Software platform for parallel image processing and computer vision

Rin-ichiro Taniguchi, Yasushi Makiyama, Naoyuki Tsuruta,
Satoshi Yonemoto and Daisaku Arita

Department of Intelligent Systems, Kyushu University
6-1, Kasuga-koen, Kasuga, Fukuoka 816 JAPAN

ABSTRACT

We describe a general purpose environment for the development of parallel image processing/computer vision algorithms: PRIME (PaRallel Image Media processing Environment). "General purpose" here means that the environment is designed so as to be used on a variety of multi-processor systems ranging from tightly-coupled computers to loosely-coupled computers. The key point of the system is that it provides an architecture-independent programming environment for image processing and computer vision. We show the outline of PRIME, its implementation, and its preliminary performance evaluation.

Keywords: Parallel programming, image processing, computer vision, architecture-independence, MPI

1. INTRODUCTION

At present, spatial and temporal image resolution has become very fine, and the algorithms required for image processing and computer vision have become quite complex. Therefore, the demand for extremely high speed image processing systems is increasing rapidly, and many high speed image processing systems, especially based on parallel processing techniques, have been developed so far. This is because, most important image processing algorithms, both low-level and high-level ones, inherently imply great deal of parallelism, and, therefore, parallel processing techniques in image processing are quite effective. However, the most important problem is software, particularly, programming support tools for parallel image processing[2].[3] Although several tools including programming languages have been proposed and developed to make it easy to develop parallel image processing programs,[1] the problem common to these tools is the lack of portability of the parallel programs. This is quite a serious problem, because we cannot reuse the programs, and we have to re-develop many new programs when a new system becomes available.

To solve this problem, the authors have been developing a system-independent parallel programming environment for image processing and computer vision, called PRIME (PaRallel Image Media processing Environment), which enables us not only to re-use parallel image processing algorithms, but also to write efficient programs easily without detailed knowledge of target parallel machines. PRIME is designed to provide the following functionalities:

- Parallel programming support
  *Data parallel and function parallel programming for loosely-coupled parallel machines and tightly-coupled ones is supported.**
- Various image data structure
  Distributed image data structures for loosely-coupled parallel machines are supported.
- Image input/output
  Distributed image input/output and display for loosely-coupled parallel machines is supported

These functionalities are realized as a C++ class library and a C++ function library. Since we realize these functionalities using a popular and general purpose programming language, we can use it for various application systems. In this paper, we will show the outline of PRIME: first, the basic strategy of the system described; then we discuss the efficiency of the PRIME referring to implementation both on loosely-coupled parallel machines using MPI(Message Passing Interface)[4] and on tightly-coupled parallel machines.

Other author information: (Send correspondence to R.T.)
Email: rin@is.kyushu-u.ac.jp; Telephone: +81-92-583-7616; Fax: +81-92-583-1338

*The current version only supports data parallel.

SPIE Vol. 3166 • 0277-786X/97/$10.00

2. DATA PARALLEL IMAGE PROCESSING

2.1. Basic scheme of implementation

Basically, parallel image processing based on a data parallel scheme consists of the following three important parts:

Part(1) Data exchange for image data distribution and gathering
This is done both before and after the parallel execution of image processing on sub images:

- Image data distribution
  An image is divided into multiple sub images and each sub image is allocated to each processor. Types of distribution include stripe, grid, integrated, duplicated, etc.
- Image data gathering
  Processed results acquired on processors are gathered into an image.
- Image data redistribution
  When the structure of distributed image data should be changed into a different one suitable for the following image processing program, image data redistribution is necessary.

Part(2) Local image processing on each processor
Each processor applies local image processing to an allocated sub-image. This processing is executed in parallel on each processor.

Part(3) Data exchanging when local image processing is executed
When each processor executes its local image processing and data allocated to other processors are required, those data should be transferred, by inter-processor communication, from the processors where the required data are stored.

In the above three parts, there are quite many variations in (2), because it differs according to image processing algorithms. On the other hand, in (1) and (3) there are not so many variations in popular image processing/computer vision algorithms. For example, filtering with a 3×3 kernel proceeds as follows:

1. An image data is divided and distributed to processors.
2. Each processor calculates the filter output referring to values of pixels in the kernel. When pixels in the kernel are allocated to other processors, those data are transferred by inter-processor communication.
3. Processed results acquired in the processors are integrated into an image.

