コンピュータビジョンと
イメージメディア 131－2
（2002．1．17）

複数カメラとPCクラスタを用いた実時間動画像処理における時刻管理と同期

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院知能システム学部門
{arita,rin}@limu.is.kyushu-u.ac.jp

概要 複数カメラとPCクラスタを用いて実時間画像処理を行う場合，それぞれのカメラとPCがどのように時刻を共有するかという問題がある．さらに，実時間処理を連続的に行わせるためにPC間での処理と通信の同期をとる必要がある．このような問題を解決するため，我々はプログラミング環境RPVを構築した．RPVは，すべてのカメラが外部同期信号によって同期されていることを前提とし，その同期信号をすべてのPCでも共有することにより，カメラとPCを同期させる機能を持っている．RPVを利用することにより，アプリケーションプログラマはデータフロー情報とデータ処理タスクを記述するだけで実時間画像処理アプリケーションを構築することが可能となる．

Time management and synchronization for real-time image processing using multiple cameras and PC cluster

Daisaku Arita and Rin-ichiro Taniguchi
Depertment of Intelligent Systems, Kyushu University

abstract For real-time image processing using multiple cameras and PC cluster, all cameras and PCs must share a clock. And for continuous real-time image processing, data processing and data transfer must be synchronized between all PCs. To solve these problems, we are construct RPV, a programming environment. RPV, which assumes that all cameras are synchronized by external synchronization signal, have a mechanism to synchronize all cameras and PCs by sharing the synchronization signal with all PCs. Using RPV, an application programmer can easily develop real-time image processing systems by describing only data flow information between PCs and data processing tasks.

1　はじめに

近年，コンピュータビジョンの分野では，複雑な対象や環境を理解し，それらとのインタラクションを行うために，複数のカメラから得られた多視点情報を実時間で処理する研究が盛んに行われている．このとき，多くのカメラを接続するためのI/O能力の向上や，膨大なデータを実時間で処理するための計算能力の向上が課題となる．この解決法として，複数の計算機をネットワークによって接続した分散並列計算機を利用することが考えられる．このような分散並列計算機では，計算機を増加させることにより必要なI/O能力と計算能力を得ることが可能である．このような背景から，我々は，汎用的なPCをネットワーク接続したPCクラスタ上での実時間画像処理に関する研究を行っている．汎用のPCを利用することにより，コストパフォーマンスが高く，柔軟性に富んだシステムを構築することが可能である．

従来より，数値計算やデータベースの世界では，ネットワーク接続された汎用のワークステーションやPCを分散型並列計算機として利用することが行われてきており，並列プログラムの記述性や移植性を高めるために，標準的なプログラミング環境も提案されている[1, 2]．また，画像処理に関しても静止画を対象とした分散処理システムもいくつか開発されている．しかし，実時間画像処理では，フレーム単位で次々と到着する画像データを滞りなく処理し，結果を出力することが要求されるため，その分散並列計算機上での実現は容易ではない．すなわち，実時間画像処理を分散並列計算機上で実現するため

には，各通信経路における1フレーム分のデータ転送，および，各ノードPCにおける1フレーム分のデータ処理が1フレーム時間（通常よく使われるカメラでは1/30秒）で終了することを保証しなければならない．このためには

- 高速かつ低負荷なデータ転送機構
- 1フレーム時間内にデータ転送とデータ処理が終了していることを保証する同期機構
- データ転送やデータ処理に遅れが生じたときに正常な状態に復帰するためのエラー処理機構

が必要である．また，応用システムを開発，維持する上では，これらの実時間処理機構の複雑なプログラミングも問題となってくる．本稿では，これらの機能を提供することにより実時間並列画像処理をPCクラスタ上で容易に開発できるプログラミング環境RPV（Real-time Parallel Vision）を提案し，その概要，実時間処理機能の実現方法，性能評価について述べる．

2 RPVの概要

2.1 システム構成

本研究で用いるPCクラスタシステム（**図1**）は，14台のノードPCからなっている．その内，6台にCCDカメラが接続されており，すべてのCCDカメラは同期信号発生装置により同期がとられている．各ノードPCはCPU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．また，各ノードPCはギガビットLANの一種であるMyrinetによって相互に結合されおり，Myrinet上の通信にはRWCPによって開発されたPM通信ライブラリ[3]を利用している．Myrinet上のPM通信ライブラリの特徴として，

- 7.5 μsの低レイテンシ
- 118MByte/秒の高スループット
- 全二重通信をサポート
- ゼロコピー通信をサポート
- 割込み受信をサポート

といった点が挙げられる．このデータからも分かるように，非圧縮フルサイズカラー画像（640×480, 1画素4バイト）を30fpsで転送するための条件，すなわち毎秒 $640 \times 480 \times 4 \times 30 = 36864000 \simeq 35$Mバイト以上のスループット，を十分満たしている．また，ゼロコピー通信と割込み受信のサポートにより，受信側のノードPCはCPUに負荷をかけることなくデータの受信を行うことが可能である．

