Vol. 41 No. 2 Transactions of Information Processing Society of Japan Feb. 2000

Regular Paper

Implementation of Parallel Image Convolution Processing Based on CORBA

Masayoshi Aritsugi,† Hiroki Fukatsu†,☆
and Yoshinari Kanamori†

To obtain the expected quality of image reproduction in image processing, it is usually necessary to perform a number of operations on the original image. Thus, it is desirable to reduce the cost of these operations. This paper discusses implementation of parallel image convolution processing based on CORBA. Employing CORBA makes it possible to exploit a cluster of heterogeneous workstations, each of which has a different level of computing power. The paper also presents an analytical model for the number of workstations appropriate for efficient image processing, and reports some experimental results.

1. Introduction

In image processing it is usually necessary to apply a number of operations to the original images in order to obtain good results. To obtain high-quality images, we sometimes need to apply complex image processing, which may consist of a sequence of image processing operations, to the original images. Combining operations to create an appropriate sequence sometimes involves applying several operations to images interactively. A typical situation in which this occurs is when a medical doctor tries to obtain the most exact areas of X-ray photographs showing cancer cells in a patient.

Thus, it is desirable to reduce the costs of image processing operations. The larger the images we manipulate, the longer we have to spend processing them. Parallel computing has been used for realizing efficient image processing[2),8),13)]. In this paper we propose an efficient image processing environment based on CORBA[12)], in which digital image convolution is performed in parallel.

We have thus far studied image manipulation based on CORBA[1),5),15)]. We model images with abstract objects called generic image objects, each of which has its own image data and operations in image processing, thereby supporting "version management" of images. Employing CORBA enables us to implement a cluster of heterogeneous workstations connected by a network. This paper extends our previous studies to align digital image convolution with distributed workstations.

This paper also discusses implementation methods for the proposed environment. To the best of our knowledge, there have been very few studies of parallel processing in a CORBA environment. We discuss two methods of setting up parallel image convolution processing operations: introducing operations that include inherent parallel processing, and introducing operations that control existing operations located in distributed heterogeneous workstations. In fact, we have implemented both methods, which will be discussed in a later section of this paper.

In addition, we present an analytical model for the number of workstations appropriate for image processing efficiently in a network of heterogeneous workstations. In the modern workplace, an organization, laboratory, or research group is likely to have the technological capacity to create an environment comprising many workstations connected by a network. Note that in such an environment not all workstations are necessarily running all the time; therefore, it is very practical to select appropriate workstations to perform parallel processing, even if the speed-up is not linear. Note also that such an environment consists of a wide variety of workstations, from different vendors and with different levels of computing power. There have been several studies on realizing parallel computing for image processing with parallel architectures[2),13)], and with a network of workstations[8),14)], and we used their parallel convolution algorithms in this study. However, to our knowledge, they focused on parallel pro-

† Department of Computer Science, Faculty of Engineering, Gunma University
☆ Presently with Fujitsu Terminal Systems Limited

cessing with homogeneous processing elements. In this paper we analyze the performance of parallel digital image convolution in a network of workstations that have different computing power. We also present some experimental results to show that the analytical model agrees with practical situations.

The remainder of this paper is organized as follows. Section 2 briefly explains the manipulation of images we have proposed for version management. The discussion in this paper is based on the manipulation method. In Section 3, two methods of implementation using parallel convolution with CORBA are proposed. A performance analytical model and experimental evaluation are shown in Section 4. In Section 5, our work is compared with that of others, and conclusions are given in Section 6.

2. Manipulation of Image Objects

In this section we briefly describe the manipulation of image objects in order to develop the discussion of this paper; more detailed descriptions, including consideration of version management, can be found elsewhere[1),5),15)].

Image processing creates a number of versions of the original images. This is simply because a desired image can be generally obtained by the interactive application of a number of image processing operations to the original image or images.

We model images with generic image objects (GIOs), which are abstractions of images—"versions" of them. GIOs are introduced for managing all relative versions of an image; every version of an image is connected with the image's GIO. An operation in image processing is applied to an image through its GIO, and the GIO is treated as a certain version of the original image.