In the above three steps, except the calculation of the filter output, procedures are the same for any kind of 3×3 filtering.

According to this observation, in this paper, we propose:

- Part(1) is abstracted into object classes called Distributed Image Data (DID for short), which are classified according to image data types and distribution structures.
- Part(3) is implemented as a higher-order function which accepts a function representing local image processing in Part(2).

By encapsulating Part(1) and Part(2) into pre-defined system constructs, object classes and higher-order functions, we re-use descriptions of local image processing in Part(2). In addition, we can easily write parallel image processing programs without describing complex inter-processor communication. In the following, we will explain our implementation of Part(1) and (3).

2.2. Implementation on image data distribution

2.2.1. Representation of distributed image data

An efficient distribution of image data depends on the architecture of the target system. Of course, if the target system is a tightly-coupled parallel machine, data distribution is not necessary. However, a loosely-coupled one requires its own message communication mechanism, as well as efficient data distribution. In addition, we should establish different communication protocols for different image types, because the data size of pixels is different. Therefore, according to the image data type and distribution structure, we have implemented object classes of distributed image data. In the current PRIME, we support the following data distribution (see 1):

Integrated: An image is allocated to a unique processor.

Duplicated: Each processor has a copy of an image.

Horizontal: An image is divided horizontally into sub-images, and each sub-image is allocated to one processor. A distribution scheme in which an image is divided into rows and distributed to each processor row by row is also supported(**HorizontalSkip**).

Vertical: An image is divided vertically into sub-images, and each sub-image is allocated to one processor. A distribution scheme in which an image is divided into columns and distributed to each processor column by column is also supported(**VerticalSkip**).

Grid: An image is divided into $n \times m$ rectangular grids, and each grid (sub image) is allocated to each processor.

2.2.2. Structure of class DID

For every DID, its copies are held by each processor, and each copy of DID holds the following information:

- the size of the whole image
- the pointer to sub-image's data allocated to the processor holding the copy of DID
- the position and the size of sub-image allocated to the processor
- the total number of processors and the ID of the processor

DID is designed so that users cannot see the details of data distribution, but can use DID just as an image data type declaration. For example, when DID representing a binary image is declared, its statement in PRIME is as follows:

```
Image2D_Binary bimage(Horizontal, "pict1.pbm");
```

This declaration means that a binary image stored in file `"pict1.pbm"` is read into a horizontally distributed image data structure, which can be referred to through the variable `bimage`. In case of a gray-valued image or a color image, the size of pixel cannot be fixed. Therefore, using the template mechanism built in C++, we have implemented DID classes for gray-valued or color images. In the following example, the size of pixel is 8bit, or represented in char.

```
Image2D_Gray<unsigned char> gimage(Vertical, "pict2.pgm");
```

Table1 shows typical DID implemented in PRIME. File input/output and display for distributed images are implemented as methods of DID. Therefore, again, users can use these facilities without knowing the details of the target system architecture.

