the first slice of convoluted results to Group 1.
- Group 1 and Group 3 send the second slice of convoluted results to Group 2.
- Group 1 and Group 2 send the third slice of convoluted results to Group 3.

Therefore, group 1, group 2 and group 3 processors sum up the corresponding pixel values for the first, second and third slice of the convoluted image volume, respectively. The output from this 3D convolution program will be used for surface reconstruction using zero-crossing detection algorithm which is described in the following section.

Reconstruction. The surface is defined at the zeros of Laplacian. Based on the results of the LoG convolution, each processor then performs a registration process to detect the surface patches at the locations of zero crossing. At the same time, these patches are linked together to form a surface. The registration and linking processes are so localized that the processor look at a neighborhood of eight voxels. In the process, a small $2 \times 2 \times 2$ window is employed to scan through the image signed volume. The direction of the scanning is from the first image to the last. For each image, the scanning is bottom to top, and within that, left to right.

At each step, the local surface structure inside the window will be determined. Notice that there are six neighbors for each voxel. If the sign of the Laplacian value is different for two neighbors, there is a facet between them. Hence, there are at most six facets for each voxel and twelve facets for each window.

Instead of determining all the facets of the window at the same time, we determine at most three new facets at each step. The position of these three facets are between the following voxel pairs:

$$\begin{array}{lcl} I_{i,j,k} & \text{and} & I_{i-1,j,k}, \\ I_{i,j,k} & \text{and} & I_{i,j-1,k}, \\ I_{i,j,k} & \text{and} & I_{i,j,k-1}. \end{array}$$

These new facets will be combined with the facets determined at the previous step and linked together to form the local surface. There may be more than one local surface inside a window.

In order to handle the boundary cases, each processor needs to communicate with its neighboring processor. There are six surfaces for each local volume of a processor. One obvious and simple scheme is that each processor sends the voxel values of its six volume surface to the corresponding neighbors, and receives from the same neighbors their boundary voxel values. The number of communication messages can, however, be further reduced by half if each processor sends only the voxels on three of the surfaces and receives the other three from its neighbors.

6. Experimental results

In this section, we present the experimental results of the proposed parallel Gaussian convolution algorithm and parallel surface reconstruction algorithm implemented on our 140-node Intel Paragon.

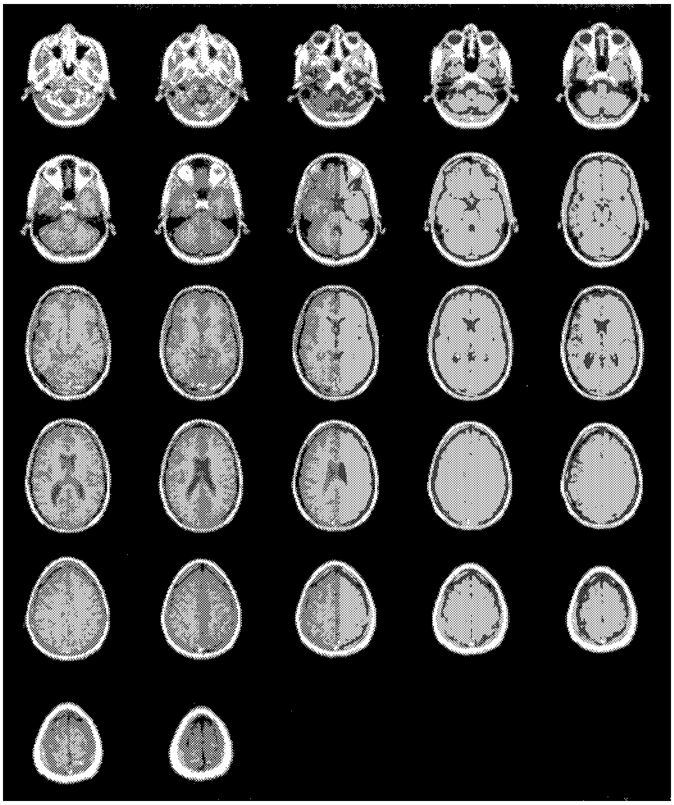


Figure 5. MRI slices of skull.

