ELSEVIER

Parallel Computing 21 (1995) 137–160

PARALLEL COMPUTING

Practical aspects and experiences

Parallel image processing applications on a network of workstations ☆

Chi-kin Lee, Mounir Hamdi *

Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Received 4 August 1993; revised 22 May 1994

Abstract

Concurrent computing on networks of distributed computers has gained tremendous attention and popularity in recent years. In this paper, we use this computing environment for the development of efficient parallel image convolution applications for grey-level images and binary images. Significant speedup was achieved using different image sizes, kernel sizes, and number of workstations. We also present a performance prediction model that agrees well with our experimental measurements and allows the highest speedup to be predicted from the knowledge of the ratio of the computation time to the communication time. The main limiting factor in our programming environment is the bandwidth of the network. Thus, it seems with emerging high-speed networks such as ATM networks, parallel computing on networks of distributed computers can be a very attractive alternative to traditional parallel computing on SIMD and MIMD multiprocessors in executing computationally intensive applications in general and image processing applications in particular.

Keywords: Image processing; Convolution algorithm; EXPRESS; Network of workstations; Distributed memory multiprocessor

1. Introduction

Parallel computing environments based on networks of computers have recently proven to be effective and economical platforms for high-performance computing in a number of disciplines [1,3,8,27]. Particularly in the areas of computer vision and computational science, where the demand for computing power is ever-in-

☆ This research work was supported in part by the Hong Kong Research Council under the grant RGC/HKUST 100/92E.

* Corresponding author. Email: hamdi@cs.ust.hk

SSDI 0167-8191(94)00068-9

creasing, network-based computing environments provide a very attractive alternative to traditional vector supercomputers and monolithic multiprocessors. From the economic point of view, network computing systems provide supercomputing power at minimal cost; an advantage highlighted by the fact that often, existing general purpose resources provide the computing platform, thereby requiring little or no additional investment. From the technical viewpoint, network computing systems provide equivalent and sometimes greater functionality, for a large number of application categories.

In this paper, we will discuss the results of using such a network environment for the implementation of fundamental parallel image processing tasks. Image processing is one of the most computationally intractable domains in computer vision and artificial intelligence research. Fortunately, image processing tasks lend themselves naturally to parallel processing. Basically, parallel image processing exploits the two fundamental modes of parallelism in image processing tasks: *image parallelism* and *function parallelism* [14,26]. Image parallelism is a kind of spatial parallelism, where the same operation is repeated on each pixel or subregion so that the image may be partitioned into a set of subimages which can be processed by multiple processing elements (PEs) for faster execution. On the other hand, function parallelism is temporal parallelism, where an image processing task consists of several levels of processing. Here an image processing function may be divided into subfunctions and utilize the scheme of pipelining. This paper is concerned with the first mode of parallelism, that is, image parallelism.

Most image processing tasks constitute the preprocessing step for numerous computer vision applications which is generally followed by object identification [28,29]. Both of these operations typically involve large amounts of computations but embody different computation paradigms [11,12,29]. Consequently, highly parallel, special purpose hardware is often used for vision applications, especially for image processing. Such special hardware are SIMD parallel computers [5,6,13,20]. We believe, however, that you can effectively specify image processing operations in a general environment, such as a network of workstations, and that you can use many of the recently developed parallel systems both for scientific *and* image processing applications. Hence, a network of workstations with its programming environment will provide users with transparent access to special image processing hardware. This is one more motivation that led us to consider the implementation of image processing tasks on a network of workstations. Further, we are aware of just few attempts for implementing image processing tasks on a network of workstations [7,15,17]. Thus, our experimental results would be a step further towards serving the purpose of evaluating the viability of a network of workstations as compared to special purpose SIMD computers or MIMD multiprocessors in executing image processing tasks in particular and computationally intensive tasks in general.

The image processing tasks that are considered in this paper involve mainly neighborhood operations which are the most common used operations in image processing [9,10,24]. Typically, the output of a neighborhood operator at a given pixel is a function of that pixel value and the values of some neighboring pixels

around it, often modified according to a kernel matrix. The kernel typically varies in size from 2×2 to 21×21 elements. There are numerous neighborhood operations such as averaging, thinning, shrinking, and template matching [18,24]. These operations are regular and repetitive which makes them very computationally intensive and, fortunately, ideal for parallel processing.

Since most of the image processing tasks which require neighborhood operations need the same computing power which is mainly dependent on the image size and the kernel size, and because it is impractical to examine all neighborhood operators, we concentrate in this paper on the implementation of one common neighborhood operation namely *convolution* with special emphasis on template matching. Template matching, which is an example of a convolution operation, is chosen for the study and the evaluation of the potential of using a network of workstations in the implementation of neighborhood image processing tasks (image parallelism) in our experiments. Template matching may be used as a simple method for filtering, edge detection, image registration, and object detection [4,5,22,23].

This paper is organized as follows. Section 2 gives an overview about convolution algorithms for both grey-level images and binary images. Section 3 presents our parallel image convolution computational model, and discusses the significance of different image partitioning methods. Section 4 presents a performance prediction model that estimates the performance of our computational model. Section 5 presents our experimental results for implementing the parallel image convolution tasks on a network of workstations. Then, we compare these results with the expected theoretical results. Finally, Section 6 concludes the paper.

2. Convolution algorithms

Convolution is the most general image processing operation. It is a fundamental low level operation in image processing and computer vision applications. The applications of convolution range from linear image processing, to edge detection [4], morphological image processing [9], feature extraction, template matching [23], and regularization theory [24]. There are mainly two types of convolutions found in image processing applications, a 1-D convolution and a 2-D convolution. In this paper, we address 2-D convolution because it is more computationally intensive, and because a 1-D convolution is simply a special case of 2-D convolution. A 2-D convolution operation is defined in the following way. For an $N \times N$ image matrix, $I[0..N-1, 0..N-1]$, and a kernel also known as mask or template with size $M \times M$, $T[-\lfloor M/2 \rfloor..\lfloor M/2 \rfloor, -\lfloor M/2 \rfloor..\lfloor M/2 \rfloor]$, the convolution output is an $N \times N$ image matrix $C[0..N-1, 0..N\text{-}1]$ where each matrix entry is defined as:

$$C[i, j] = \sum_{u=-\lfloor M/2 \rfloor}^{\lfloor M/2 \rfloor} \sum_{v=-\lfloor M/2 \rfloor}^{\lfloor M/2 \rfloor} I[i+u, j+v] \times T[u, v] \quad 0 \le i, j < N. \tag{1}$$

Each element of the image matrix corresponds to a value of a single image pixel, and each element of the kernel matrix corresponds to a kernel weight. An image pixel is represented by an 8-bit (1 byte) integer for grey-level images and by a single bit (0 or 1) for binary images. The size of the image, $N \times N$, depends on the resolution used, and the size of the kernel, $M \times M$, depends on the application. As can be seen from Eq. 1 above, the value of $C[i, j]$ depends only on the values of a small number of its neighbors. The number of neighboring pixels is directly related to the kernel size and is equal to M^2. Thus, to determine a single convolution value using Eq. 1, we have to perform $O(M^2)$ multiplications. For an $N \times N$ image matrix, the time complexity would be $O(N^2M^2)$ to perform the image convolution using a sequential machine. This can be a very time-consuming task when N is large.

