RPV: A Programming Environment for Real-time Parallel Vision —Specification and programming methodology—

Daisaku Arita, Yoshio Hamada, Satoshi Yonemoto and Rin-ichiro Taniguchi

Department of Intelligent Systems, Kyushu University
6-1 Kasuga-koen, Kasuga, Fukuoka 816-8580 Japan
{arita,yhamada,yonemoto,rin}@limu.is.kyushu-u.ac.jp

Abstract. A real-time distributed image processing system requires data transfer, synchronization and error recovery. However, it is difficult for a programmer to describe these mechanisms. To solve this problem, we are developing a programming tool for real-time image processing on a distributed system. Using the programming tool, a programmer indicates only data flow between computers and image processing algorithms on each computer. In this paper, we outline specifications of the programming tool and show sample programs on the programming tool.

1 Introduction

Recently, the technology of computer vision is applied to more various fields. CDV (Cooperative Distributed Vision) project[1, 2] in Japan aims to establish scientific and technological foundations to realize real world oriented practical computer vision systems. One of the research issue of the CDV project is observation of objects/environments with multiple sensors. When we use multiple sensors, or cameras, a distributed system with multiple computers is more suitable than than a centered system with only one computer because the performance of a distributed system can be easily increased to adapt the number of sensors by increasing the number of computers. To construct such a high-performance distributed system with low cost, we are developing a PC cluster, a set of PCs connected via high speed network, for real-time image processing[3, 4].

Though PC-based distributed systems have many merits, they have also some problems. One of them is that it is difficult for a user to make a system with high performance and stability, because when a user writes programs for a real-time distributed vision system, he or she must make attentions to data transfer, synchronization and error recovery. Their description requires a lot of knowledge about both hardware and software such as network, interruption, process communication and so on, and it is not quite easy. In this paper, we will propose PRV(Real-time Parallel Vision) programming tool for real-time image processing on a distributed system. Using RPV programming tool, a user does not have to write programs of data transfer mechanism, synchronization mechanism and error recovery functions, but only need to write programs of data flow between computers and processing algorithms on each computer.

2 System Overview

2.1 Hardware Configuration

Our PC cluster system consists of 14 Pentium-III×2 based PCs. All the PCs are connected via Myrinet, a crossbar-switch based gigabit network, and six of them have real-time image capture cards, ICPCI, which can capture uncompressed images from a CCD camera in real-time. Six CCD cameras are synchronized by a sync-generator, and, therefore, image sequences captured by those cameras are completely synchronized.

In addition, the internal clocks of all the PCs are synchronized by *Network Time Protocol*[5], and the time stamp when each image frame is captured is added to each image frame. Comparing the time stamps of image frames captured by different capturing components with each other, the system identifies image frames taken at the same time.

2.2 Software Architecture

On our PC cluster we consider that the following parallel processing schemes and their combinations are executed. From the viewpoint of program structure, each PC corresponds to a component of a structured program of image processing.

Data gathering Images captured by multiple cameras are processed by PCs and integrated on the succeeding processing stage.

Pipeline parallel processing The whole procedure is divided into sub-functions, and each sub-function is executed on a different PC sequentially.

Data parallel processing Image is divided into sub-images, and each sub-image is processed on a different PC in parallel.

Function parallel processing Images are multicast to multiple PCs, on which different procedures are executed in parallel, and their results are unified in the succeeding processing stage.

2.3 Modules

In each PC, the following four modules are running to handle real-time image data(See Figure 1). Each of them is implemented as a UNIX process.

Data Processing Module(DPM) This module is the main part of the image processing algorithms, and is to process data input to the PC. It receives data from a DRM and sends data to a DSM via UNIX shared memory.
In DPM, any programs should consist of three elements: a main loop to process input stream, in which one iteration is executed in one frame time; pre-processing before entering the loop; post-processing after quitting the loop (Figure 1). The main loop is executed according to the following procedure to process image sequence continuously.

1. Wait for a signal from FSM to start processing. If a signal arrives before starting to wait, an error recovery function is invoked.