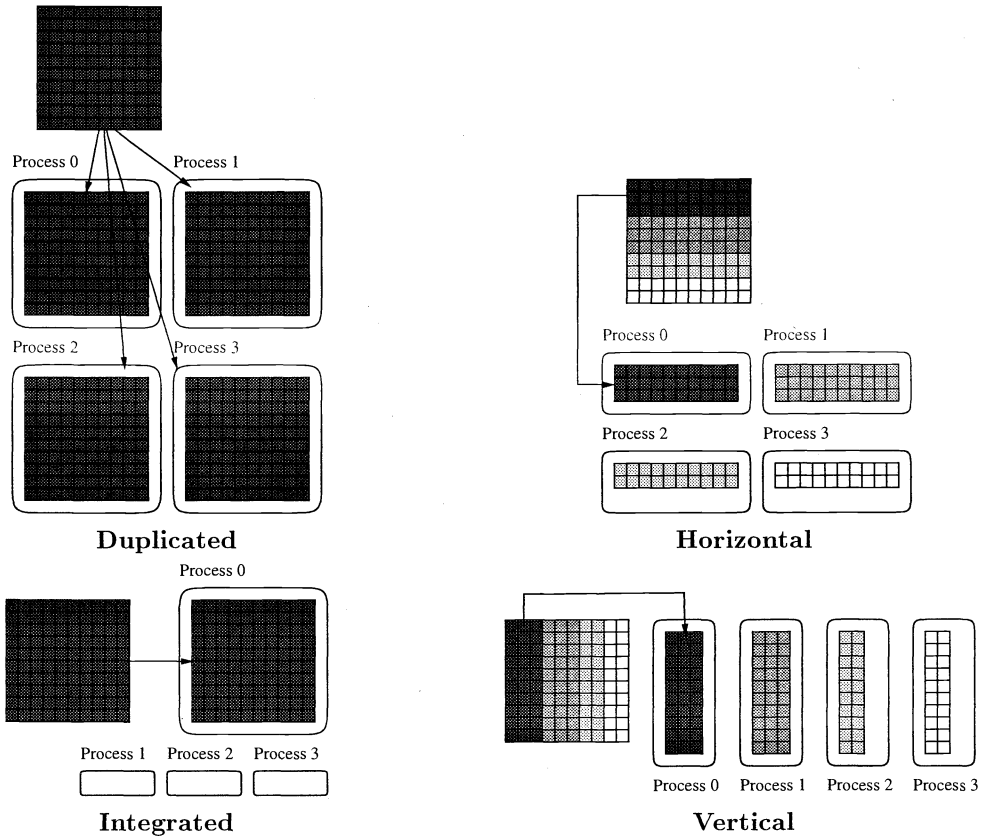

Figure 1. Major types of data distribution

Data type	DID class name
Binary image	`Image2D_Binary`
Gray image	`Image2D_Gray`
Color image	`Image2D_RGB`
Label image	`Image2D_Label`
Image pair	`Pair`
Image sequence	`Sequence`

Table 1. Major DID classes

3. ABSTRACTION OF INTER-PROCESSOR COMMUNICATION

As mentioned previously, only a limited number of patterns of inter-processor communication is required to execute local image processing. In other words, parallel algorithms of image processing can be classified into several classes based on a pattern of data access, which can be regarded as a communication pattern among processes, and also as a way of synchronization among processes. From this point of view, the followings are the major basic classes of image processing built in PRIME.

- Point operation
- Neighborhood operation

- Iterative neighborhood operation
- Histogram generating operation
- Divide-and conquer operation
- Butterfly operation
- Time-sequence operation

We abstract the inter-processor communication of these image processing classes into higher-order functions, called *communication functions*, and implement them on target parallel machines. Each communication function is ideally a higher-order function which takes the description of the architecture-independent computation, and then synthesizes and returns the whole program of the parallel image processing algorithms. When data is supplied to the synthesized program, the result of image processing is achieved (See Fig.2(a)). However, since we have used C++ as the base language, considering its availability and popularity in the community of image processing and computer vision researchers, this ideal implementation is not simple. This is because a function which returns a function as its result is not clearly described in C++. Instead, we implemented slightly modified higher-order functions, as shown in Fig.2(b), which are reduced from the higher-order functions expressed in Fig.2(a), and which take the description of the architecture-independent local computation and data to be processed at once, and which return the processed result.

For example, let us consider an averaging filter operation with a communication function for 3×3 filtering. This operation consists of the followings:

Communication function: pre-defined

Input: image, local computation function, mask size

Output: result image

Operation: communication to get local pixel values, execution of local computation function

Local computation function: user defined

Input: values of pixels within the mask

Output: output pixel value, or result of operation

Operation: averaging of input values

Local computation function is represented in PRIME as follows:

```
int Average3x3(int** mask, int rx, int ry, void* arg)
{
     return
        (mask[0][0] + mask[0][1] + mask[0][2] + mask[1][0] + mask[1][1]
         + mask[1][2] + mask[2][0] + mask[2][1] + mask[2][2]);
}
```

On the other hand, the interface to the communication function is defined as follows:

```
Filter(&image1, &image2, Average3x3, (char*)0, RX, RY);
// image1: input image (Image2D_Gray)
// image2: output image (Image2D_Gray)
// RX, RY: the size of mask (RX:3, RY:3)
```

Once the communication functions are installed, not only target independent but proper and efficient parallel image processing algorithms can be described regardless of the architecture of the target system using these functions. Fig.3 shows a short but complete parallel image processing program. The program, which does not seem to require further explanation, consists of (1)image data input, (2)Laplacian filtering execution, (3)image display, and (4)image file output. This shows that with PRIME we can easily write parallel image processing programs, which can be executed on various multi-processor systems without modification.