The implementation of convolution operations on binary images is very similar to that implemented on grey-level images. In this case, the image pixel values and the kernel weights are 1 bit elements (0 or 1). To perform convolution on a binary image, we use a variant of the above Eq. 1 as follows [19]:

$$C[i, j] = \sum_{u=-\lfloor M/2 \rfloor}^{\lfloor M/2 \rfloor} \sum_{v=-\lfloor M/2 \rfloor}^{\lfloor M/2 \rfloor} (\neg I[i+u, j+v] \oplus T[u, v]) \wedge M[u, v])$$

$$0 \le i, j < N. \tag{2}$$

where $\oplus$ stands for XOR, I is the binary image matrix ($I[i, j] \in \{0, 1\}$), T is the kernel matrix which specifies the value of the do-care pixels (0 or 1), and M specifies whether a pixel in this kernel is a do-care or a don't-care term [9]. Like the convolution for grey-level images, the value of each pixel of the binary image depends only on a small number of pixels which is directly related to the size of the binary kernel used.

3. A network of workstations environment

Our computing environment consists of a number of SUN workstations connected by an Ethernet network using EXPRESS. Thus, from a parallel point of view, each SUN workstation (node) is considered as a single PE, and the whole network looks like a distributed memory MIMD computer. EXPRESS, developed by Parasoft [21], is a programming environment for writing parallel programs for MIMD multiprocessors. It is simply a software layer which executes above the individual operating systems of the autonomous SUN machines that are networked. Further, EXPRESS, is a portable parallel programming environment, and is available on most commercial parallel machines. There are other similar parallel programming environments such as PVM and P4 [8,25] which have been used by many researchers in the implementation of parallel programs. One major difference between a network of workstations and a parallel machine both running EXPRESS is that the Ethernet network connecting the workstations has a much lower bandwidth and a bigger latency time than those of an interconnection

network (e.g. hypercube, mesh) of a parallel machine. This makes the design of a parallel application for a network of workstations more difficult because the careful consideration of the communication overhead is much more crucial.

3.1 Parallel program model

EXPRESS allows the development of parallel programs in the following two different styles:

(1) The conventional *Host-Node* style in which the control part of the program appears in the host program and the computation intensive part in the node programs.
(2) The *Cubix* style in which only the node programs are written, and they interface to the outside world through graphics and text server.

The *Cubix* style is easier to use than the *Host-Node* style since the programmer does not have to worry about writing a host program. On the other hand, the *Host-Node* style offers more flexibility and better scalability. Our parallel image processing convolution software has been designed using the *Host-Node* style. The host program is used to partition a given image into a number of subimages and distribute them to the nodes, and to collect the resultant convolved subimages from the nodes. The node programs are mainly used to carry out the convolution computation on their respective subimages. The convolution operation can be easily replaced by other neighborhood operations to perform other image processing functions. Hence, our *Host-Node* program is general enough to be used in a wide range of image processing applications.

The efficiency of exploiting image parallelism on a network of workstations is determined mainly by the communication overhead. This overhead is mainly due to the distribution of subimages to a set of workstations, data exchange during the convolution computation, and the collection of local results. Hence, this communication overhead is directly related to the number of subimages. Therefore, it is important to determine the best image partitioning in terms of the number of workstations employed. When we partition the image into a number of workstations to perform convolution, the value of each pixel depends on the values of some neighboring pixels. However, some of these neighboring pixels might be belonging to the subimage allocated to a different workstation. This is the case for all pixels on the boundary of the subimages allocated to different workstations. This is illustrated in Fig. 1, where two subimages are separated by a vertical boundary, and the thick square is the kernel region. In this case, when a workstation needs to calculate the convolution of the pixels on the border of subimage 1, the kernel will require the values of some pixels in subimage 2 which are residing in a different workstation. Consequently, an extra communication overhead would be incurred besides distributing the original $N \times N$ image pixels to the workstations. The amount of the extra communication overhead depends on the length of the boundaries of the subimages and the kernel size. This implies that the bigger the number of workstations employed to share the workload, the bigger the number of subimages generated, and hence a higher extra communication overhead is introduced, especially for large kernel sizes.

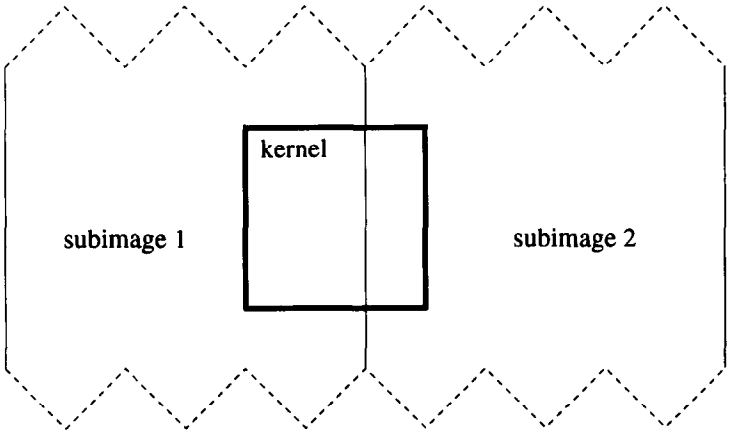

Fig. 1. Pixels on the boundary of a subimage need neighboring pixels from another subimage for the computation of their convolution values.

There are two methods used to handle this extra communication [22]:

(1) Overlap mapping: each workstation obtains its subimage and the boundary pixels (belonging to different subimages) needed to find all the convolution values of its boundary pixels. This is illustrated in Fig. 2 where a workstation gets all the data bounded by the dotted square instead of the thick solid square.
(2) Non-overlap mapping: each workstation obtains its subimage without the pixels needed by its own boundary pixels during the convolution and belonging to a different subimage. That is each node only obtains the data bounded by the thick solid square in Fig. 2. However, when it requires some pixel values that are residing on different subimages, it has to communicate with its neighboring workstations to obtain them.

By examining both methods, we can see that the second method requires less data to be transferred from the host to the nodes. On the other hand, the first method requires no communication between the workstations while performing convolution, unlike the second method. Further, the amount of extra data that have to be sent between the host and the nodes is equal to the amount of data to be communicated between the workstations to perform the convolution using the second method. For this reason, we adopted the first method in our implementation. Moreover, using the first method, all workstations work independently from each other during their computation of the convolution of their subimages, and they don't need to recognize their neighbors. This is similar to the Single Program Multiple Data (SPMD) mode [2,16].

3.2. Image partitioning method

Since the extra communication overhead introduced in the previous section is directly proportional to the number of pixels on the boundary of the subimages,

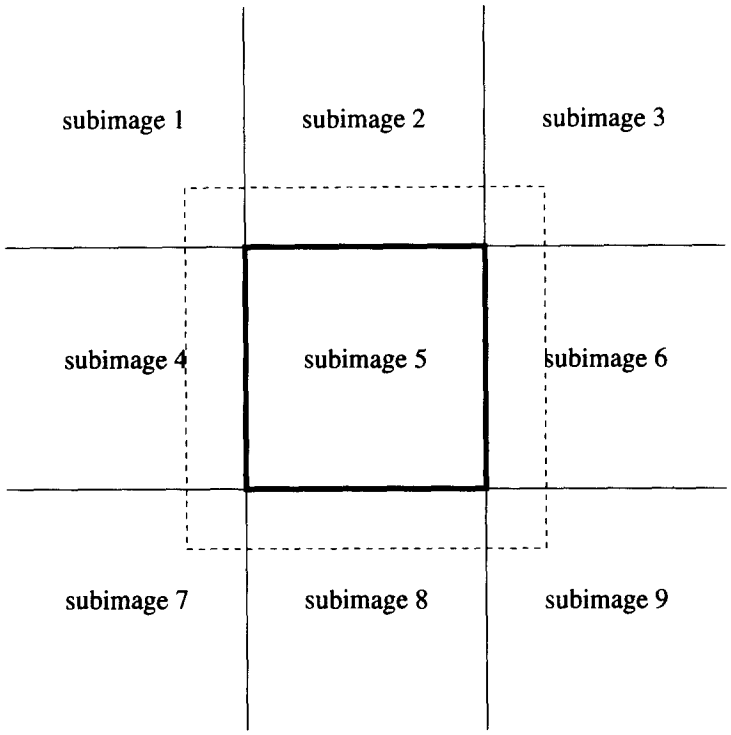

Fig. 2. The subimage size using the overlap mapping method.

different partition methods of the whole image will produce different amounts of communication overhead. Actually, there are numerous ways of partitioning a given image into a given number of workstations. However, in order to maintain and manipulate the image matrix easily, and because certain partitioning schemes (e.g. diagonal partitioning) are hard to analyze, only vertical and horizontal partitioning directions are considered. Without loss of generality, we assume the original image to be a square image with width L (number of pixels). Now, we examine two image partitioning methods denoted *row partition* and *cross partition* respectively.