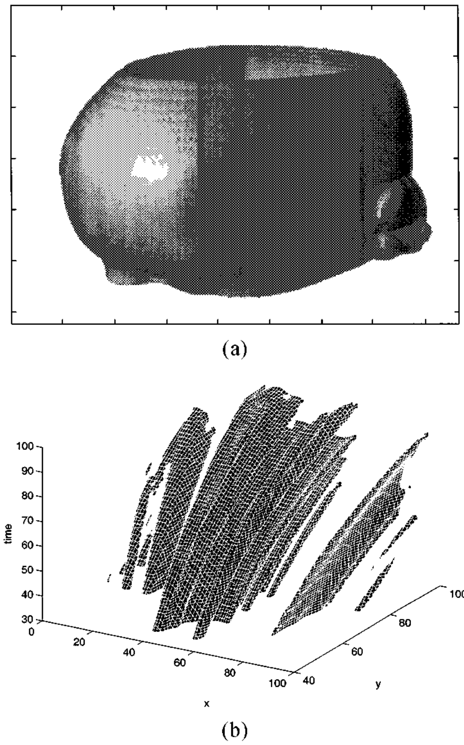
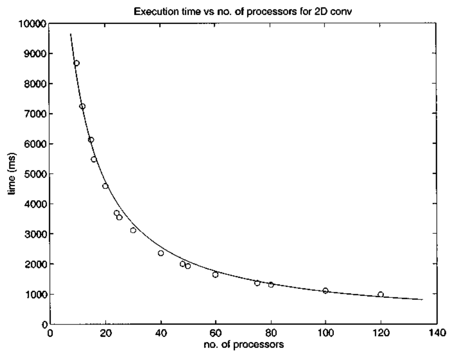
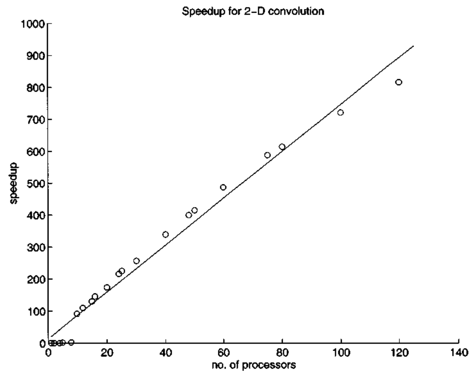


Figure 6. (a) The top-front view of the skull. (b) Spatiotemporal-surface detected from image evolution.

The Intel Paragon XP/S based on Intel's i860 XP processor is a distributed-memory MIMD machine. Its processing nodes are arranged in a two-dimensional rectangular grid. The system consists of three types of nodes: compute nodes, which are used for the execution of parallel programs; service nodes, which offer capabilities of a UNIX system, including compilers and program development tools; and I/O nodes, which are interface to mass storage and LANs. Paragon Mesh Routing Chips (MRCs), connected by high-speed channels, are the basis of the communication network, where nodes may be attached. There are two independent channels—one for each direction—between two neighboring nodes. The channels are 16 bits wide and have a theoretical bandwidth of 175 Mbps. The MRCs can route messages autonomously and are independent of the attached nodes. Communication is based on wormhole routing with a deterministic routing algorithm. All the three types of nodes are implemented by the same General Purpose (GP) node hardware. A 32-bit address bus and a 64-bit, 50 MHz (i.e. 400 Mbps) data bus connects all the components of the GP node's compute and network interface parts. The i860 XP is a Reduced Instruction Set Computer (RISC) processor with a clock speed of 50 MHz and a theoretical peak performance of 75 Mflops (64-bit arithmetic: 50 Mflops add, 50 Mflops multiply) and 100 Mflops (32-bit arithmetic: 50 Mflops add, 25 Mflops multiply). The interface of the GP node to the interconnection network performs such that message-passing is separated from computing.



