コンピュータビジョンと
イメージメディア 119－2
（1999．11．17）

実時間並列動画像処理ツール RPV

濱田 義雄、有田 大作、谷口 倫一郎

九州大学大学院システム情報科学研究科知能システム学専攻
{yhamada,arita,rin}@limu.is.kyushu-u.ac.jp

実時間並列動画像処理を行うためには、実時間データ転送、同期、エラー処理が必要となる。しかし、プログラマが、これらのメカニズムを記述することは大変難しい。そのため、我々は、PC クラスタ上での実時間動画像処理のためのプログラミングツール RPV を開発している。RPV を使うことにより、プログラマは、PC 間のデータフロー情報と、各 PC で動く画像処理アルゴリズムを記述するだけで、並列動画像処理を行うことが出来る。本報告では、この RPV の概要について述べ、RPV を用いたサンプルプログラムを示す。

和文キーワード: 実時間動画像処理、 並列処理、 PC クラスタ、 並列プログラミング環境

RPV : A tool for real-time parallel video image processing

Yoshio Hamada, Daisaku Arita, Rin-ichiro Taniguchi

Department of Intelligent Systems, Kyushu University

A real-time distributed image processing system requires mechanisms of 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 RPV for real-time image processing on a PC-cluster. Using RPV, a programmer indicates only data flow between PCs and image processing algorithms on each PC. In this paper, we outline specifications of RPV and show sample programs on using RPV.

English Keyword: real-time video processing, parallel procesing, PC-cluster, parallel programming environment

1 はじめに

近年、複数のカメラを利用するアプリケーション、例えば広範囲な物体追跡、モーションキャプチャ、自動講義撮影などに関する研究が盛んに行われている [1]。このように複数のカメラを利用する場合、物理的な制約およびI/O 能力の限界により、1 台の計算機に接続できるカメラの台数が制限されてしまうという問題がある。

これらの問題を解決するために、我々は複数のPC を高速ネットワークで結合したPC クラスタを用い、その上で並列・分散型の画像処理アルゴリズムを記述するための枠組を構築し、容易に実時間並列動画像処理アプリケーションを作成できるRPV(Real-time Parallel Vision) の実現を目指している。最近のPC の処理能力の向上と価格の低下はめざましく、複数のPC をネットワークで結合したPC クラスタを利用することにより、性能、コストの両面ですぐれた並列分散システムを構築することができ、カメラの台数を増やす場合も、PC の台数を増やすことで容易に対応できる。

PC クラスタのような分散環境で実時間の画像処理を行う場合、PC 間の同期が問題となる。そこで、RPV では、3 種類の同期機構 (前向き同期、バリア同期、後向き同期) を実現し、PC 間の同期を保証している。次に、本システム上で、実時間動画像処理アプリケーションを容易に作成できるプログラミングツールを構築する。このツールにより、ユーザは同期の問題や、実際に通信を行う方法について考慮する必要なく、容易にRPV を用いた動画像処理アプリケーションを作成することができる。

本報告では、RPV の概要について述べ、RPV 上での並列分散動画像処理プログラミングを容易にするプログラミングツール、このツールを用いたサンプルプログラムを示す。

2 ハードウエア構成

本研究で用いるPC クラスタは、14 台のノードPC からなっている。各PC はPentiumIII を2 個搭載しているAT 互換機であり、OS にはLinux を用いている。各PC は高速ネットワークMyrinet で相互に結合されている (図1)。Myrinet は、アメリカのMyricom 社が開発したスイッチングハブを中心としたスター型の構成をとるギガビットLAN の一種である。Myrinet 上での高速通信のためにRWCP によって開発されたPM 通信ライブラリを用いている [2]。また、6 台のCCD カメラがビデオキャプチャボード[1]によりノードPC に接続されている。

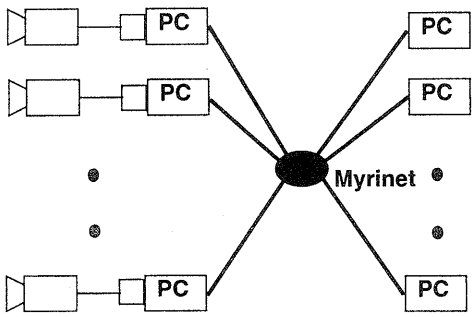

図 1: ハードウェア構成

3 RPV における同期処理

3.1 処理中の同期処理

動画像処理では、実時間で画像データを受信、処理、送信することが要求される。RPV で用いるPC クラスタのような環境で、これらのデータ転送、処理を実現するためには、PC 間の同期をとることが必要である。そこでRPV では、以下のような3 種類の同期機構を提供する [3][4][5]。