(1) *Row partition*

Using the row partition method, the image is divided horizontally into n equal subimages for n workstations. Each workstation performs convolution on subimages of size L^2/n pixels. The advantage of this partitioning method is that it can divide a given image into any number of subimages of comparable sizes. This implies that all workstations involved in the computing of the convolution have comparable workload [2]. However, the total number of boundary pixels of the subimages, $L_{boundary}$, is big and is given by:

$$L_{boundary} = (n-1)L. \tag{3}$$

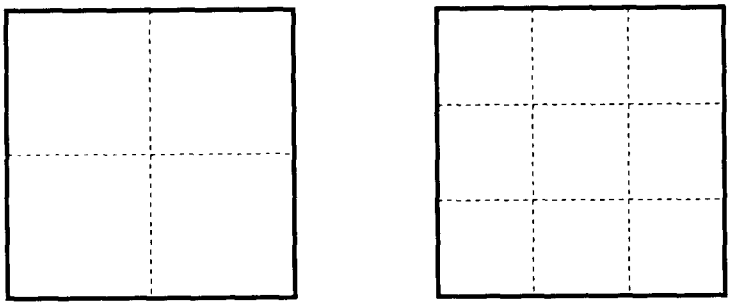

Fig. 3. The partitioning of an image into 4 and 9 subimages respectively using the cross partition method.

(2) *Cross partition*

Using the cross partition method, the image is divided evenly using both the horizontal and vertical dimensions as shown in Fig. 3 for any number of workstations, $n = i^2$ where i is any integer > 0. The number of boundary pixels of all subimages is optimum, that is minimum, for any square image size, and is given by:

$$L_{boundary} = 2(\sqrt{n} - 1)L. \tag{4}$$

However, the cross partition method places a restriction on the number of workstations used. Essentially the number of workstations used must be a square number, that is, 4, 9, 16, 25...etc. Hence, using the cross partition method, we may not be able to use it for the allocation of all available computing resources.

In order to remedy the disadvantages of both methods, we propose an alternate partition method, denoted *heuristic partition*. The heuristic partition method works as follows given an image and n workstations.

Step 1. If $n = 1$, stop.

Step 2. If n is even, we divide the image into two equal subimages, A and B, through the minimum number of boundary pixels (either vertical or horizontal pixels).

Step 3. Else, we divide the image into two subimages, A and B, through the minimum number of boundary pixels such that the ratio of the number of pixels of subimage A to that of subimage B is equal to $\lfloor n/2 \rfloor : \lfloor n/2 \rfloor + 1$.

Step 4. Repeat step 1 for subimages A and B respectively.

Fig. 4 shows the results of the heuristic partition method when $n = 5$ and $n = 7$. For any number of workstations, n, the number of pixels of all subimages are equal, but the subimages, may have different shapes.

As mentioned before, the number of boundary pixels of the subimages is directly related to the way the image is partitioned *and* the size of the kernel convolution. Suppose the kernel size is $M \times M$, then the number of boundary

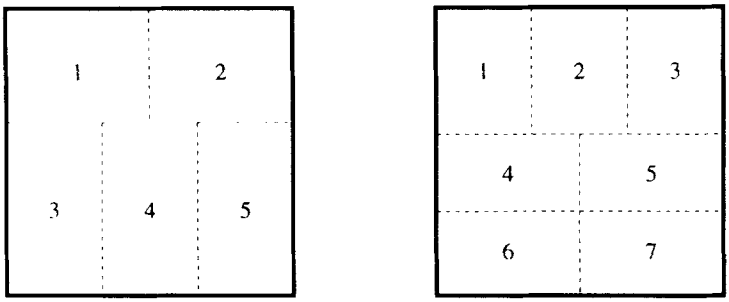

Fig. 4. The partitioning of an image into 5 and 7 subimages respectively using the heuristic partition method.

pixels is $C_B = L_{boundary} \times (M-1)$ + a small number corresponding to the number of pixels on the corners of the subimages' boundaries. Usually, the number of pixels on the corners is small relative to the number of pixels on the boundaries especially for *coarse grain* parallel computation where the number of subimages is not very large which is the case when using a network of workstations. Hence, the number of corner pixels can be negligible compared to the total number of boundary pixels. Thus, distributing the original image from the host to the nodes involves sending L^2 pixels. The percentage of the number of extra (boundary) pixels with respect to the total number of pixels in the image is approximately C_B/L^2, denoted *boundary overhead*. For example, if a 1024×1024 image matrix is divided into 9 subimages and the kernel size is 11×11, then:

By using the row partition method, the *boundary overhead* $= 8L(M-1)/L^2 \approx 7.8\%$.

By using the cross partition method, the *boundary overhead* $= 4L(M-1)/L^2 \approx 3.9\%$.

By using the heuristic partition, the *boundary overhead* $= 4.24L(M-1)/L^2 \approx 4.1\%$.

The above estimates show that the *boundary overhead* is small compared to the overhead resulting from sending all pixels of the image, especially when the kernel size is much smaller than the image size which is the case in most image processing convolutions. Consequently, the partition method is not very crucial in the overall performance of the parallel image processing convolution. In our experimental results, we used the heuristic partition method because it represents a compromise between the row partition method and the cross partition method.

4. Performance prediction model

Before presenting the experimental results, it is desirable and advantageous to develop an analytical model to describe the behavior and the performance of the parallel image convolution program. The analytical model is not only used to verify the experimental results but also to predict the speedup tendency. The speedup

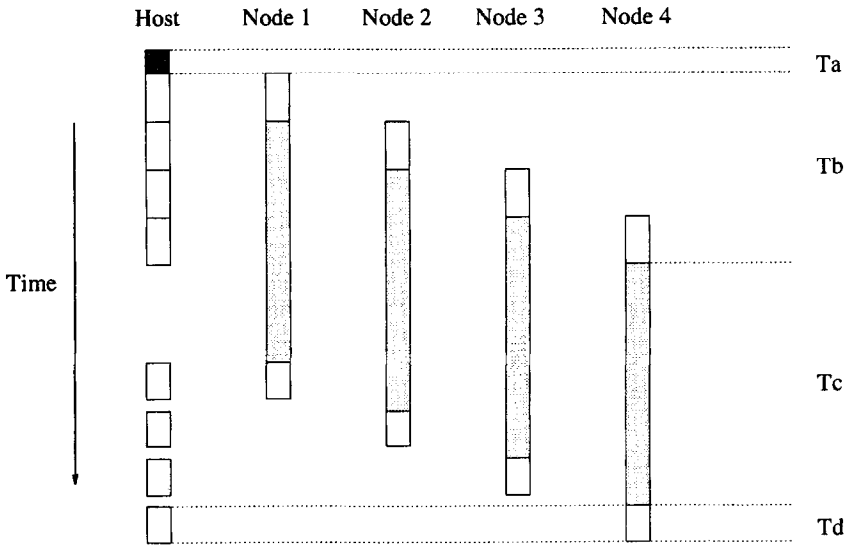

Fig. 5. The activities of all processes involved in the parallel image convolution program.

tendency can be used to project the maximum number of workstations that can be used while preserving a positive speedup. Using a large number of workstations can be sometimes counter productive, that is, the speedup that they achieve is lower than that of a smaller number of workstations. The main reasons for that are the low bandwidth of the Ethernet and the high granularity of certain applications. Hence, this analytical model can save us a lot time running different experiments to find out the number of workstations that achieve the highest performance for a given image processing convolution application.