To avoid restrictions on various resources, such as the underlying operating systems, the programming languages in which users can develop image database applications, and database management systems, we decided to develop our whole system by using CORBA[12)]. There are several techniques available for building distributed systems, such as CORBA and DCOM (distributed component object model). Although our system could be implemented with another technique, we have been using CORBA mainly because (1) it is the standard to which a large number of institutions subscribe, and (2) systems implemented in different kinds of programming languages can later be integrated.

```
class ImageProcessing{
public:
  CORBA::any* preProcessing(....);
  virtual CORBA::any*
            executeProcessing(....);
  CORBA::any* postProcessing(....);
....
};

class Edge:public ImageProcessing{
public:
  CORBA::any* executeProcessing(....);
  CORBA::any* edge(....);
};

class Smooth:public ImageProcessing{
public:
  CORBA::any* executeProcessing(....);
  CORBA::any* smooth(....);
};
```

Fig. 1 Definitions of the classes `ImageProcessing`, `Edge`, and `Smooth`.

Generic image objects are physically composed of two objects: image data objects (IDOs) and image processing objects (IPOs). An IDO holds pixel data, while an IPO corresponds to an image processing operation. Users send a message to a GIO in order to perform an image processing operation; this can be processed by binding the corresponding IDO and IPO dynamically; thereby we maintain image processing operation codes separately from pixel data, and, moreover, we can develop new operations and apply them to existing images fairly easily.

We develop classes for IPOs by following the interfaces defined in the OMG Interface Definition Language (IDL). To make the discussion concrete, we use C++ to express classes. **Figure 1** shows the definitions of the classes `ImageProcessing`, `Edge`, and `Smooth`.

Classes for image processing operations, including classes `Edge` and `Smooth`, are defined as subclasses of the class `ImageProcessing`, which has two methods, `preProcessing` and `postProcessing`, for processes commonly executed on images before or after the execution of any image processing operation, and a virtual method, `executeProcessing`.

Figure 2 depicts how image processing sent to a GIO is performed in the basic architecture.

1. A user obtains a reference to an IDO that holds the image to be processed. Assume that the image is stored in Site b.
2. The user obtains a reference to an IPO that

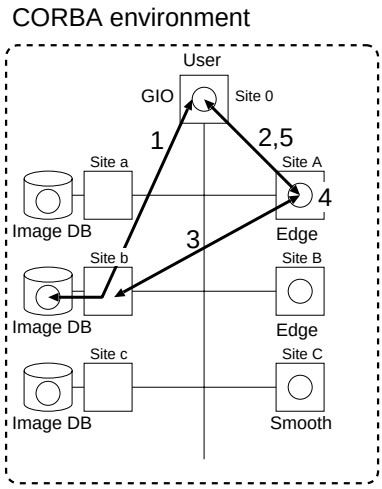

Fig. 2 Basic image processing architecture.

corresponds to the operation the user wants to execute. Assume that the IPO is stored in Site A. Then, the user sends message `executeProcessing` to the IPO with the IDO.

3. The IPO obtains the information on the image necessary for image processing, such as the size, pixels, etc.

4.1. The IPO executes `preProcessing` on the IDO.

4.2. The IPO executes the operation on the result of Step 4.1. In the figure, `edge` is performed.

4.3. The IPO executes `postProcessing` on the result of Step 4.2.

5. The IPO returns the final result to the user.

In the following, we try to perform Step 4.2 in parallel with a network of workstations.

3. Parallel Image Processing

In this section we give a rough overview of convolution algorithms. A detailed explanation can be found in several papers including Lee and Hamdi[8]. We then propose implementation methods for setting up parallel algorithms.

3.1 Convolution Algorithms

Parallel convolution has been addressed by many researchers[2),8),13)] because convolution is a general image processing operation; operations including smoothing, edge detection, and template matching are categorized under convolution; and it is easy to set up parallel convolution operations.

To make convolution parallel, an image is partitioned into the number of workstations employed. We assume that a workstation can calculate one fragment of the image. Let us consider the case where an image of size $N \times M$ is divided into the number of workstations, with no overlap when we calculate pixels close to the boundaries between subimages. **Figure 3** shows how the kernel overlaps the boundary between subimages `i` and `i+1`. In this case, when a workstation calculates the convolution of the pixels of subimage `i`, the kernel requires some pixels in subimage `i+1` that are delivered to another workstation, and, consequently, induces an extra communication overhead.

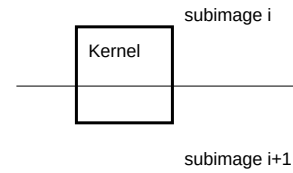

Fig. 3 Kernel overlapping a boundary between subimages.