(a)



(b)

Figure 7. Impact of no. of processors on the execution time.

We first show the results of the surface reconstruction process using a rendering package, and then we present the performance of our algorithms.

6.1. Visualization

Real MRI slices of a human skull. This data set consists of twenty-seven real MRI slices of a human skull, as shown in Figure 5. Each slice contains a lot of details (for instance the tissue of brain) inside the skull. The reconstructed skull is shown in Figure 6(a). It should be noticed that some of the details inside the skull are lost because the contrast of the tissue is too low.

Synthetic image sequences. This realistic data set is generated using a ray tracer called RayShade which is available in our system. Figure 6(b) is the resultant detected surface from a sequence of one hundred images taken by a camera undergoing forward motion. The surface indicates looming effect from the objects in the images.

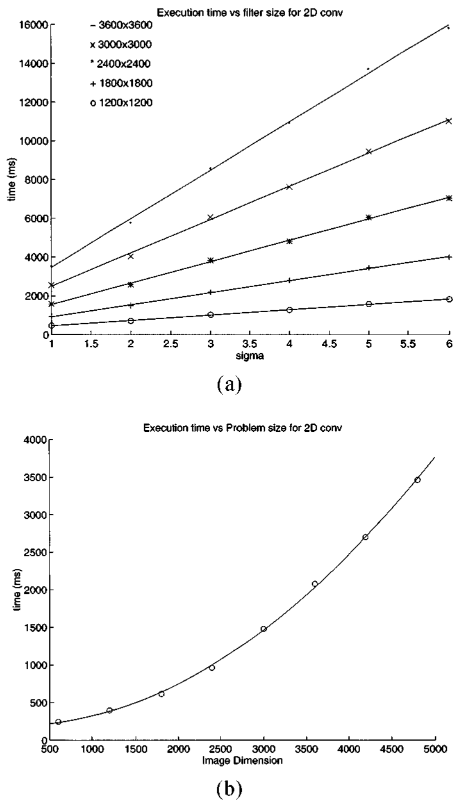


Figure 8. (a) The plot of execution time vs filter size. (b) Execution time for various image size.

6.2. Experiments

There are three variables which affect the performance of our parallel convolution algorithm. These variables are the number of processors used, the size of the filter, and the size of the image data. In this section, we investigate the impact of these parameters on the computation time. In addition, we verify the analysis given in the last section by experimental results.

6.2.1. Performance with 2D image

Number of processors. In this experiment, we study how the number of processors can affect the execution time. We set the size of the image to be 2400×2400 (integers) which is approximately equal to 16M bytes. The σ was set to 1 such that the filter size was 9. With increased number of processors used, the convolution problem was decomposed into more smaller subtasks which require less processing time for each processor.

The execution time (in msec) for convolution is shown in Figure 7(a). The time curve is nearly perfect hyperbolic which agrees with Equation 24 of execution time. The time points of number of processors equals 1 to 9 is not included in the figure