Although communication functions are target system dependent, of course, the number of these functions is limited, and, therefore, its implementation seems farely simple.

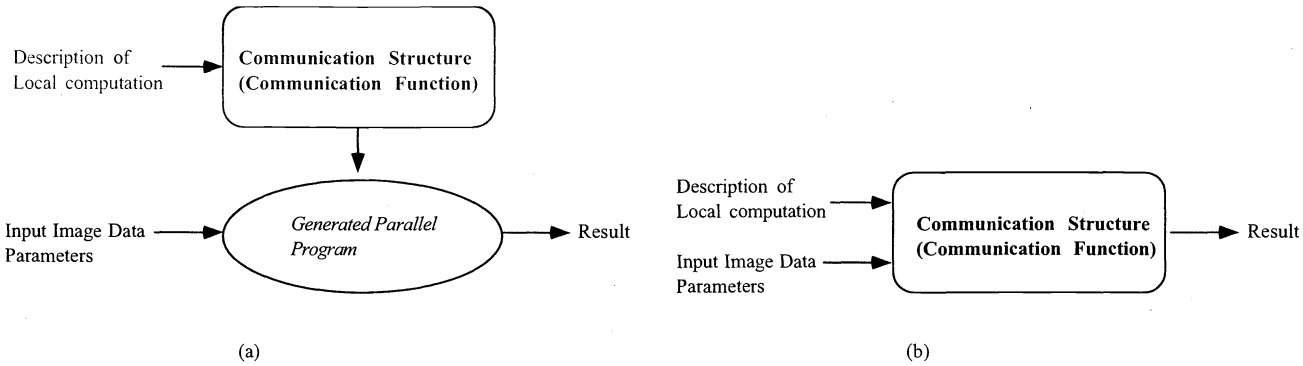

Figure 2. Hiding architecture dependency by means of a higher-order function

```
//Whole program of Laplacian operator
#include <prime.h>
void main(int argc,char** argv){
    PRM_Init(&argc,&argv);
    Image2D_Gray<int> image1(Horizontal,"picture1");
    Image2D_Gray<int> image2();

    Filter(&image1,&image2,laplacian,(char*)0,3,3);

    image2.display();
    image2.save("picture2");
    PRM_Finalize();
}
//
//Local computation
int laplacian(int** mask,int rx,int ry,void* arg)
{
    return
       (-mask[0][0] - mask[0][1] - mask[0][2]  - mask[1][0] + 8*mask[1][1]
        - mask[1][2] - mask[2][0] - mask[2][1] - mask[2][2]);
}
```

Figure 3. A programming example in PRIME

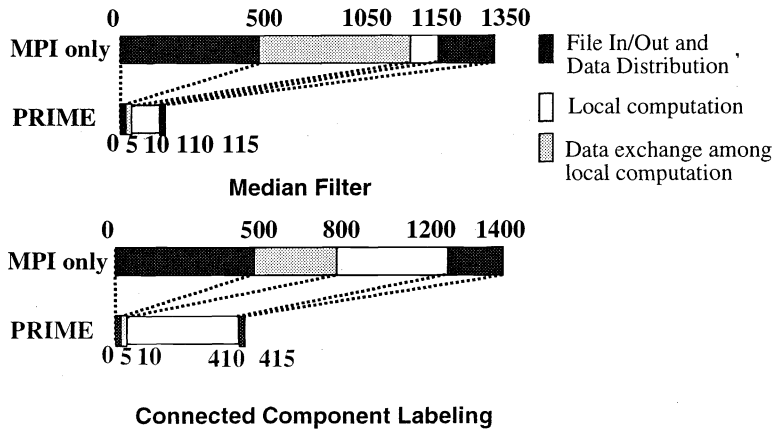

Figure 4. The size (in the number of lines) of programs written in PRIME

4. IMPLEMENTATION AND EVALUATION

Currently we have been implementing PRIME on two kinds of architecture: a tightly-coupled system and a loosely-coupled system connected via LAN (10Mbps Ethernet).[†]

4.1. Implementation on a tightly-coupled machine

We are porting PRIME on a Cray CS6400, which consists of 20 Sparc processors and has 1GB of shared main memory. Parallel execution is implemented on Solaris threads and "communication" is achieved by shared memory access. The implementation is rather straightforward, because data partitioning is not necessary and any processor can directly access data on the memory without explicit inter-PE communication. Porting to other tightly-coupled systems is not difficult. Only descriptions of thread/process creation should be modified.