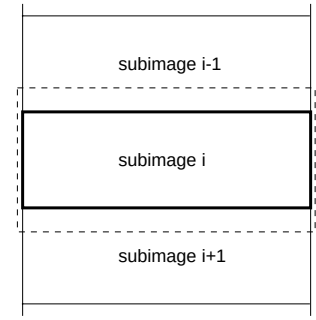

Fig. 4 A part of image divided using the row partition method with overlap.

To avoid the macrocommunication overhead, we adopt an overlap mapping method (**Fig. 4**). Using this method, there is no extra communication among workstations, each workstation can calculate pixels independently, and no workstation needs to know which other workstations have pixels close to a boundary.

There are several partition methods, such as row partition, cross partition, and heuristic partition. In this paper, for the sake of simplicity, we adopt a method of partitioning rows into square images. In the row partition method, a given square image is divided horizontally into n subimages, each of which has the same size, for n workstations, as shown in Fig. 4. A workstation receives all the data bounded by the dotted rectangle, instead of by the thick solid square.

Since we investigate a network of worksta-

tions with various levels of computing power, we have to consider an additional element of parallel image processing: how fragments of images are delivered to the workstations employed for processing. In this study, we deliver fragments in order of the computing power of all workstations. We will discuss this in Section 4.

In the rest of this section, we propose two methods of implementing a parallel image processing operation in a network of workstations. One method introduces IPOs that include parallel processing, and the other introduces IPOs that control existing IPOs distributed among heterogeneous workstations.

3.2 IPOs That Include Parallel Processing

One method we propose in this paper for parallel convolution is to introduce operations that include parallel processing. According to the manipulation of image objects we have proposed, this method introduces IPOs, which have the ability to perform image processing in parallel. Note that the interface of the new IPOs is the same as that of those shown in Fig. 1. That is, users can manipulate the new IPOs just as they do existing IPOs, or IPOs for sequential image processing operations.

This method can be implemented with parallel programming environments such as PVM (Parallel Virtual Machine)[10] and implementations of MPI (Message Passing Interface)[9], which have been used by many researchers in writing code for parallel programs. In fact, we have built some convolution operations with PVM. **Figure 5** shows how an IPO, which we have built with PVM, performs an image processing operation in parallel in a network of workstations.

Steps 1, 2, 3, and 5 in Fig. 2 are the same in this case. Let us see in detail how Step 4 in Fig. 2 is performed in parallel in this method.

4.1. The IPO, which is denoted as Master in Fig. 5, executes `preProcessing` on the IDO.
4.2. The IPO generates slaves in PVM. The number of slaves is equal to the number of workstations employed for parallel processing.
4.3. The IPO partitions the image data that the IDO holds into the number of slaves.
4.4. The IPO delivers fragments of the image to the slaves.
4.5. Each slave applies the operation in image processing to the fragment received from the IPO, and returns the result to the IPO.
4.6. The IPO generates the final result from the results of Step 4.5 sent from the slaves.
4.7. The IPO executes `postProcessing` on the final result.

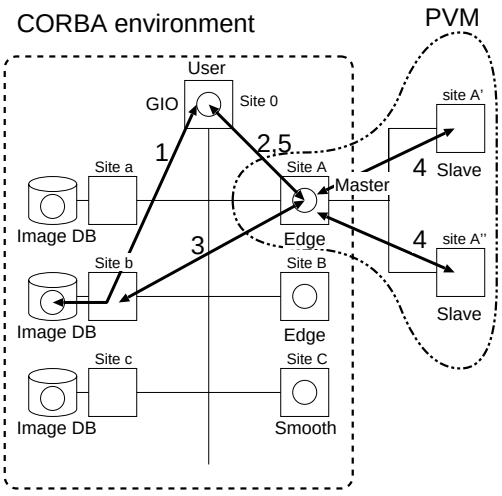

Fig. 5 An IPO that includes parallel processing with PVM.

Note that the method we have built is based on CORBA; that is, existing applications that manipulate conventional IPOs can be applied to introduced IPOs, and as a result, users can benefit from parallel processing of the IPOs. This is independent of which parallel programming environment is employed to implement IPOs that include parallel processing. While we employ PVM, IPOs implemented with an implementation of MPI, for example, can be integrated without any modification of existing application programs.

Although we can realize an efficient image processing environment with this method of parallel processing, we have to write a set of code for both master and slave, each with a parallel programming environment. Of course, we need to know not only CORBA programming but also parallel programming environments in order to use this method. Note that in convolution algorithms, processing of a slave program is the same as that of a program without parallel processing. But slave programs coded for this method can be used only through a master program.