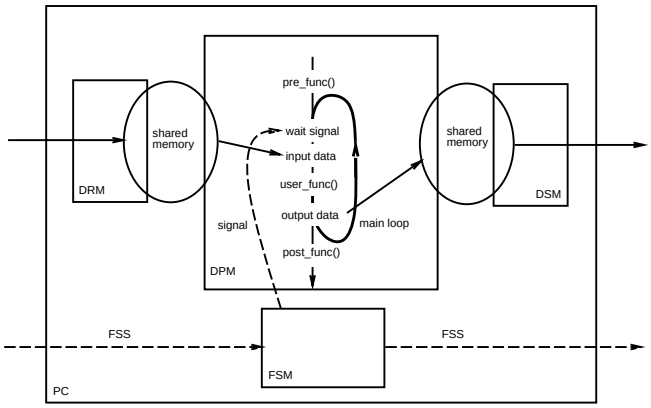


Fig. 1. Modules and Functions

2. Get input data. If input data has not been received, an error recovery function is invoked.
3. Execute a user-defined function representing one iteration of the main loop, which is named `user_func` here. Function `user_func` receives synchronous input data I and asynchronous input data A and sends output data O. Synchronous input data I are main streams of data, which originates from image capture cards and are transferred between PCs synchronously. They are synchronized at the beginning of function `user_func` (described at previous step). Asynchronous input data A can be used for feedback and cooperative processing. They are not synchronized at the beginning of function `user_func`.
4. Put output data. Because output data are directly written to shared memory in order to avoid data copy, only a notification of write-done is sent to DSM.

Before entering the main loop, a pre-processing function, which is named `pre_func` here, is executed. Function `pre_func` is a user-defined function, which is used to initialize DPM and to prepare for the main processing. After exiting the main loop, a post-processing function, which is named `post_func` here, is executed.

Data Receiving Module(DRM) This module is to receive data from other PCs via messages[1], and has buffers for queuing data. When a data request demand arrives from its succeeding DPM, it returns pointers to data.

Data Sending Module(DSM) This module is to send data to other PCs via messages, and has buffers for queuing data. When processed data arrives from its preceding DPM, it sends the data to the succeeding PCs.

[1] Message passing mechanism is developed using PM library[6].

Frame Synchronization Module(FSM) This module is introduced to make executions of different DPM synchronize with each other[4]. FSM sends FSSs to the succeeding FSM, and/or receives FSSs from the preceding FSM. FSM also sends start signals to activate the DPM in the PC.

3 RPV Programming Tool

Describing an entire program of real-time image processing on the PC cluster is not simple, because we have to describe real-time data transfer and synchronization mentioned above. To make the programming simple, we are developing RPV, as a C++ library, a programming environment for real-time image processing on the PC cluster. With RPV, users only have to describe essential structures of the programs, which are image processing algorithms on each PC and connection, i.e., data flow among PCs.

3.1 Class `RPV_Connection`

Data flow among PCs is described in Class `RPV_Connection`. Each PC sends and receives data according to the value of Class `RPV_Connection`. The specification of class `RPV_Connection` is shown in Fig 2. Member `keyword` indicates which function should be invoked in the PC (See examples in Figure 5 and Figure 6). The value of `RPV_Connection` varies with the PC, and it should be careful to define the values consistently in the programs of the PCs. To avoid this difficulty, here, we have designed a method with which the value of `RPV_Connection` can be defined by referring to a unique "connection file." The information is stored in a table with the following column headings:

#PCno keyword i_PC i_size i_num o_PC o_size a_PC a_size a_num

Each row describes connections on one PC. The columns are space-separated and show the PC number, keyword, IPC_m, sizes of $I_{m,t}$, S, OPC_n, sizes of $O_{n,t}$, APC_l, sizes of $A_{l,r}$ and R. Multiple specifications in one column are separated by commas. A '-', a 'c' in column i_PC and a '<' in column o_size are used to indicate that there is no entry in a column, the PC captures images from a camera and the PC broadcasts the left-neighbor data respectively.

3.2 Function `RPV_Invoke`

Function `RPV_Invoke` generates all modules (DRM, DPM, DSM and FSM). Its arguments specify the connection relation among PCs (in `RPV_Connection`), the synchronization mode and parameters required to execute DPM including function names corresponding to `user_func`, `pre_func` and `post_func`. The specification of Function RPV_Invoke is shown in Figure 3.