Based on the *Host-Node* style of our parallel image convolution program, Fig. 5 describes the activities of all processes involved. The dark grey box represents the host's setup time. The light grey boxes represent the convolution computation time on the different workstations, and the white boxes represent the communication time for sending the image data between the host and the nodes. Suppose there are 4 active nodes (workstations), the host program sends 4 subimages and the kernel matrix to all the nodes one after the other. Each node starts the convolution computation of its subimage as soon as it receives the subimage and the kernel matrix. Hence, there is some overlapping between sending/receiving the subimages data and the convolution computation on the nodes as shown in Fig. 5.

Using this parallel computational model, the execution time of the parallel program can be broken up into four terms:

- *Ta*: the host setup time which is mainly due to performing the image partition.
- *Tb*: the communication time for sending all subimages' data and the convolution kernel data to all workstations involved.
- *Tc*: the average computation time of the convolution on a single workstation.
- *Td*: the communication time for receiving the partial results of the image convolution from one workstation.

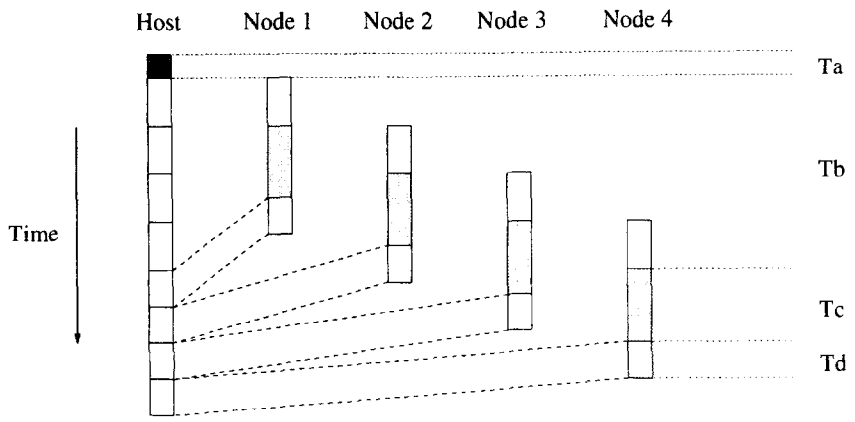

Fig. 6. The activities of all processes involved in the parallel image convolution program when the computation time is very small.

Hence, the execution time of the parallel program, using n workstations, is the summation of the 4 terms and is given by:

$$T_{(n)} = Ta + Tb + Tc + Td. \tag{5}$$

We can see that as the number of workstations increases in the execution of the image convolution, Tc and Td will be smaller but Tb may increase. In case the computation time is too small, some nodes may start sending back their results before the host has finished distributing all subimages. This is illustrated in Fig. 6. Hence, the execution time of the parallel image convolution in this case which rarely happens is bounded below by the communication time which is given by:

$$T'_{(n)} = Ta + Tb + n \times Td. \tag{6}$$

4.1. Computation and communication times

In order to formulate the equations that give us the expected speedup curves of our parallel program, we have to formulate the equations for the computation time and communication time which make up the execution time of the whole program. The computation time is proportional to the number of multiplications of two pixels in grey-level images and the number of logical 'XOR' and 'AND' operations in binary images. Let us denote each iteration in Eqs. 1 and 2 as one *computation step*. Thus, the convolution of an $N \times N$ image matrix using an $M \times M$ kernel requires N^2M^2 computation steps. Let γ be the time to perform one computation step. Then, the computation time, T_{comp}, is given by:

$$T_{comp} = N^2M^2\gamma. \tag{7}$$

If the workload is divided evenly over n nodes, then each node's processing time is given by:

$$T_{comp} = \frac{N^2M^2}{n}\gamma. \tag{8}$$

The total communication time of the parallel program is the summation of two components: *latency time* and *transmission time*. The latency time, α, is a fixed startup overhead time needed to prepare sending a message from one workstation to the other. The transmission time is proportional to the size of image data. Let β be the incremental transmission time per byte. Note that it is usually the case that $\alpha \gg \beta$. Then the communication time, T_{comm}, to send P bytes of data (messages) is defined as:

$$T_{comm} = \alpha + P\beta. \tag{9}$$

Since a single message sent between two workstations cannot be infinitely long, it is restricted to be K bytes long to simplify the buffer management of the workstations, and because of the Ethernet protocol. Therefore, the actual communication time to send P bytes of data is given by:

$$T_{comm} = \lceil P/K \rceil \alpha + P\beta. \tag{10}$$

i.e. P bytes of data will be packed into $\lceil P/K \rceil$ messages to be transmitted one after the other. For the binary image case, we only use 1 bit to represent a pixel value (0 or 1). To send P pixels from the host to the node, the volume of data communication is just $\lceil P/8 \rceil$ as compared to the volume of data communication for grey-level images. Therefore, the communication time is smaller when using binary images.

4.2 Parallel performance

If the values of α, β and γ are known, then the execution time of the parallel image convolution program can be easily estimated as will be shown below. To formulate the equations of the parallel image convolution execution time for an $N \times N$ image matrix, an $M \times M$ kernel matrix, and n workstations, we need to determine the value of each of the following terms:

Ta: It is mainly the time spent in partitioning the image using the heuristic partition method. Compared with the communication time and the convolution time, it is negligible.

Tb: It includes the communication time to send the kernel matrix and all the subimage matrices to all workstations involved in the processing of the image convolution. The size of an $M \times M$ kernel is M^2 bytes which can be packed into $\lceil M^2/K \rceil$ messages. Hence, the latency time of sending these messages is $\lceil M^2/K \rceil \alpha$, and the transmission time is $M^2\beta$. Similarly, the size of a subimage is $(N^2 + C_B)/n$ bytes approximately where C_B bytes is the size of the extra pixels to be sent because of our partition scheme described earlier. Hence, the communication time for the kernel data and the subimage data are given respectively by:

$$\left\lceil \frac{M^2}{K} \right\rceil \alpha + M^2\beta, \tag{11}$$

and

$$\left\lceil \frac{N^2 + C_B}{nK} \right\rceil \alpha + \frac{N^2 + C_B}{n} \beta. \tag{12}$$

Consequently, the communication time, Tb, is approximated by:

$$n\left(\left\lceil \frac{M^2}{K} \right\rceil \alpha + M^2\beta + \left\lceil \frac{N^2 + C_B}{nK} \right\rceil \alpha + \frac{N^2 + C_B}{n} \beta \right). \tag{13}$$

Tc: Since each workstation has to compute the convolution of only its subimage with size N^2/n pixel values, the convolution time on a single workstation is given by:

$$\frac{N^2 M^2}{n} \gamma. \tag{14}$$

Td: The host has to receive n partial results resulting from the convolution carried on the subimages by n workstations concurrently. Each workstation sends back N^2/n pixel values. Therefore, the communication time to receive these partial results from one workstation is given by:

$$\left\lceil \frac{N^2}{nK} \right\rceil \alpha + \frac{N^2}{n} \beta. \tag{15}$$