3.3 IPOs That Control Existing IPOs

As we have seen, although we can realize an efficient image processing environment by introducing IPOs that include parallel processing,

```
class Master_Edge:public Edge{
public:
  CORBA::any* executeProcessing(....);
};

class Master_Smooth:public Smooth{
public:
  CORBA::any* executeProcessing(....);
};
```

Fig. 6 Definitions of the classes `Master_Edge` and `Master_Smooth`.

programming costs for implementing this environment might become large. Here we propose another method of parallel processing, by introducing IPOs that control existing IPOs in a network of workstations. We assume naturally that workstations employed for parallel processing have already stored IPOs used before parallel processing occurs.

We first prepare the master class for each operation in image processing. **Figure 6** shows the interface of the classes `Master_Edge` and `Master_Smooth` written in IDL, which will control the existing `Edge` and `Smooth` IPOs, respectively, shown in Fig. 1. This is required because we need to distinguish between conventional IPOs corresponding to each image processing operation and new IPOs controlling the conventional ones. It should be noted, however, that the interface of each kind of IPO uses the method `executeProcessing`, in order to apply its image processing operation. Consequently, existing application programs using conventional IPOs can benefit from the introduction of the new IPOs.

Figure 7 depicts how an IPO or `Master_Edge` in the figure, performs an image processing operation in parallel in a network of workstations. Similarly, Steps 1, 2, 3, and 5 in Fig. 2 are the same in this case. Let us see in detail how Step 4 in Fig. 2 is performed in parallel in this method.

4.1. The IPO, `Master_Edge` in the figure, executes `preProcessing` on the IDO.
4.2. The IPO partitions the image data that the IDO holds into the number of workstations employed for parallel processing.
4.3. To each slave, the IPO calls the operation `edge` in the figure with a fragment generated in Step 4.2 as the argument.
4.4. Each slave, or IPO with the ability to perform the operation sequentially, processes the fragment received from the IPO and returns the result to the IPO.
4.5. `Master_Edge` generates the final result from the results of Step 4.4 sent from the slaves.

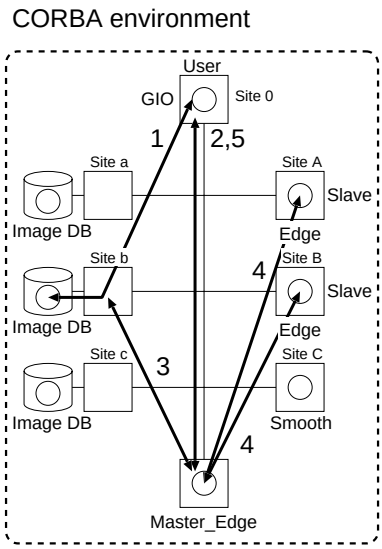

Fig. 7 An IPO that controls existing IPOs.

4.6. The IPO executes `postProcessing` on the final result.

In this method of parallel processing, we make good use of IPOs corresponding to each operation in image processing that exists in a network of workstations, and under the conditions of this use we do not need to modify them. We can thus exploit each IPO when performing the operation directly, as shown in Fig. 2.

4. Performance Evaluation

The main contribution of our study is toward a consideration of parallel image processing in a network of workstations with heterogeneous computing power. In this section we present an analytical model for such environments, and show and investigate some experimental results.

4.1 Performance Model

Advances in computer technology have made it possible to establish environments comprised of many powerful workstations connected by a network. Note that in such a network not all workstations are always running; in fact, the majority of workstations may be idle. Note also that various kinds of workstations are connected by a network. In this study we borrow the performance prediction model presented by Lee and Hamdi[8] and modify it to take account of a wide variety of workstations.

We analyze the execution time of parallel image processing with three costs, T_a, T_{bi}, and T_c, as shown in **Fig. 8**, in which Sites 1, 2, and 3 appear.

T_a: The preprocessing cost for parallel image

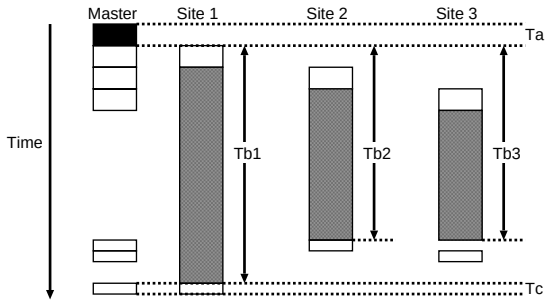

Fig. 8 Three costs of parallel image processing.

processing. This includes obtaining an image from a database, partitioning it into the number of workstations, and activating slave processes on distributed workstations.