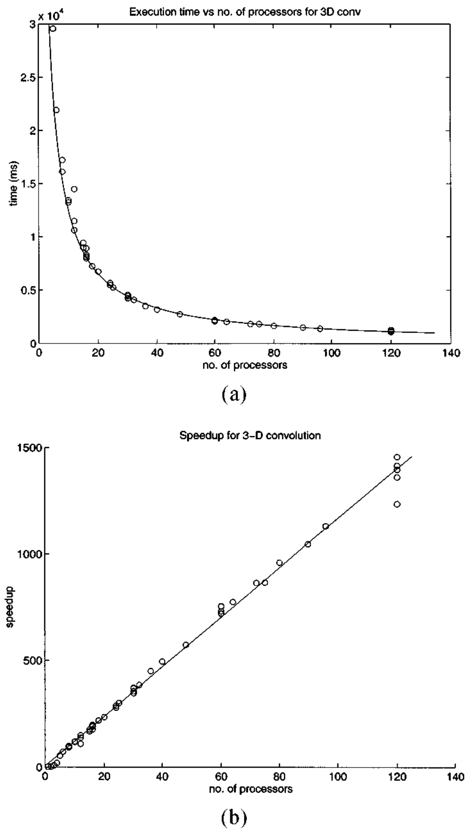


Figure 9. Impact of no. of processors on the execution time.

because their values are huge. Figure 7(b) shows the speedup of the program from which we see that super-linear speedup is achieved. This super-linear speedup is expected because heavy demand for memory accesses is required for small number of processors. As the size of the data caches is limited, there are excessive number of page faults when the image data and result data exceed the cache size. Therefore the execution times increase tremendously for the experiments using small number of processors.

We have implemented a program for measuring the lowest execution time achieved by a single processor. The lowest execution time achieved is 800000ms (around 13min) for the same setup of 2400 $\times$ 2400 image and filter size 9 while the execution time for the parallel problem with number of processors 10 and 20 is 8800ms and 4600ms, respectively.

Filter size. For the second experiment, we alter sigma of the Gaussian from 1 to 6. Recall that the filter size is equal to 9σ. Hence the filter size varies from 9 to 54. From Equation 24, the execution time is proportional to the filter size if both image size and the number of processors is fixed. In the experiment, the number of

Table 1. Execution time for various mesh configurations.

No. of processors	Mesh configuration	Execution time (ms)
12	12×1	14472
	6×2	57730
	4×3	53158
16	16×1	7979
	8×2	8125
	4×4	8313
30	30×1	4262
	15×2	4544
	10×3	4239
	6×5	4228
60	60×1	2182
	30×2	2188
	15×4	2151
	12×5	2088
	10×6	2082
120	120×1	1273
	60×2	1155
	30×4	1111
	15×8	1080

processors is fixed at 60. Five sets of results are obtained for different image size from 4M bytes to 36M bytes. The experimental results are presented in Figure 8(a). From this figure, we can see that the slope of the lines increases as the image becomes larger. In fact, a careful inspection reveals that the slope is proportional to the image size, $n_x n_y$, confirming Equation 24.

Image size. The third experiment studies the impact of the image size on the performance. In this experiment, we fixed the number of processors to 60 and sigma to 1 (i.e. filter size equals 9). Besides, the image dimensions, n_x and n_y, are assumed to be equal. From Figure 8(b), we see that the execution time is a quadratic function of the image dimension which confirms Equation 24.

6.2.2. 3D volume

Number of processors. The size of the image sequence for this experiment is $240 \times 240 \times 120$ (integers) which is more than 26 Mbytes and the filter size is fixed at 9 (i.e. $\sigma = 1$). A single processor takes 26 min to compute the 3D Gaussian convolution of this sequence. The plot of execution time (in msec) against number of processors is shown in Figure 9(a). As suggested by Equation 39 of execution time, the time curve is perfect hyperbolic. The timings with 1 to 5 processors are not included in the figure because their values are huge. Figure 9(b) shows the speedup of the program from which we see that super-linear speedup as again achieved because