The execution time of the parallel image convolution program, $T_{(n)}$, simply the summation of all the terms calculated above (refer to Eq. 5) and is given by:

$$T_{(n)} = n\left(\left\lceil \frac{M^2}{K} \right\rceil \alpha + M^2\beta + \left\lceil \frac{N^2 + C_B}{nK} \right\rceil \alpha + \frac{N^2 + C_B}{n} \beta \right) + \frac{N^2 M^2}{n} \gamma + \left\lceil \frac{N^2}{nK} \right\rceil \alpha + \frac{N^2}{n} \beta. \tag{16}$$

When n increases, the computation time on each node, $N^2M^2\gamma/n$, decreases. However, if the computation time Tc, is too small, Eq. 5 cannot be used to find the execution time of the parallel image convolution program as indicated graphically by Fig. 6. We rather use the lower bound execution time, $T'_{(n)}$, given by Eq. 6:

$$T'_{(n)} = \left(\left\lceil \frac{M^2}{K} \right\rceil \alpha + M^2\beta + \left\lceil \frac{N^2 + C_B}{nK} \right\rceil \alpha + \frac{N^2 + C_B}{n} \beta + \left\lceil \frac{N^2}{nK} \right\rceil \alpha + \frac{N^2}{n} \beta \right) n. \tag{17}$$

By equating Eqs. 16 to 17 (i.e. $T_{(n)} = T'_{(n)}$), we find the intersecting point which

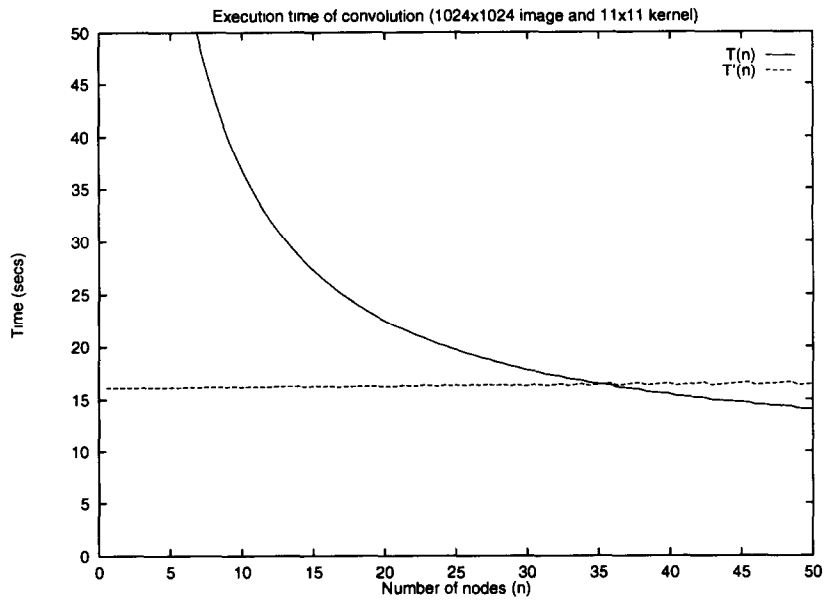

Fig. 7. Intersecting point of T(n) (Eq. 16) and T'(n) (Eq. 17).

represents the number of workstations that achieve the minimum execution time of the parallel image processing convolution in the following way:

$$\frac{n-1}{n}N^2\left(\frac{\alpha}{K}+\beta\right)=\frac{N^2M^2\gamma}{n}$$

$$(n-1)\left(\frac{\alpha}{K}+\beta\right)=M^2\gamma$$

$$n=\frac{M^2\gamma}{\alpha/K+\beta}+1. \tag{18}$$

Fig. 7 shows the intersecting point of Eqs. 16 and 17 when $N=1024$, $M=11$, and for some fixed values of α, β and γ. The determination of the values of α, β and γ is presented in the Appendix of this paper.

The exact speedup, S, of our parallel image convolution program is given by:

$$S=\frac{T_s}{T_{(n)}}. \tag{19}$$

where T_s is the execution time of a sequential image convolution program on a single workstation, and $T_{(n)}$ is the execution time of the parallel convolution program on n workstations as given by Eq. 16.

Now, let us calculate the upper bound of the above speedup, S_{limit}. That would give us an upper limit on the scalability of a network of workstations in image convolution operations. In this case, we assume that $N^2 \gg M^2$ which is realistic

for most practical image processing convolutions. Thus, the distribution and collection of an $N \times N$ image matrix dominates the whole execution time where the computation time is small as illustrated in Fig. 6. In this case, $T_{(n)} \approx 2N^2(\alpha/K + \beta)$. Therefore, under the above assumptions, the speedup limit can be approximated by:

$$S_{limit} \approx \frac{N^2M^2\gamma}{2N^2(\alpha/K+\beta)}$$

$$\approx \frac{M^2\gamma}{2(\alpha/K+\beta)}. \tag{20}$$

Basically, the speedup limit depends on the ratio of the computation time to the communication time. The communication time is dependent on the amount of data to be transmitted between the host and the nodes and on the Ethernet network bandwidth. One way to obtain a high speedup besides minimizing the amount of data transmitted between the host and the nodes is to use other high speed networks such as FDDI or ATM networks [3,27].

5. Experimental results

In order to fully investigate the performance and the suitability of our parallel image processing convolution on a network of workstations, we conducted extensive experiments by varying the essential parameters, the kernel size ($M \times M$), the image size ($N \times N$), and the number of workstations (n). We essentially wanted to answer the following questions: *how the execution time of the parallel image convolution varies with respect to the essential parameters, n, M, and N.* It would also have been interesting if we had the opportunity to experiment with different networks (e.g. ATM, FDDI) and see their impact on the execution time of the parallel image convolution as compared to the Ethernet. This is another avenue for further research in this topic. The execution times are recorded for the parallel programs (the host program and the node program) and the sequential program as a function of M, N and n. Then, we compare the execution times of the sequential program and the parallel program in order to determine the speedup ratio as given by Eq. 19. We have chosen pattern matching as an example of a convolution operation. However, these results may well be for other neighborhood operations as most of them are similar.

All the experiments have been carried out using SUN Sparc IPC workstations connected by an Ethernet network. All the software has been written under the EXPRESS parallel programming environment. An image pixel is set to be one byte long for grey-level images and 1 bit long for binary images. Under this environment, the latency time, α, and the incremental transmission time per byte, β, are found to be 2.115 *ms* and 2.074 $\mu s/byte$ respectively. The computation factor, γ, is determined to be 2.20 μs for grey-level images (see Appendix for details).

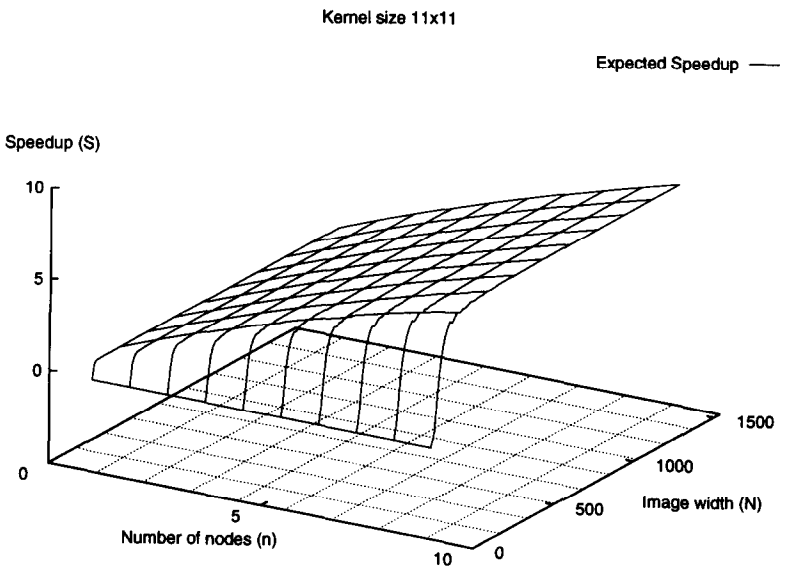

Fig. 8. The expected speedup as a function of the image size and number of nodes.