A user programs only these functions, `pre_func`, `user_func` and `post_func`, without concerning data transfer and synchronization. These functions can pass data via their last arguments.

```
class RPV_Connection{
   int myPC_no;          // PC number
   char* keyword;        // keyword indicating functions invoked in a PC
   int input_PC_num;     // the number of PCs from which data are received: M
   int* input_PC;        // PC numbers data are received from: IPC_m
                         //    (m = 0, ..., M - 1)
   int* input_data_size; // sizes of received data: sizes of I_{m,t}(m = 0, ..., M - 1)
   int input_frame_num; // the number of frames processed one time: S
   int output_PC_num;    // the number of PCs to which data are sent: N
   int* output_PC;       // PC numbers data are sent from: OPC_n(n = 0, ..., N - 1)
   int* output_data_size; // sizes of sent data: sizes of O_n, t(n = 0, ..., N - 1)
   int asynch_PC_num;    // the number of PCs from which asynchronous data are
                         //    received: L
   int* asynch_PC;       // PC numbers asynchronous data are received from: APC_l
                         //    (l = 0, ..., L - 1)
   int* asynch_data_size; // sizes of asynchronous data: sizes of A_l, r(l = 0, ..., L - 1)
   int asynch_data_num; // the number of asynchronous data processed one time: R
};
```

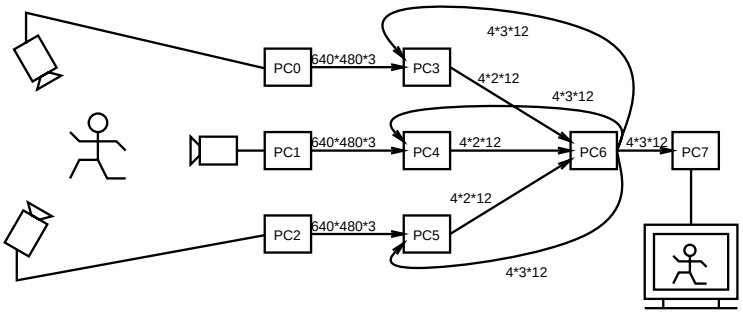
Fig. 2. Definition of Class `RPV_Connection`

```
void RPV_Invoke(
   RPV_Connection* connect,                          // Connection informations
   RPV_SynchMode sync_mode,                          // synchronization mode
   int frame_num,                                    // the number of processed frame
   void* (*pre_func)(void*),                         // Function executed before loop
   void* pre_func_arg,                               // Argument for pre_func
   void* (*user_func)(RPV_Input*, RPV_Output*, // Function executed in loop
                      RPV_Asynch*, void*),
   void* user_func_arg,                              // Argument for user_func
   void* (*post_func)(void*),                        // Function executed after loop
   void* post_func_arg                               // Argument for post_func
);
```

Fig. 3. Function `RPV_Invoke`

3.3 Sample Programs

Since programming style with RPV is based on SPMD (Single Program Multiple Data) paradigm, the same `main` program is executed on all the PCs. Of course, image processing functions to be executed vary with the PC, depending on the keyword described in the connection file. Figure 4 is an outline of a sample system, which captures video images of a human with 12 color markers, calculates 3D positions of color markers and animates an avatar in a display



Numeral values are data sizes of data flow. For example 640*480*3 means image width * image height * byte per pixel and 4*2*12 means sizeof(int) * 2D coordinates * 12 markers.

Fig. 4. Sample System

PC	keyword	i_PC	i_size	num	o_PC	o_size	a_PC	a_size	a_num
0	smooth	c	640*480*3	1	3	640*480*3	-	-	-
1	smooth	c	640*480*3	1	4	640*480*3	-	-	-
2	smooth	c	640*480*3	1	5	640*480*3	-	-	-
3	calc2D	0	640*480*3	1	6	4*2*12	6	4*3*12	1
4	calc2D	1	640*480*3	1	6	4*2*12	6	4*3*12	1
5	calc2D	2	640*480*3	1	6	4*2*12	6	4*3*12	1
6	calc3D	3,4,5	4*2*12,4*2*12,4*2*12	1	7,3,4,5	4*3*12,<,<,<	-	-	-
7	display	6	4*3*12	1	-	-	-	-	-

Fig. 5. Sample Connection File

in real-time. PC0, PC1 and PC2 capture and smooth video images. PC3, PC4 and PC5 extract color markers and calculate 2D positions of color markers. PC6 calculates 3D positions of color markers. PC7 make an animation of an avatar.