4.2. Implementation on a loosely-coupled machine

The implementation of PRIME on loosely-coupled machines is one of the most important issues. The current implementation is based on MPI(Message Passing Interface)[‡].[4] Therefore, as far as MPI is supported on a target system, porting PRIME on it causes no problem. However, when the original, or raw, communication mechanism should be used to achieve higher execution efficiency, we should write some portions of communication descriptions.

4.3. Performance Evaluation

Now, we show several preliminary performance evaluation results. We have used two typical examples: one is median filtering, which has a neighborhood communication pattern, and can be simply described and efficiently executed on PRIME; the other is connected component labeling, which requires global communication, and which cannot be efficiently executed.

Fig.4 shows the comparison of the size (in the number of lines) of programs written with PRIME and of programs written only with MPI primitives. This indicates that PRIME quite effectively simplifies the program description, especially reducing the description of the inter-PE communication and the distributed data handling on loosely-coupled machines.

Fig.5 shows the speed-up rate on a CS6400 and on a WS cluster which consists of up to 8 Sparc stations connected via 10Mbps Ethernet. The image size here is 640×480 for median filtering and 512×512 for connected component labeling. The speed-up 1.0 means here that the performance is equal to that of the program optimized for sequential

[†]We are also porting it on a PC cluster connected via Myrinet, 1.28Gbps switching network.
[‡]we mainly consider its execution on WS/PC clusters.

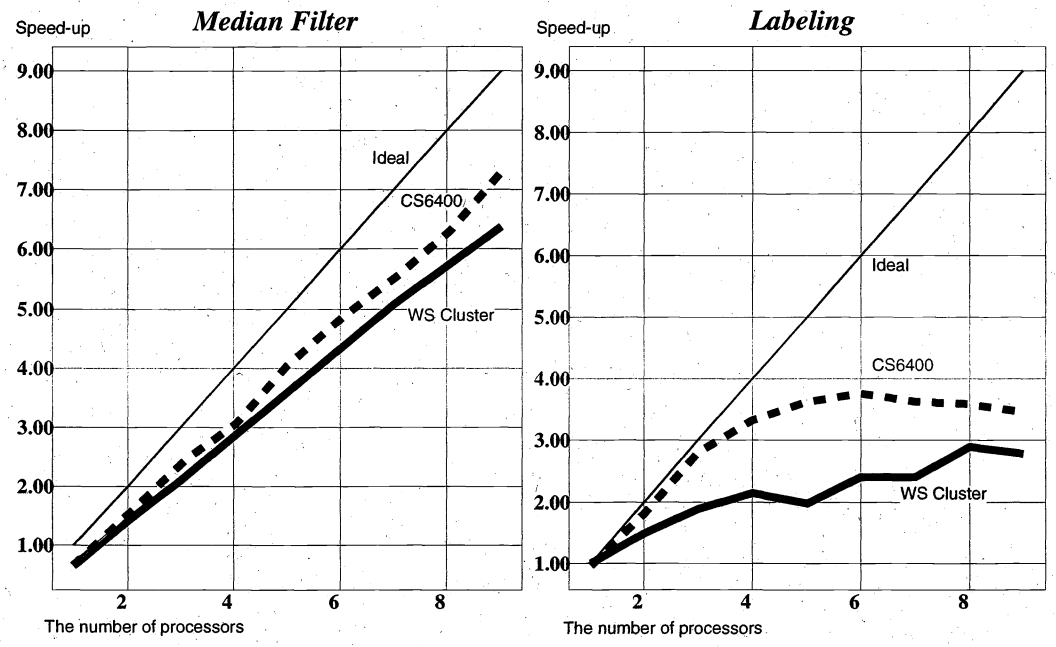

Figure 5. Performance evaluation of PRIME

v

execution on a single processor [§]. The figure shows that fairly high performance can be achieved for image processing consisting of local operations. In the connected component labeling, we cannot achieve high performance as in the median filtering, because the connected component labeling includes sequential characteristics, and essentially, it is not easy to achieve quite high performance on multi-processor systems.