- 前向き同期

  PC に対して、各フレームの処理開始のタイミングを知らせる同期。この同期により、全てのPC が同じタイミングで処理を開始することになる。また、タイミング通知時に、処理送れ、データ未受信といったエラーの検出を行う。エラーが検出された場合、エラー処理を実行する。このエラー処理はユーザが選択することができる。

  エラー処理
  - データ落ち型

    処理を続行し、その間に受信するデータは破棄し、最新の入力データのみを使用する。
  - 不完全データ転送型

    処理を中止し、代替データを出力する。代替データとして、(1) 処理途中結果、(2) 前フレームの結果、(3) あるバッファ

[1] Imaging Technology 社の IC-PCI

に書き込まれたユーザ定義のデータのいずれかを選択できる。

- 完全保持型

  処理を続行し、その間に受信したデータはキューイングする。

- バリア同期

  複数のPCからのデータを受信するPCがある場合、そのPCは、同時刻に処理されるべき全てのデータの到着を待って、処理を開始しなければならない。このための同期がバリア同期である。

  あるPCで、過去数フレームの入力を用いて処理を行う場合、過去のデータを、どう揃えるかという時間軸のバリア同期もある。このバリア同期のレベルとして、以下をユーザが選択できる。

  - 途中、データ落ちが起こっている場合は、そのデータが関係する処理は全く行わない。
  - 途中、データ落ちが起こっている場合は、前のデータを代用し、データを揃えて処理を行う。

- 後向き同期

  無駄な処理を回避するために、他のPCに対してデータ落ちを通知し、同じフレームのデータに関する処理を中止するための同期。この同期は、データ落ちを最小限におさえる役割を持つ。

3.2 その他の同期処理

- システム立ち上げ時

  全てのPCの準備できたことをひとつのPCで確認し、準備ができ次第、全てのPCへ処理開始時刻を通知する。処理開始時刻に達すると、全PCが処理を開始する。そのためには、カメラ、PCで時刻の管理をすることが重要である。RPVでは、以下のように時刻の管理を行っている。

  - すべてのカメラは外部同期信号を与えることによって、完全に同期されている。

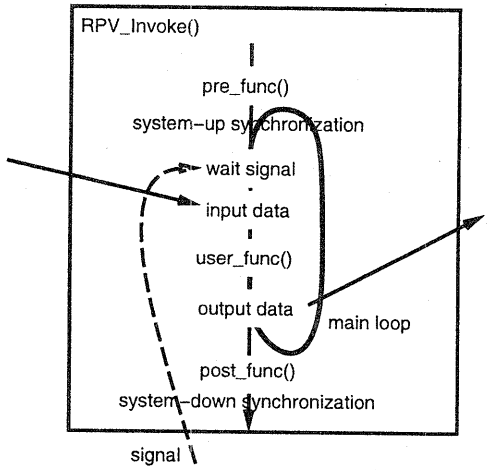

図 2: 関数 RPV_Invoke の動き

  - カメラが接続されたすべてのPCで同時刻に画像キャプチャを開始する。このためにNTPを導入し、PCの内部時計を一致させている。

- システム終了時

  全てのPCの処理が終了したことを確認し、終了処理に入る。

4 RPVにおけるプログラミング

この節では、RPVを利用した実時間並列動画像処理アプリケーションのプログラミングについて述べる。RPVは、C++上のクラスライブラリとして実装され、各PC毎にユーザ関数を起動する関数や、PC間のデータフロー情報をもつクラスなどを提供する。

このRPVは、SPMD(Single Program Multiple Data)モデルであり、一つのプログラムが各PC上で実行され、それぞれユーザが指定した画像処理関数がデータ処理を行う。このため、ユーザは、データがPC間をどのように流れ、各PCでどういう処理を施すかということを把握することが必要であるが、実際のデータ転送や、PC間の同期はRPVによって自動的になされる。すなわち、ユーザは、RPVによって、同期の問題や、実際に通信を行う方法について考慮する必要なく、

- PC間のデータフロー情報
- 各PCで行う画像処理アルゴリズム

```
void RPV_Invoke(
    RPV_Connection* connect,                          // データフロー情報
    struct RPV_FSM sync_mode,                         // 同期モード
    int frame_num,                                    // 処理するフレーム数
    void* (*pre_func)(void*),                         // ループ前に実行される前処理関数
    void* pre_func_arg,                               // pre_func の引数
    void* (*user_func)(RPV_Input*, RPV_Output*,
                       RPV_Asynch*, void*),           // ループで実行される関数
    void* user_func_arg,                              // user_func の引数
    void* (*post_func)(void*),                        // ループを出た後に実行される後処理関数
    void* post_func_arg                               // post_func の引数
);
```