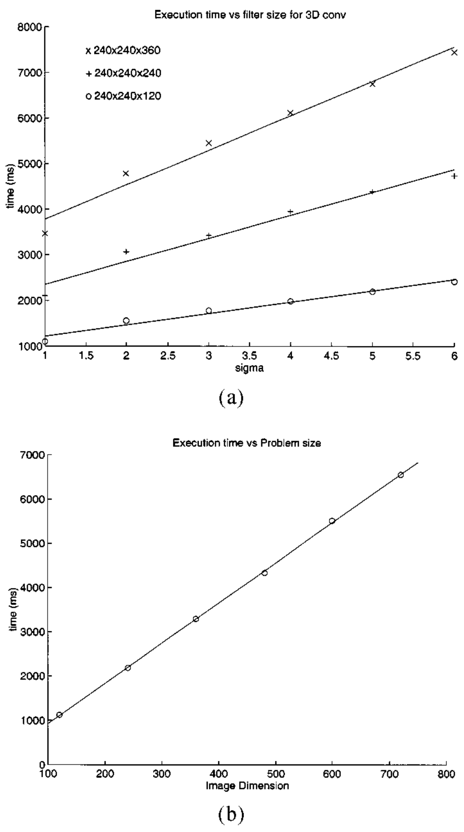


Figure 10. (a) The plot of 3D convolution time vs filter size. (b) The plot of execution time vs number of frames.

less cache missing occurred in experiments with large number of processors. We can also observe that the speedup was still linearly increasing with more than 100 processors.

Configuration of processors. We have shown by analytical result that the configuration of processors can also affect the performance of the program. To examine this effect, we used various mesh configuration. Table 1 presents the execution time of 3D convolution on various configurations. From this table, we can notice that the strip partitioning is preferable to the checkerboard partitioning for small number of processors such as 12 and 16. On the other hand, the checkerboard partitioning performs better for larger number of processors such as 30, 60 and 120.

Filter size. The influence of the filter size is significant for the computation time of 3D convolution. In the next experiment, the filter size was altered from 9 to 54 by adjusting σ from 1 to 6 and the number of processors was fixed at 120. The size of each image slice was 240×240. The result is shown in Figure 10(a) in which each line corresponds to a different number of slices in the sequence. As can be observed, the computation time increased linearly with the filter size, and the rate of increase (slope) was proportional to the number of slices in the sequence.

Image size. Obviously, for a fixed number of processors and filter size, the computation time would increase with an increase in the problem size. Equation 39 tells us that the computation time depends linearly on the number of slices of the sequence. Figure 10(b) shows the experimental results which verify the linear relationship. In this experiment, we used 120 processors and a filter of length 9.

7. Conclusions

In this paper, an efficient algorithm for the convolution of the three-dimensional image data volume has been proposed. The proposed algorithm uses the direction of the convolution to guide the partitioning of image data volume to minimize of communication overhead. The algorithm uses both the task and data parallelism strategies and yields substantial speedup.

An analytical study and extensive experimentation have revealed that the proposed parallel convolution algorithm scales very well with increasing both the problem size and the number of processors. In particular, it performs very well at convolution of large image data volume. The results achieved indicate the efficacy of the proposed convolution algorithm.

References

1. B. Arambepola. Architecture for high-order multidimensional convolution using polynomial transforms. *Electronics Letters*, 26(12):801–802, June 1990.
2. H. H. Baker. Building surfaces of evolution: The Weaving Wall. *International Journal of Computer Vision*, 3:51–71, 1989.
3. V. Berzins. Accuracy of Laplacian edge detectors. *Computer Vision, Graphics, and Image Processing*, 27:195–210, 1984.
4. M. Bomans, K. H. Hohne, U. Tiede, and M. Riemer. 3-D segmentation of MR-images of the head for 3-D display. *IEEE Transaction on Medical Imaging*, 9(2):177–183, June 1990.
5. J. Canny. A computational approach to edge detection. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.
6. J. H. Chang, O. Ibarra, T. C. Pong, and S. Sohn. Convolution on a pyramid computer. In *Proceedings of the International Conference on Parallel Processing*, pp. 780–782, 1987.
7. A. N. Choudhary. Performance of vision algorithms on multiple clusters in netra. *Proceedings of the Fourth Annual Parallel Processing Symposium*, April 1990.
8. H. E. Cline, W. E. Lorensen, R. Kikinis, and F. Jolesz. Three-dimensional segmentation of MR images of the head using probability and connectivity. *Journal of Computer Assisted Tomography*, 14(6):1037–1045, 1990.
9. S. G. Dykes, X. Zhang, Y. Zhou, and H. Yang. Communication and computation patterns of large scale image convolutions on parallel architectures. In *Proceedings of the Eighth International Parallel Processing Symposium*, pp. 926–931, 1994.
10. T. J. Kenney, *et al.* A serial/parallel color matrix, 2d convolution and 9-tap filter asic with a systems perspective. In *Proceedings of the Fifth Annual IEEE International ASIC Conference and Exhibit*, pp. 181–184, September 1992.