T_{bi}: The time period extending from the time at which the Master in Fig. 8 begins to send fragments of the image to Site i to the time at which Site i completes its image processing.

T_c: The communication cost for receiving a fragment of image processing results from a Site.

In a network of workstations with different levels of computing power, performance might be affected by the way in which fragments of images are delivered to workstations. Consider, for example, a case in which two workstations, say A and B, with different levels of computing power (say A is more powerful than B) are used for parallel image processing. It is obvious that delivering a fragment to A and then to B takes more time than delivering a fragment to B and then to A. We can usually determine the computing power of each workstation in our laboratory. Hence we deliver fragments in order of the computing power of all workstations that we employ. In the following, we assume that all workstations are free and devote themselves to image processing after receiving a fragment.

Given an $N \times M$ image matrix and n workstations connected by a network, let us determine the value of T_a, T_{bi}, and T_c. In the following, values expressed with *dbsyst* and *distsyst* are determined by which database system and distributed system are used, respectively.

T_a: Let D_{dbsyst} be the time for obtaining an image from a database, $Par_{distsyst}$ be the time for partitioning the image into n fragments, and $Act_{distsyst}$ be the time for activating one slave process on a workstation. T_a is expressed as

$$D_{dbsyst} + Par_{distsyst} + n \times Act_{distsyst}. \quad (1)$$

T_{bi}: Let B and $P_{i_{distsyst}}$ be the number of pixels that can be transmitted through a network per second and the time for image processing on site i, which has the i-th highest computing power in a network of workstations we employ, per pixel (i.e., $P_{i_{distsyst}} \geq P_{j_{distsyst}}$ where $i < j$). Neglecting the extra number of pixels for the overlap mapping method (Fig. 4), the time, T_{bi}, from beginning to send fragments of the image to workstations to the time at which site i completes its image processing is approximated by

$$i \times \frac{N \times M}{n} \times \frac{1}{B} + \frac{N \times M}{n} \times P_{i_{distsyst}}. \quad (2)$$

T_c: Neglecting the fact that the size of the image processing result is smaller than that of given image, T_c is approximated by

$$\frac{N \times M}{n} \times \frac{1}{B}. \quad (3)$$

Using these values, we can calculate the execution time of the parallel image processing, T, as follows:

(1) for $i = 1$ to n do
  $ArrayT[i] \leftarrow T_{bi}$
(2) $ArrayC \leftarrow$ sort $ArrayT$
(3) $T_{b+c} \leftarrow \frac{N \times M}{B}$
(4) for $i = 1$ to n do
  if $T_{b+c} \leq ArrayC[i]$ then $T_{b+c} \leftarrow ArrayC[i] + T_c$
  else $T_{b+c} \leftarrow T_{b+c} + T_c$
(5) $T \leftarrow T_a + T_{b+c}$

Calculating the minimum value of T, we can obtain the number of workstations that achieve the minimum execution time of the parallel image processing.

4.2 Experimental Results

We have been developing efficient image processing environments with workstations in our laboratory, whose configuration is shown in **Table 1**. The workstations were connected by a 10-Mbps Ethernet. We created an image database with ObjectStore[11]. We adopted Orbix[3] as a CORBA environment and OOSA[4] as ODA. As we mentioned before, we have implemented a method introducing IPOs that include parallel processing with PVM[10]. In the following, we call this Method 1, and the other method introducing IPOs that control existing IPOs distributed among heterogeneous workstations Method 2.

Table 1 Testbed configuration.

Site	1	2	3	4	5	6	7	8	9
Machine type	Sun Ultra 1	Sun Ultra 1	Sun Ultra 1	Sun Ultra 1	Sun Ultra 30	Sun Ultra 30	Sun Ultra 30	Sun Ultra 30	Sun Ultra 30
CPU clock	143 MHz	167 MHz	167 MHz	167 MHz	248 MHz	248 MHz	248 MHz	296 MHz	296 MHz
Memory size	64 MB	128 MB	128 MB	128 MB	128 MB	128 MB	256 MB	128 MB	128 MB
OS	Solaris 2.5.1	Solaris 2.5	Solaris 2.5.1	Solaris 2.5	Solaris 2.5.1	Solaris 2.6	Solaris 2.6	Solaris 2.6	Solaris 2.5.1
Class	D	C	C	C	B	B	B	A	A

Table 2 D_{dbsyst}.