If we substitute these experimental values of α, β, and γ into Eq. 19 of our performance prediction model, we can get values of the expected speedup for various combinations of image sizes and number of workstations where the kernel size is equal to 11×11 as shown in Fig. 8. As can be seen from this figure, the speedup gets higher as the number of workstations gets higher. When the image size is small, the speedup drops quickly as the communication overhead wipes out the gain from parallelism.

If we fix the image size to be 1024×1024, and then we vary the kernel size, we can see the effect of the kernel size on the overall speedup. This expected speedup is illustrated in Fig. 9. The speedup gets higher as the kernel size gets bigger because more computation is needed to perform the convolution. Since the computation time is $O(N^2M^2)$ and the communication time is $O(N^2 + M^2)$, using large kernel sizes will increase the ratio of the computation time to communication time and causes a higher speedup.

To validate the expected speedups, we performed various experiments and calculated the exact speedup achieved using our network of workstations. The results presented here are for performing convolution on grey-level images. Using pattern matching as an example of a convolution operation, some execution time results are placed in the Appendix for reference. Fig. 10 shows the speedup curves as a function of the image size and the number of workstations where the kernel size is 11×11. Compared with Fig. 8, they both have the same speedup tendencies. As can be seen from the figure, the speedup is small when the image size is small (e.g. 32×32). This is because the communication time is too high relative to the computation time. This suggests that for small images, it is better to perform

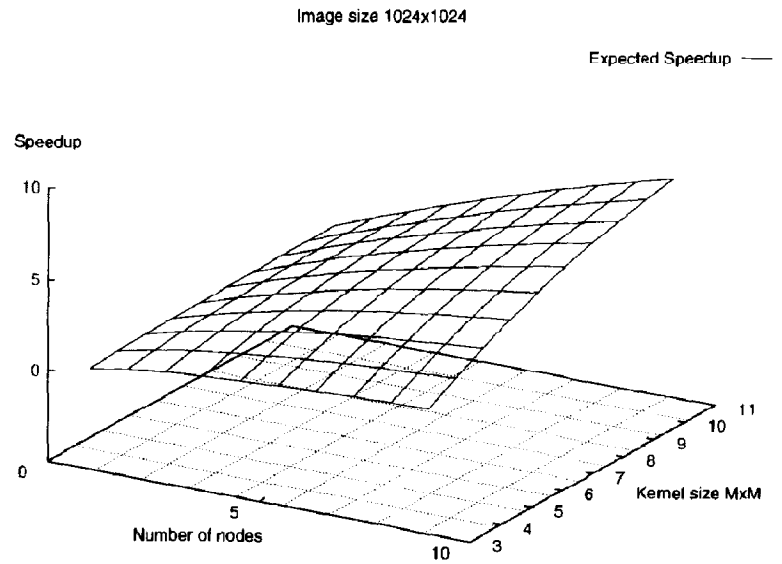

Fig. 9. The expected speedup as a function of the kernel size and the number of nodes.

convolution on sequential machines. This is realistic, since small images do not require a lot of computing power.

Fig. 11 shows the speedup curves as a function of the kernel size and the

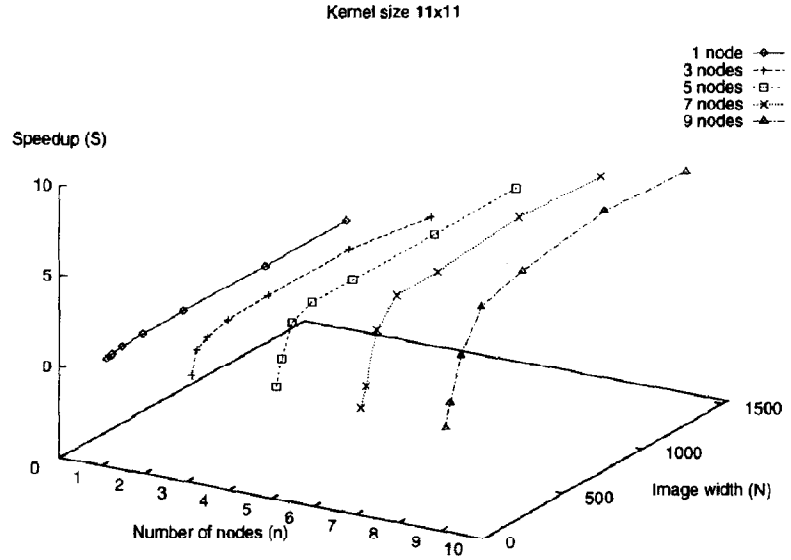

Fig. 10. The experimental speedup as a function of the image size and number of nodes for an 11×11 kernel.

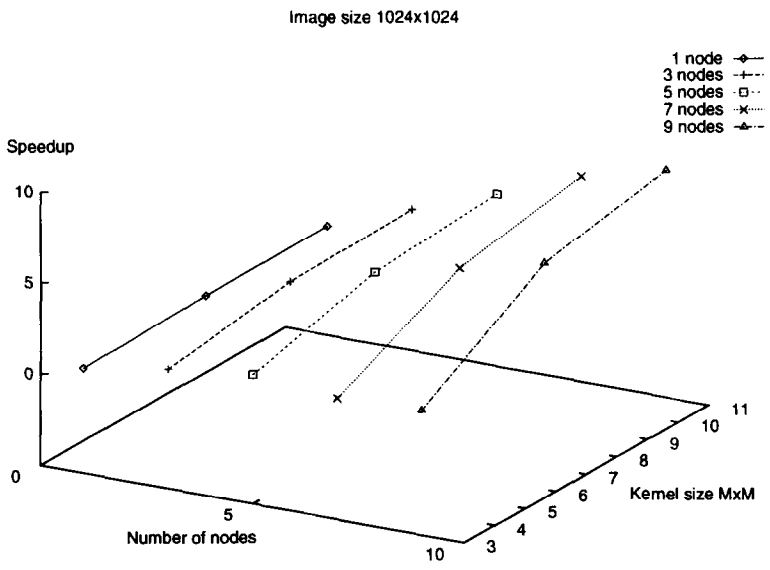

Fig. 11. The experimental speedup as a function of the kernel size and number of nodes for an 1024×1024 image.

number of nodes for an 1024×1024 image. The larger the kernel size is, the higher the speedup is. When the kernel size is large, more computations have to be performed at each pixel in the convolution step. However, the communication time is more or less the same in all cases for a given image size since the effect of the kernel size on the communication time is not very significant. Therefore, the ratio of the computation time to communication time is mainly dependent on the kernel size in this case, and thus increases as the kernel size increases. This implies that the percentage of execution time due to communication time decreases and so the speedup increases. This also suggests that no matter which neighboring function we are performing (e.g. shrinking, smoothing, etc.), we can obtain a high speedup as long as the ratio of the computation time to the communication time is high.