図 3: 関数 RPV_Invoke

```
class RPV_Input{
    void*** data_ptr;       入力データへのポインタ
    int input_PC_num;       入力先の PC の数
    int* input_PC;          入力先の PC 番号
    int frame_num;          使用する入力データの数
    int* frame_no;          データのフレーム番号
    int* data_size;         入力データのサイズ
};
```

(a) クラス RPV_Input

```
class RPV_Output{
    void** data_ptr;        出力先へのポインタ
    int output_PC_num;      出力先の PC の数
    int* output_PC;         出力先の PC 番号
    int* data_size;         出力データのサイズ
};
```

(b) クラス RPV_Output

図 4: 関数 RPV_Invoke の引数 (一部)

を記述するだけで、実時間動画像処理アプリケーションを作成することができる。

4.1 各 PC で行う画像処理アルゴリズムの記述

各ノード PC で、ユーザが記述した画像処理関数を起動するために関数 RPV_Invoke を用いる。この関数 RPV_Invoke の中では、前段の PC から次々と送られてくるデータ、また場合によっては、後段の PC からのフィードバックなどの非同期データを入力データとしてユーザ関数へ渡し、ユーザ関数を実行後、出力される処理結果を次の PC へと送信する。具体的には、図 2のように関数 RPV_Invoke の内部で、第一引数のデータフロー情報から、データ送受信などの準備を行い、前処理関数 pre_func を実行する。その後、全ての PC で pre_func が終了したことを確認し、メインループに入る。メインループ内では、前向き同期による処理開始シグナルを契機として、前段の PC からの入力データと次段の PC へ出力するデータの書き込み先のポインタを引数として、ユーザ関数 user_func の実行し、次段の PC へ処理結果データを転送する、といった処理を繰り返す。このメインループ内での処理で、処理遅れなどのエラーが検出された場合、エラー処理が自動的に実行され、通常の処理へと戻る。frame_num フレーム分のデータを処理すると、メインループを抜け、後処理関数 post_func を実行、全ての PC の処理が終了したことを確認し、RPV 全体の処理を終了する。この関数 RPV_Invoke の仕様を図 3に示す。

また、第 2 引数の sync_mode で、同期機構に関する選択を行う。この構造体により、処理遅れなどのエラーが生じた際のエラー処理の種類、バリア同期のレベル、エラー処理で代替データの提供を選択した場合の代替データを選択する。この同期機構の選択により、ユーザは処理に適した同期機構を選択することができる (3.1節参照)。

ユーザ関数は、入力データ情報 (RPV_Input)、出力データ情報 (RPV_Output)、非同期データ情報 (RPV_Asynch)、ユーザ定義の引数 (user_func_arg) の四つの引数を持つ関数として定義される (図 4)。このユーザ関数内では、データの送受信や同期について考慮する必要がなく、静止画像を入力とし、それを処理し、結果を出力する静止画像処理と同じ形式である。そのため、動画像処理ということを意識することなく、容易にユーザ関数をプログラミングできる。

```
class RPV_Connection{
    int myPC_no;              自 PCno
    char* keyword;            処理のキーワード
    int input_PC_num;         受信データの数
    int* input_PC;            input_PC から受信
    int* input_data_size;     受信データのサイズ
    int input_frame_num;      1 回処理に使用するフレーム数
    int output_PC_num;        送信データの数
    int* output_PC;           output_PC へ送信
    int* output_data_size;    送信データのサイズ
    int asynch_PC_num;        非同期データを受ける数
    int* asynch_PC;           asynch_PC から非同期データを受ける
    int* asynch_data_size;    非同期データサイズ
    int asynch_data_num;      1 回の処理に使用する非同期データフレーム数
    int connect_PC_num;       使用する PC の数
    int* connect_PC;          使用する PC 番号
};
```

図 5: クラス RPV_Connection

4.2 PC 間のデータフロー情報の記述

クラス RPV_Connection は、PC 間のデータフロー情報を持つクラスである。このクラスは、PC 間のデータフロー情報を記述したファイルによって初期化される。各 PC は、このクラスの持つ情報をもとに、PC クラスタ内でのデータの送受信を行う。また、クラス RPV_Connection のメンバ **keyword** により、各 PC で行う処理を指定できる。