Image Size	32×2520	128×2520	512×2520
$D_{ObjectStore}$	0.380796	1.05362	3.76777

Table 3 $Par_{distsyst}$.

Image Size	32×2520	128×2520	512×2520
$Par_{method1}$	0.105456	0.105456	0.105456
$Par_{method2}$	0.072035	0.235139	0.89418

Table 4 $Act_{distsyst}$.

$Act_{method1}$	0.029110222
$Act_{method2}$	1.02871

In the experiments, we used smoothing as a convolution-type image operation. Three images were used for experiments: their respective sizes were 32×2520, 128×2520, and 512×2520. **Tables 2**, **3**, and **4** show the values of D_{dbsyst}, $Par_{distsyst}$, and $Act_{distsyst}$, respectively, that we used for calculating the expected time T.

Table 5 shows the values of $P_{i_{distsyst}}$ that we used for the calculation. As shown in Table 1, we used nine workstations and categorized them into four classes, namely, A, B, C, and D, depending on their computing power. Images were stored in an object database on Site 9, which was categorized into class A.

Note that, as shown in Tables 3, 4, and 5, the cost for image processing in Method 2 was lower than that in Method 1, while the values of $Par_{distsyst}$ and $Act_{distsyst}$ in Method 2 were larger than those in Method 1, given the resources we used for the experiments.

Figures 9, **10**, and **11** show the execution times for a smoothing operation with a 7×7 kernel on 32×2520, 128×2520, and 512×2520 images, respectively. In the figures, `Method 1` and `Method 2` are the execution times measured in the experiments by Methods 1 and 2, respectively, and `Expected Method 1` and `Expected Method 2` are those derived from the calculation with the values shown in Tables 2, 3, 4, and 5 in Methods 1 and 2, respectively.

During the experiments, neither the workstations nor the network were dedicated. Thus, the expected times must have included the cost of other tasks. In fact, the expected times were shorter than the measured times except when processing on Site 9. The reason the expected times exceeded the measured times when processing on Site 9 is that we did not exclude the data transmission cost from the expected times.

While the time for processing is not large in comparison with that for data transmission when small images are processed, the performance of parallel processing might be worse. Figure 9 indicates this situation. Although Method 1 improved the performance slightly, parallel processing by Method 2 grew less effective as the number of workstations rose. This is because the value of $Act_{distsyst}$ in Method 2 was large while that in Method 1 was small, in comparison with the values of $P_{i_{distsyst}}$.

The analytical model worked well for processing a 128×2520 image by Method 2, as shown in Fig. 10. However, it did not work for Method 1 in the experiments. This might be because the time spent on other tasks in the analysis affected the performance beyond our expectation. We need further analysis to investigate such a possibility; this is included in our plans for future work. As in Fig. 9, parallel processing of the 128×2520 image did little to reduce the cost of image processing.

Figure 11 shows that, in processing large images, the analytical model can predict the number of workstations needed to minimize the execution time, which suggests that, when the time required for image processing is large in comparison with that required for other tasks in

Table 5 $P_{i_{distsyst}}$.

Class	A	B	C	D
Method 1	5.279575e-05	5.779048e-05	8.879123e-05	10.04058e-05
Method 2	3.487723e-05	4.101035e-05	6.167813e-05	7.127604e-05

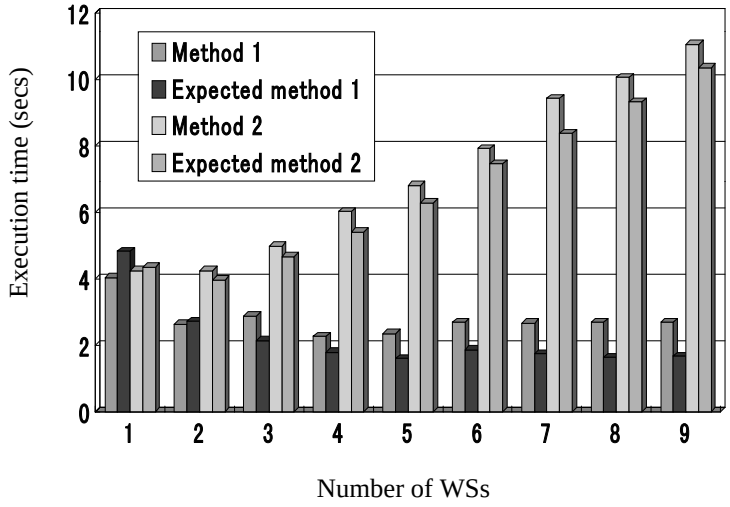

Fig. 9 A 32×2520 image with a 7×7 kernel.