Figure 5 is a connection file that initializes class `RPV_Connection`. In this sample, functions with keyword "`smooth`", functions with keyword "`calc2D`", a function with keyword "`calc3D`" and a function with keyword "`display`" are invoked in PC0, PC1 and PC2, in PC3, PC4 and PC5, in PC6 and in PC7 respectively. PC0, PC1 and PC2 capture images, and PC6 broadcasts output data to PC7, PC3, PC4 and PC5.

Figure 6 is function `main` of all PCs, which calls function `RPV_Invoke`. Figure 7 is a program of function `user_func` executed in PC3 – PC5. Because, as shown in these sample programs, a user only programs static image processing algorithms and describes information about data flow, he or she can easily construct a real-time distributed image processing system.

```
int main(void)
{
   ifstream fs(CONNECTION_FILE_NAME);
   const RPV_Connection* connect = RPV_MainInit(fs);
   fs.close();
   if (strcmp(connect->keyword, "smooth") == 0) {
      RPV_Invoke(connect, RPV_DataMissing, 0, NULL, NULL, &Smooth,
                  NULL, NULL, NULL);
   }
   else if (strcmp(connect->keyword, "calc2D") == 0) {
      ReadBackgroundArg read_background_arg(BACKGROUND_FILE_NAME);
      RPV_Invoke(connect, RPV_DataMissing, 0,
                  &ReadBackground, &read_background_arg, &Calculate2D,
                  &read_background_arg.background, NULL, NULL);
   ..............
}
```

Fig. 6. Sample Program (main)

```
#include <fstream.h>
#include <rpv.h>

void* Calculate2D(const RPV_Input* id, RPV_Output* od,
                  const RPV_Asynch* ad, void* a)
{
   const RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>* i_data
      = (const RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>*)
                        id->data_ptr[0][0];
   Positions2D<MARKER_NUM>* o_data
      = (Positions2D<MARKER_NUM>*)od->data_ptr[0];
   const Positions3D<MARKER_NUM>* a_data
      = (const Positions3D<MARKER_NUM>*)ad->data_ptr[0][0];
   const RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>* background
      = (RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>*)a;

   RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>* sub_data
      = Subtraction(i_data, background);

   // Searching for markers from the subtracted image
   SearchMarker(sub_data, a_data, o_data);

   return NULL;
}
```

Fig. 7. Sample Program (Calculate2D)

4 Conclusion

In this paper, in order to makes it easy to program real-time distributed image processing systems, we proposed RPV programming tool, which is implemented as a C++ library. Because a user have to only describe data flow and processing algorithms on PCs, he or she does not have to concern such problems of the systems like data transfer, synchronization and error recovery.

Our future works are as follows:

- performance, program simplicity and applicability of RPV programming tool should be carefully evaluated.
- sophisticated job scheduling scheme should be considered.
- error recovery process should be improved.

Acknowledgement

This work has been supported by "Cooperative Distributed Vision for Dynamic Three Dimensional Scene Understanding (CDV)" project (JSPS-RFTF96P00501, Research for the Future Program, the Japan Society for the Promotion of Science).

References

1. Takashi Matsuyama: "Cooperative Distributed Vision", *Proceedings of 1st International Workshop on Cooperative Distributed Vision*, pp.1–28, 1997.
2. Takashi Matsuyama: "Cooperative Distributed Vision – Integration of Visual Perception, Action, and Communication –", *Proceedings of Image Understanding Workshop*, 1999.
3. D. Arita, N. Tsuruta and R. Taniguchi: Real-time parallel video image processing on PC-cluster, *Parallel and Distributed Methods for Image Processing II, Proceedings of SPIE*, Vol.3452, pp.23–32, 1998.
4. D. Arita, Y. Hamada and R. Taniguchi. A Real-time Distributed Video Image Processing System on PC-cluster, *Proceedings of International Conference of the Austrian Center for Parallel Computation(ACPC)*, pp.296–305, 1999.
5. D. L. Mills: Improved algorithms for synchronizing computer network clocks, *IEEE/ACM Trans. Networks*, Vol.3, No.3, pp.245–254, 1995.
6. H. Tezuka, A. Hori, Y. Ishikawa and M. Sato: PM: An Operating System Coordinated High Performance Communication Library, *High-Performance Computing and Networking* (eds. P. Sloot and B. Hertzberger), 1225 of Lecture Notes in Computer Science, Springer-Verlag, pp.708–717, 1997.