11. H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, October 1977.
12. H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics*, 23(3):79–88, July 1989.
13. C. D. Hansen and P. Hinker. Massively parallel isosurface extraction. In *Proceedings: Visualization '92*, pp. 77–83, October 1992.
14. D. D. Haule and A. S. Malowany. High-speed 2-d hardware convolution architecture based on vlsi systolic arrays. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, June 1989.
15. K. H. Hohne and W. A. Hanson. Interactive 3D segmentation of MRI and CT volumes using morphological operations. *Journal of Computer Assisted Tomography*, 16(2):285–294, 1992.
16. F. Jutand, N. Demassieux, and A. Artieri. A new vlsi architecture for large kernel real time convolution. In *International Conference on Acoustics, Speech and Signal Processing*, April 1990.
17. H. K. Kwan and T. S. Okullo-Oballa. 2-d systolic arrays for realization of 2-d convolution. *IEEE Transactions on Circuits and Systems*, 37(2), 1990.
18. T. Lindeberg. Detecting salient blob-like image structures and their scales with a scale-space primal sketch: a method for focus-of-attention. *International Journal of Computer Vision*, 11(3):283–318, December 1993.
19. W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(3):163–169, June 1987.
20. D. C. Marr and E. C. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London Series B*, 207:187–217, 1980.
21. O. Monga, S. Benayoun, and O. D. Faugeras. From partial derivatives of 3D density images to ridge lines. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 354–359, 1992.
22. V. S. Nalwa. *A Guided Tour of Computer Vision*, 1st ed., Addison-Wesley, Reading, Mass., 1993.
23. S. Ranka and S. Sahni. Convolution on simd mesh connected multicomputer. In *Proceedings of International Conference on Parallel Processing*, August 1988.
24. D. Raviv. Parallel algorithm for 3D surface reconstruction. *Proceedings of the SPIE—The International Society for Optical Engineering*, 1192:285–296, 1990.
25. O. Schwarzkopf. Computing convolutions on mesh-like structures. In *Proceedings of the Seventh International Parallel Processing Symposium*, pp. 695–699, 1993.
26. A. Silberschatz, J. L. Peterson, and P. Galvin. *Operating System Concepts*, 3rd ed. Addison-Wesley, Reading, Mass., 1991.
27. G. E. Sotak and K. L. Boyer. The Laplacian-of-Gaussian kernel: A formal analysis and design procedure for fast, accurate convolution and full-frame output. *Computer Vision, Graphics, and Image Processing*, 48:147–189, 1989.
28. S. Talele, T. Johnson, and P.E. Livadas. Surface reconstruction in parallel. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pp. 102–106, 1992.
29. F. Ulupinar and G. Medioni. Refining edges detected by a LoG operator. *Computer Vision, Graphics, and Image Processing*, 51:275–298, 1990.
30. A. P. Witkin. Scale-space filtering. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pp. 1019–1022, Karlsruhe, West Germany, August 1983.
31. T. S. Yoo and D. T. Chen. Interactive 3D medical visualization: A parallel approach to surface rendering 3D medical data. In *Proceedings of the Symposium for Computer Assisted Radiology*, pp. 100–105, June 1994.
32. X. Zhang and H. Deng. Distributed image edge detection methods and performance. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pp. 136–143, 1994.