データフロー情報はファイルにまとめて記述されるため、ユーザはプログラムの修正、再コンパイルを行うことなく、利用する PC と各 PC で行う処理を変更することができる。

5 サンプルプログラム

ここでは、RPV プログラミングツールを用いたサンプルプログラムを示す。図 6に、サンプルシステムの概要を示す。このシステムは、12 個のカラーマーカをつけた人間の動画像から、マーカの 3 次元位置を計算し、結果を表示する処理を実時間で行うシステムである。この例では、PC0、1、2 で 3 台のカメラから動画像の獲得、平滑化を行い、PC3、4、5 でマーカの 2 次元位置を計算し、PC6 でマーカの 3 次元位置を計算し、PC7 でアニメーションを作成している。

図 7は、クラス RPV_Connection を初期化するデータフロー情報を記述したファイルである。このファイル内に、データフローに関する情報を記述し、このファイルの内容をもとに、実際のデータ通信がなされる。

図 8が全ての PC で起動されるメイン関数である。このメイン関数内で、**keyword** にしたがって適切なユーザ関数を引数とする RPV_Invoke を起動している。このようにユーザは、一つのプログラムを各 PC で起動すると、RPV は、ユーザが指定した **keyword** にしたがって、適切なユーザ関数を起動する。

各 PC で起動する関数は、図 9のように定義する。このユーザ関数 **Calculate2D** では、640 × 480 pixel のカラー画像と、後段の PC からのフィードバックを入力として、平面画像上におけるマーカの 2 次元位置を計算し、その座標を出力している。このように、ユーザは、実際のデータ転送や動画像処理を考慮する必要なく、各 PC で処理される画像処理アルゴリズムを記述することができる。

6 おわりに

本報告では、実時間並列画像処理アプリケーションを容易に作成するための、RPV について示した。RPV を用いると、ユーザは、データ転送や同期、エラー処理と言った分散システム上で処理を行う際の問題に悩まされることなく、データフローと画像処理アルゴリズムを記述するだけでプログラムを作成することができる。また、データフローの情報と、処理アルゴリズムは完全に分離されており、ユーザは、互いの影響について注意する必要がなく、簡単にデータフローや処理アルゴリズムの変更が可能である。

今後は、RPV プログラミングツールを用いたア

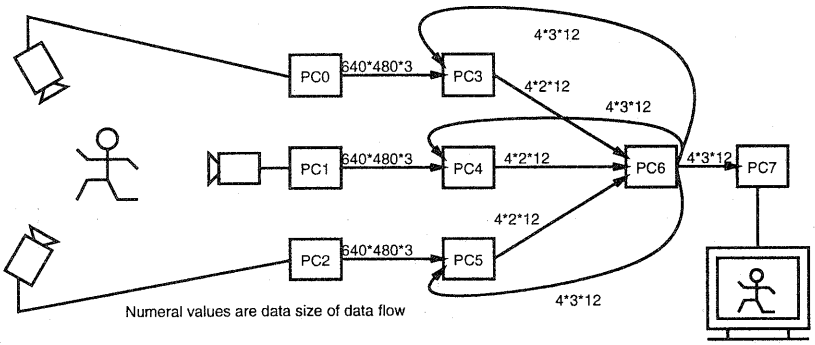

図 6: サンプルシステム

#PCno	keyword	i_PC	i_size	i_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	-	-	-	-	-

図 7: データフロー情報の記述ファイル： 各 PCno に対し、処理関数の keyword、入力元 PC(i_PC)、入力データサイズ (i_size)、1 回の処理に使用するフレーム数 (i_num)、出力先 PC(o_PC)、出力データサイズ (o_size)、非同期データ入力元 PC(a_PC)、非同期入力データサイズ (a_size)、1 回の処理に使用する非同期入力データフレーム数 (a_num) が記述される。

プリケーションの開発と評価を行っていく。

謝辞

本研究は、日本学術振興会未来開拓学術研究推進事業「分散協調視覚による動的 3 次元状況理解」プロジェクト (JSPS-RFTF 96P00501) の補助を受けて行った。

参考文献

[1] 松山隆司. “分散協調視覚 –視覚・行動・コミュニケーション機能の統合による知能の創発–”. 画像の認識・理解シンポジウム *MIRU'98*, TP1-1, 1998.

[2] 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.

[3] 有田大作, 濱田義雄, 谷口倫一郎. “PC クラスタにおける実時間並列動画像処理の性能評価”. 情報処理学会研究会資料, CVIM115-15, 1999.