Fig. 12 illustrates a different interpretation of the data in Fig. 11 by joining the corresponding points along the x-axis. It shows that the larger the kernel size is, the steeper the speedup curve is. However, the curve corresponding to the 3×3 kernel does not go up when more nodes are used, meaning that it is counter productive to use more than 7 workstations in this case. Using Eq. 18 of our performance prediction model, we find n to be 6 in this case which is close to the peak of our experimental curve. This is an affirmation of the accuracy of our performance prediction model presented in Section 4. In Section 3.2, it is noted that more communication time will be introduced when an image is partitioned into a big number of subimages. Thus, when more nodes are involved, more partitioning takes place and the communication time will also increase, and so the speedup will decrease. This is more significant in the 3×3 kernel curve since its

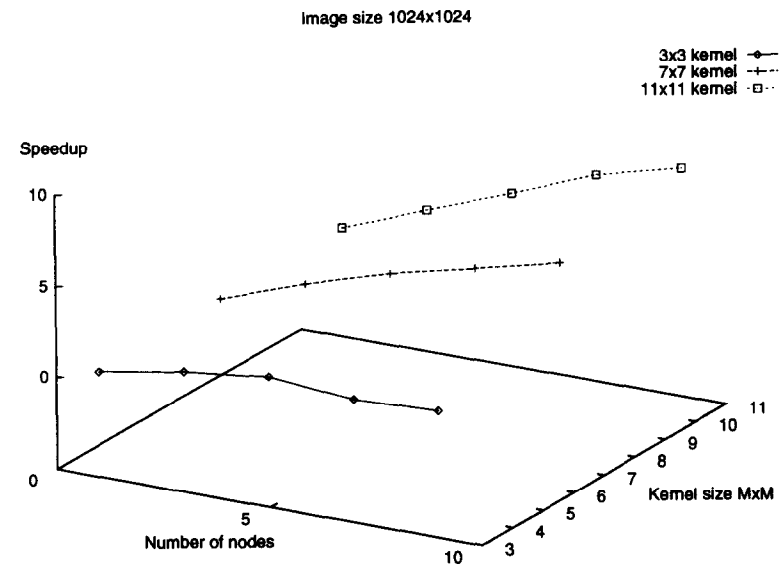

Fig. 12. The experimental speedup as a function of the kernel size and number of nodes for an 1024×1024 image.

computation time is small and so the ratio of the computation time to communication time decreases faster even for the addition of the same amount of additional communication time.

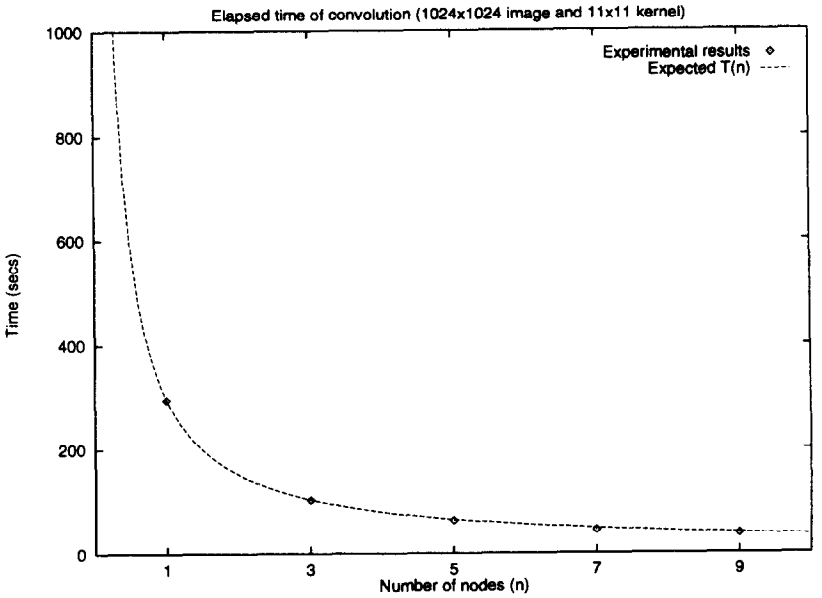

Fig. 13. Comparison between the experimental execution time and the predicted execution time.

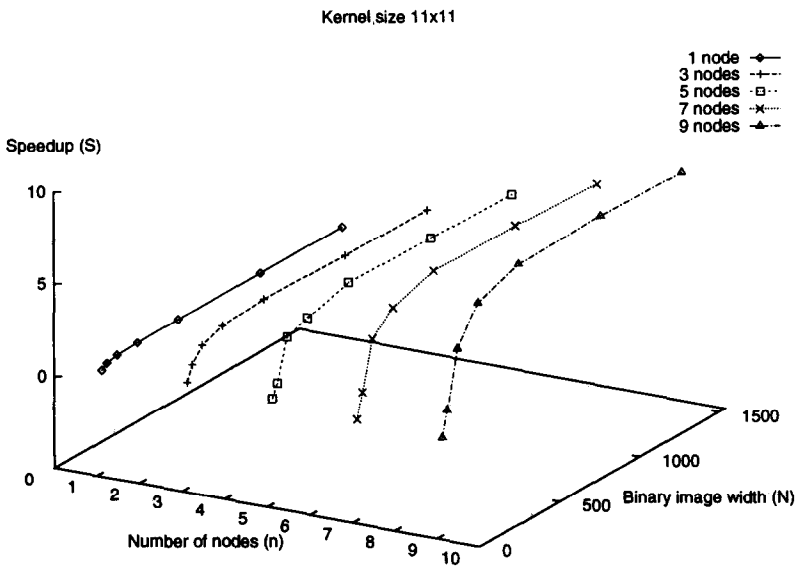

Fig. 14. The experimental speedup as a function of the binary image size and number of nodes for 11×11 kernel.

Fig. 13 compares the experimental execution time with the expected execution time of our performance prediction model, $T_{(n)}$, when the image matrix is 1024×1024 and the kernel size is 11×11. It shows that our prediction model agrees with the experimental results quite well. This in turn saves a lot of time trying to find the best performance without any help.

Binary image differs from grey-level images in both the communication time and the computation time. However, even though the communication and the computation time for the parallel binary image convolution are both lower than those for grey-level images, their ratio is very similar to that for grey-level images. This is illustrated by the the speedup tendency of the parallel image convolution for binary images shown in Fig. 14. Further, the same performance prediction model can be adopted for binary images, and it agrees closely with the experimental results. Moreover, we believe that the same speedup tendency would be obtained for color images.

6. Conclusion

In this paper, we have implemented efficient parallel image convolution algorithms for grey-level images and binary images on a network of IPC SUN Sparc workstations connected by an Ethernet network. Our experimental results indicate that significant speedup can be achieved in this computing environment for these applications. We also presented a performance prediction model that agrees well with our experimental results, and can be used to reduce the number of experi-

ments that should be carried to find the highest performance for a given image processing application on a network of workstations. The main limiting factor of this computing environment is the bandwidth of the network. Hence, with the emergence of high speed networks, this computing environment can be an attractive alternative to traditional MIMD and SIMD multiprocessors for computationally intensive applications especially in the area of image processing.

Appendix

A. Determination of α, β and γ

In this section, we show how the values of α, β, and γ used in our performance prediction model are found. The computation factor, γ, is found by measuring the time taken by a sequential program which performs N^2M^2 multiplications only. Since the execution time, T_s, is equal to $N^2M^2\gamma$, γ can be obtained easily in the following way:

$$\gamma = \frac{T_s}{N^2M^2}. \tag{21}$$

The computational factor, γ, is equal to 2.2 μs in our computing environment using SUN Sparc IPC workstations for grey-level images.