Fig.6 shows the overhead of PRIME in the median filtering. It is caused by a lot of function-calls evoked when each pixel value is calculated by a local computation function. This experiment shows that the overhead of PRIME is almost negligible, and we can conclude that PRIME can quite effectively write efficient parallel image processing programs.

5. CONCLUSION

We have given an overview of PRIME, a parallel image media processing environment, and its preliminary performance evaluation. The aim of this system is to provide architecture-independent programming environment ranging from tightly-coupled multi-processor systems to loosely-coupled multi-processor systems. The key points to achieve this goal are the encapsulation of the distributed image data structures by the object-oriented paradigm, and the abstraction of inter-PE communication by higher-order functions. We have shown the simplicity of the programming through some experiments, and also shown its high efficiency in the program execution.

Since the current version of PRIME only provides data parallel execution on homogeneous parallel systems, which means not all the processor elements have the same architecture, we are planning to provide support for function parallel execution and the execution on heterogeneous parallel systems. In this work, task partitioning and load balancing are the key issues for efficient execution.

REFERENCES

1. Special issue on parallel processing for computer vision and image processing, *Computer*, Vol.25, No.2, 1992.

[§]There is no overhead caused by parallel program execution.

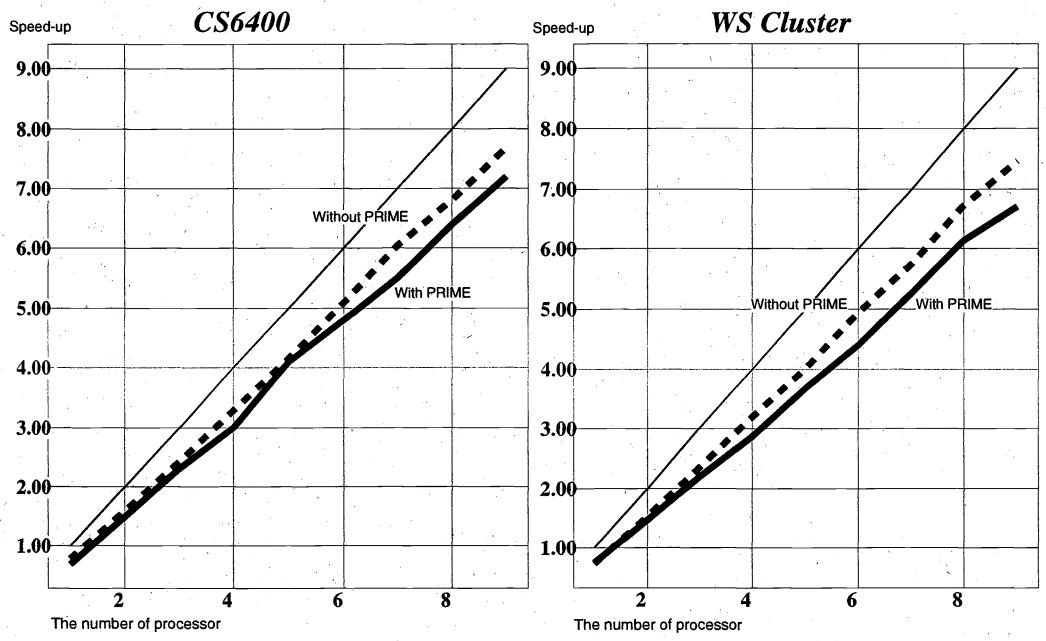

Figure 6. Overhead estimation of PRIME

2. Wallace.R.S, et ail, "Machine-independent image processing:performance of Apply on diverse architecture," *Proc. of Image Understanding Workshop*, pp.602-608, 1988.
3. Webb.J.A., " Unifying Concepts in Architecture Independent Parallel Image Processing in Adapt," *Proc. of SPIE Image Processing and Interchange*, pp.228-239, 1992.
4. Message Passing Interface Forum, "MPI: A message-passing interface standard," *Int. J. of Supercomputer Applications*, 8(3/4), 1994.