[4] Daisaku Arita, Naoyuki Tsuruta and Rin-ichiro Taniguchi. “Real-time parallel video image processing on PC-cluster”. *in Parallel and Distributed Methods for Image Processing II, Proceedings of SPIE*, Vol. 3452, pp.23–32, 1998.

[5] Daisaku Arita, Yoshio Hamada and Rin-ichiro 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.

```
 1  #include <stdio.h>
 2  #include <rpv.h>
 3
 4  int main(void)
 5  {
 6      FILE *fp;
 7      fp = fopen("connect.cnt", "r");
 8      RPV_Connection* connect = RPV_Init(fp);
 9      fclose(fp);
10
11      struct RPV_FSM miss_FSM;
12      miss_FSM.FSM = RPV_DATA_MISSING;
13
14      if (strcmp(connect->keyword, "smooth") == 0) {
15          RPV_Invoke(connect, miss_FSM, 0, NULL, NULL, &Smooth, NULL, NULL, NULL);
16      }
17      else if (strcmp(connect->keyword, "calc2D") == 0) {
18          ReadBackgroundArg read_background_arg(BACKGROUND_FILE_NAME);
19          RPV_Invoke(connect, miss_FSM, 0,
20                      &ReadBackground, &read_background_arg, &
21                      Calculate2D, &read_background_arg.background,
22                      NULL, NULL);
23      }
24      else if (strcmp(connect->keyword, "calc3D") == 0) {
25          ReadCalibrationArg read_calibration_arg(CALIBRATION_FILE_NAME);
26          RPV_Invoke(connect, miss_FSM, 0,
27                      &ReadCalibration, &read_calibration_arg,
28                      &Calculate3D, &read_calibration_arg.calibration_data,
29                      NULL, NULL);
30      }
31      else if (strcmp(connect->keyword, "display") == 0) {
32          RPV_Invoke(connect, miss_FSM, 0, NULL, NULL, &Display, NULL, NULL, NULL);
33      }
34      return 0;
35  }
36
```

図 8: サンプルプログラム (メイン関数)：8 行目の RPV_Init で、ファイルからデータフロー情報をクラス RPV_Connection に読み込む。その際、ファイル内に矛盾がないか検査も行う。14 行目から 33 行目にかけては、クラス RPV_Connection のメンバ keyword にしたがって適切なユーザ関数を引数とする RPV_Invoke を起動している。例えば、keyword が “calc2D” の場合、前処理関数 ReadBackgraoud と、画像処理関数 Calculatre2D が引数として渡される。

```
1  #include <fstream.h>
2  #include <rpv.h>
3
4  struct ReadBackgroundArg {
5      char* filename;
6      RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT> background;
7      ReadBackgroundArg(const char* s);
8  };
9
10 void* ReadBackground(void* a)
11 {
12     ReadBackgroundArg* arg = (ReadBackgroundArg*)a;
13
14     ifstream fs(arg->filename);
15     arg->background.Read(fs);
16     fs.close();
17
18     return NULL;
19 }
20
21 void* Calculate2D(const RPV_Input* id, RPV_Output* od, const RPV_Asynch* ad, void* a)
22 {
23     const RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>* i_data
24 = (const RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>*)id->data_ptr[0][0];
25     Positions2D<MARKER_NUM>* o_data
26       = (Positions2D<MARKER_NUM>*)od->data_ptr[0];
27     const Positions3D<MARKER_NUM>* a_data
28       = (const Positions3D<MARKER_NUM>*)ad->data_ptr[0][0];
29     const RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>* background
30 = (RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>*)a;
31
32     RPV_RGB24<IMAGE_WIDTH,IMAGE_HEIGHT>* sub_data
33       = Subtraction(i_data, background);
34
35     // Searching for markers from the subtracted image
36     // using previous 3D positions of markers
37     // for robustness against occlusion.
38     SearchMarker(sub_data, a_data, o_data);
39
40     return NULL;
41 }
42
```

図 9: サンプルプログラム (Calculate2D)： 10 行目から 19 行目までが前処理関数であり、21 行目以下の Calculate2D が画像処理関数である。23、24 行目で入力データ、26、27 行目で 出力先のバッファ、28 行目でフィードバックデータへのポインタをそれぞれ取得し、29、30 行目でユーザ引数を読み込んでいる。その後、背景差分を行った後、差分後の画像、フィードバックデータを入力として、マーカの 2 次元座標位置を計算している。