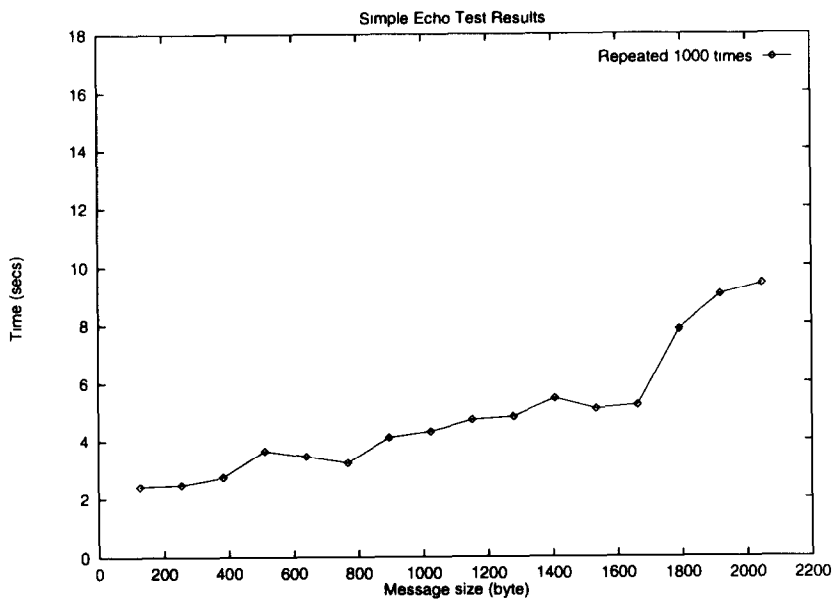

Fig. 15. The communication time of the echo test as a function of the image size.

Table 1
The execution time (secs) of pattern matching on a single workstations for different image sizes and kernel sizes

1 Workstation			
N	M = 3	M = 7	M = 11
32	0.173	0.234	0.357
64	0.214	0.621	1.277
128	0.665	2.095	4.540
256	2.110	8.211	17.925
512	7.808	31.136	71.003
1024	31.781	123.830	294.789
1536	72.463	278.496	644.588

In order to find the values of α and β, we run some simple echo tests which send/receive a number of messages between two workstations. Fig. 15 shows the corresponding communication times of sending/receiving messages using the echo test which consists of two processes where each process resides on a single workstation. Both processes do nothing but simply send and receive messages. For example, process 1 may send 500 messages, each of which is P bytes long to process 2 and then process 2 echoes the same amount of messages back to process 1. The echo test takes longer time to transmit longer messages, so the communication time is assumed to be a linear function given by, $T_{comm} = \alpha + P\beta$. The communication times of the echo test for various message sizes are shown in Fig. 15. The curve goes up steadily when the message size < 1792 bytes. Beyond this point, the curve jumps up quickly because of the fragmentation needed for long messages. Using the linear regression method to fit a straight line to this curve when the message size is in the range 128–1664 bytes, we find the values of α and β which are the message latency time and the incremental transmission time per byte. They are found to be 2.115 ms and 2.074 μs respectively in our computing environment.

Table 2
The execution time (secs) of pattern matching on 5 workstations for different image sizes and kernel sizes

5 Workstations			
N	M = 3	M = 7	M = 11
32	0.307	0.323	0.342
64	0.358	0.492	0.706
128	0.516	0.843	1.078
256	1.306	2.287	3.879
512	3.028	7.776	15.507
1024	10.871	29.212	63.616
1536	23.712	71.792	138.175

Table 3
The execution time (secs) of pattern matching on 9 workstations for different image sizes and kernel sizes

9 Workstations			
N	M = 3	M = 7	M = 11
32	0.348	0.372	0.382
64	0.362	0.384	0.404
128	0.833	0.725	1.077
256	1.271	1.759	2.906
512	3.585	5.372	10.288
1024	12.159	18.781	38.040
1536	28.994	42.168	87.220

B. Experimental data

The tables in this section show the execution time in seconds of the parallel image convolution (using pattern matching as an example) for various sizes of grey-level images, kernel sizes, and different number of workstations.

References

[1] G.S. Almasi, T. McLuckie, J. Bell, A. Gordon and D. Hale, Parallel distributed seismic migration, *Future Generation Comput. Syst.* 8(1–3) (1992) 9–26.
[2] I.G. Angus, G.C. Fox, J.S. Kim and D.W. Walker, *Solving Problems On Concurrent Processors* (Prentice-Hall, 1989).
[3] H.E. Bal, *Programming Distributed Systems* (Silicon Press, Summit, NJ, 1990).
[4] D.H. Ballard and C.M. Brown, *Computer Vision* (Prentice-Hall, 1985).
[5] C. Chakrabarti and J. JAJA, VLSI architectures for template matching and block matching. in: V.K. Prasanna Kumar, ed., *Parallel Architectures and Algorithms for Image Understanding* (Harcourt Brace Jovanovich, 1991).
[6] R. Cypher and J.L. Sanz, SIMD architectures and algorithms for image processing and computer vision, *IEEE Trans. on Acoustics, Speech, and Signal Processing* 37 (12) (1989) 2158–74.
[7] T.T. Elvins and D. Nadeau, NetV: an experimential network-based volume visualization system, in: *Proc. Visualization '91* (1991) 50–57.
[8] G.A. Geist and V.M. Sunderam, Network-based concurrent computing on the pvm system *Concurrency: Practice And Experience* 4 (1992) 293–311.
[9] C.R. Giardina and E.R. Dougherty, *Morphological Methods in Image and Signal Processing* (Prentice-Hall, 1988).
[10] R.C. Gonzalez and R.E. Woods, *Digital Image Processing* (Addison-Wesley, 1992).
[11] M. Hamdi and R.W. Hall, Compound networks for parallel image processing, in: *Computer Architecture For Machine Perception '91* (1991) 355–367.
[12] M. Hamdi and R.W. Hall, Image processing on augmented mesh-connected parallel computers, *J. Comput. Software Eng.* (1994).
[13] J.F. Jenq and S. Sahni, Reconfigurable mesh algorithms for image shrinking, expanding, clustering and template matching, in: *Fifth Int. Parallel Processing Symp.* (1991) 648–656.
[14] S.Y. Lee and J.K. Aggarwal, Exploitation of image parallelism via the hypercube, *Int. Conf. Parallel Processing* (1988) 344–350.
[15] J.J. Li, Parallel volume rendering of medical images, Technical report, Laboratoire LIP-IMAG, Ecole Normale Superieure de Lyon, 1991.

[16] G.J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons* (John Wiley, 1987).
[17] K.L. Ma and J.S. Painter, Parallel volume visualization on workstations, *Computer and Graphics* 17(1) (1993) 31–37.
[18] M. Maresca and H. Li, Morphological operations on mesh connected architecture: A generalized convolution algorithm, in: *Proc. Int. Conf. on Computer Vision and Pattern Recognition* (1986) 299–304.
[19] B.R. Meijer, Rules and algorithms for the design of templates for template matching in: *Proc. 11th IAPR Int. Conf. on Pattern Recognition*, vol. 1 (1992) 760–3.
[20] P.J. Narayanan, L.T. Chen and L.S. Davis, Effective use of SIMD parallelism in low- and intermediate-level vision, *IEEE Computer* 25(2) (1992) 68–73.
[21] Parasoft Corporation, An overview of the Express system, 1992.
[22] S. Ranka and S. Sahni, *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition* (Springer-Verlag, 1990).
[23] S. Ranka and S. Sahni, Image template matching on MIMD hypercube multicomputers, *J. Parallel Distributed Comput.* 10 (1990) 79–84.
[24] A. Rosenfeld and A.C. Kak, *Digital Picture Processing* (Academic Press, 1982).
[25] V.M. Sunderam, A framework for parallel distributed computing, *Concurrency: Practice And Experience* 2 (1990) 315–339.
[26] L. Uhr, *Parallel Computer Vision* (Academic Press, 1987).
[27] A. Umar, *Distributed Computing: A Practical Synthesis* (Prentice-Hall, 1993).
[28] C.C. Weems, Architectural requirements of image understanding with respect to parallel processing, in: *Proc. IEEE* 79 (1991) 537–47.
[29] C.C. Weems, C. Brown, J.A. Webb and T. Poggio, Parallel processing in the darpa strategic computing vision program, *IEEE Expert* 6 (1991) 23–38.