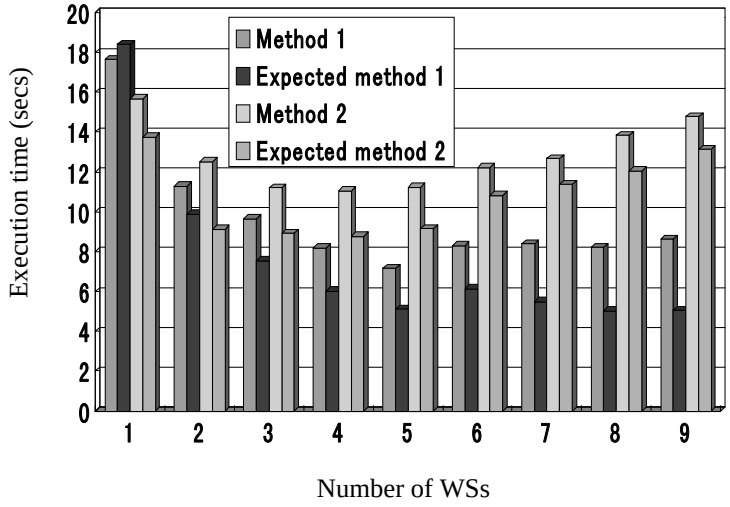

Fig. 10 A 128×2520 image with a 7×7 kernel.

such an environment, the analytical model can work well.

It should be noted that Fig. 11 shows that the speed-up with parallel processing in a network of workstations with different levels of computing power is not as easily effected as in a network of homogeneous workstations. For example, in Method 1, the execution time with five workstations was smaller than that with six workstations, and the time with eight workstations was the smallest. On the other hand, in Method 2, although the execution time with seven workstations was smaller than that with six workstations, image processing with five

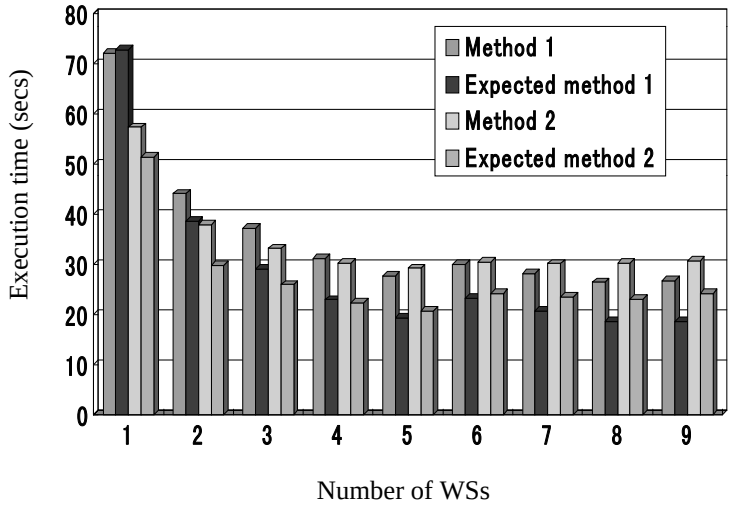

Fig. 11 A 512×2520 image with a 7×7 kernel.

workstations required the smallest cost. To our knowledge, no previous study has offered such a model of complex situations and performance.

5. Related Work

Parallel digital image convolution has been studied by many researchers. This is mainly because convolution is one of the most general image processing operations and it is easy to set up parallel convolution algorithms.

Ranka and Sahni[13] developed efficient algorithms for image template matching on MIMD hypercube multicomputers. Several algorithms for image processing on SIMD computers can be found in Cypher and Sanz[2]. While their studies used highly parallel architectures, we investigated parallel image convolution in a network of workstations.

Lee and Hamdi[8] discussed efficient parallel image convolution algorithms on a network of workstations, presented a performance prediction model, and reported their experimental results. The study described in this paper was motivated by their work. Our study differs from theirs in the following two respects: (1) We have studied the implementation of an efficient image processing environment based on CORBA, the standard for building distributed systems. (2) We have taken account of a network of workstations with different levels of computing power in parallel image processing.

Several parallel processing tools for using a network of workstations have been proposed. Squyres, et al.[14] showed an implementation of a cluster-based parallel image processing toolkit with MPI. Keahey and Gannon[6),7] implemented a system called PARDIS, extending the CORBA object model by introducing SPMD objects, enabling users to write programs with data-parallel computations in a distributed environment with ease. We intend to design and implement a software tool for developing parallel image processing by the methods described in this paper.

6. Conclusions

In this paper we have discussed methods of implementing parallel image convolution processing based on CORBA, and presented an analytical model. We have also reported some experimental results obtained with a network of workstations. Taking account of workstations with different levels of computing power, the number of workstations necessary to achieve the minimum execution time can be calculated by using the analytical model. Although the speed-up was not linear, we were able to obtain better performance with workstations in our laboratory, which are not always busy.

In the study, we evenly partitioned images into the number of workstations. If we partition images according to the computing power of each workstation, we might be able to obtain better performance. This goal is included in our plans for future work. In the study, we assumed that all workstations participating in

parallel processing were free and devoted themselves to image processing after receiving a fragment. However, in the real world this may not be true. We will extend the analytical model for application to more general situations.

References

1) Aritsugi, M., Tabata, M., Fukatsu, H., Kanamori, Y. and Funyu, Y.: Manipulation of Image Objects and Their Versions under CORBA Environment, *Proc. Intl. Workshop on Database and Expert Systems Applications* (*DEXA 97*), pp.86–91 (1997).
2) Cypher, R. and Sanz, J.: SIMD Architectures and Algorithms for Image Processing and Computer Vision, *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol.37, No.12, pp.2158–2174 (1989).
3) IONA Technologies Ltd.: Orbix 2 Programming Guide (1996).
4) IONA Technologies Ltd.: Orbix+ObjectStore Adapter Programming Guide (1997).
5) Kawashima, S., Tabata, M., Kanamori, Y. and Masunaga, Y.: Version Modeling for Image Objects, *Trans. IEICE*, Vol.J79-D-I, No.10, pp.843–852 (1996). (in Japanese).
6) Keahey, K. and Gannon, D.: PARDIS: A Parallel Approach to CORBA, *Proc. 6th IEEE Intl. Symposium on High Performance Distributed Computation* (1997).
7) Keahey, K. and Gannon, D.: PARDIS: CORBA-Based Architecture for Application-Level Parallel Distributed Computation, *SC97 Conference Proceedings* (1997). (*http://www.supercomp.org/sc97/proceedings/*).
8) Lee, C.-K. and Hamdi, M.: Parallel Image Processing Applications on a Network of Workstations, *Parallel Computing*, Vol.21, No.1, pp.137–160 (1995).
9) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1995).
10) The MIT Press: PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing (1994).
11) Object Design, Inc.: ObjectStore C++ API Reference Release 5.0 (1997).
12) Object Management Group, Inc.: The Common Object Request Broker: Architecture and Specification, Revision 2.1 (1997).
13) Ranka, S. and Sahni, S.: Image Template Matching on MIMD Hypercube Multicomputers, *Journal of Parallel and Distributed Computing*, Vol.10, No.1, pp.79–84 (1990).
14) Squyres, J., Lumsdaine, A. and Stevenson, R.: A Cluster-Based Parallel Image Processing Toolkit, Technical Report TR 95-1, Department of Computer Science and Engineering, University of Notre Dame, IN (1995).
15) Tabata, M., Aritsugi, M. and Kanamori, Y.: Implementing Version Management Mechanism for Image Objects under Distributed Environment, *IPSJ Trans. Databases*, Vol.40, No.SIG 5 (TOD 2), pp.79–90 (1999). (in Japanese).

(Received June 4, 1999)
(Accepted November 4, 1999)

Masayoshi Aritsugi received his B.E. and D.E. degrees in computer science and communication engineering from Kyushu University in 1991 and 1996, respectively. Since 1996, he has been a research associate at the Faculty of Engineering, Gunma University. His research interests include object-oriented programming languages/databases in parallel and distributed computing environments. He is a member of IPSJ and IEICE.

Hiroki Fukatsu received his B.E. and M.E. degrees in Computer Science from Gunma University in 1997 and 1999, respectively. He is presently with Fujitsu Terminal Systems Limited.

Yoshinari Kanamori received his D.E. degree from Tohoku University in 1975. Since 1991, he has been a Professor at the Department of Computer Science, Gunma University. His research interests include database systems and image processing. He is a member of IPSJ, IEICE, ACM, and IEEE-CS.