2.2 並列処理方式

RPVでは，以下の並列処理方式を実現している．

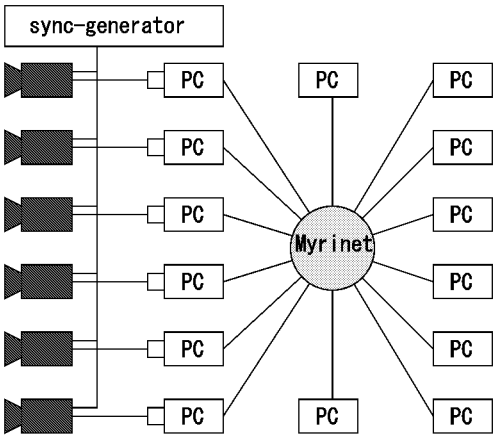


図 1: PCクラスタシステム

- パイプライン並列処理
  処理の段階ごとに1台の計算機を割り当てる並列処理であり，実時間画像処理では，一つのフレームに対する処理を各計算機で順々に行うという方式をよく用いる．この場合，ある時刻では，各計算機は異なったフレームに対する処理を並列に行っていることになる．従って，計算機台数を増やすと，並列に処理するフレーム数が増えるため，データ処理能力は向上するが，あるフレームが入力されて，その結果が出力されるまでに必要な時間は，計算機台数分のフレーム数だけ遅れることになりレイテンシーは増大してしまう．
- データ並列処理
  データを分割し複数の計算機で同時に処理する並列処理であり，実時間画像処理では以下の二つの方式が考えられる．
  - 空間方向データ並列処理
    各フレームのデータを分割して各計算機で処理を行う方式．画像を分割して並列に処理を行うなど，画像処理ではよく利用される並列処理である．また，フレームデータを分割するのではなく，複数のカメラによって同時に獲得されたデータに対して複数の計算機で同時に処理する方式も空間方向データ並列処理の一つである．
  - 時間方向データ並列処理
    フレームごとに異なる計算機で処理を行う方式．視体積交差法のような分割後の処理の負荷に偏りが生ずるデータ処理タスクや，画像領域分割のようにフレームデータの分割が難しいデータ処理タスクを並列化するときに利用できる．

- 機能並列処理
  同じフレームのデータに複数の計算機で同時に異なる処理を行う並列処理であり，原理的には，空間方向データ並列処理と同様の効果がある．ただし，同時に異なる処理を行わなければならないので，処理の内容によってはこの並列処理を実現できるとは限らない．また，計算機間で処理量が異なると，処理量の最も多い計算機の処理時間に，全体の処理時間が拘束され，計算機台数分の処理性能の向上が得られなくなってしまう．

2.3 プログラミングの概要

実時間並列画像処理アプリケーションを構築するとき，アプリケーションによって異なるのは，ノードPC間のデータフローと各ノードPCにおけるデータ処理タスクのみである．これら以外の機能，具体的には

- データ転送機構
- 同期機構
- エラー処理機構

についてはすべてのアプリケーションで共通に利用できる．これらの機能を実現するためには，ハードウェア，OS，ネットワークについての多くの知識が必要であり，プログラミングは容易ではない．そこで，RPVではこれらの機能をC++のクラスライブラリとして提供することで，アプリケーションを容易に記述できるようにしている．これにより，アプリケーションプログラマはデータフロー情報とデータ処理タスクを記述するだけでよいことになる．

2.3.1 ノードPC間のデータフロー情報の記述

図2で示したクラス `RPV_Connection` は，ノードPC間のデータフロー情報を持つクラスである．各ノードPCは，このクラスの持つ情報をもとに，ノードPC間のデータの送受信を行う．通常このクラスは，ノードPC間のデータフロー情報を記述したファイル（以下コネクションファイル）によって初期化される．このコネクションファイルにはPCクラスタ全体のデータフローを記述する．このようにデータフロー情報は一つのファイルにまとめて記述されるため，データフロー情報の管理が容易であるだけでなく，プログラムの修正，再コンパイルを行うことなく，利用するノードPCと各ノードPCで行う処理を変更することができる．コネクションファイルは，1行に1台のノードPCのデータフロー情報を記述し，各行には以下の項目が記述される．

PCno PCクラスタ内のノードPCに付けられた番号

keyword ノードPCで行うデータ処理を指定する文字列

i_PC 受信同期データの転送元ノードPC番号

i_size 受信同期データのサイズ

i_num 同時に使用する受信同期データの数

o_PC 送信データの転送先ノードPC番号

o_size 送信データサイズ

ai_PC 受信非同期データの転送元ノードPC番号

ai_size 受信非同期データのサイズ

ai_num 同時に使用する受信非同期データの数

2.3.2 各ノードPCでのデータ処理アルゴリズムの記述

RPVにおけるデータ処理は**図3**に示す流れで行われる．まず，初期化1でコネクションファイルの読み込みが行われ，クラス `RPV_Connection` が初期化される．つぎに，変数 `keyword` によって分岐を行い，ノードPCごとに引数の異なる関数 `RPV_Invoke` を起動する．このようにするのは，プログラムの修正，再コンパイルを行うことなくコネクションファイルを修正するだけで，各データ処理タスクを実行するノードPCを変更できるようにするためである．**図4**に示すのが関数 `RPV_Invoke` の仕様であり，関数 `RPV_Invoke` の引数によってそのノードPCで実行されるデータ処理タスクが指示される．また，第2引数の `sync_mode` で，同期機構に関する選択を行う（**3.4**参照）．

関数 `RPV_Invoke` の動作は以下の通りである．

1. 初期設定として，モジュールの起動（**3.1**参照），通信路の初期化を行う．
2. `RPV_Invoke` の引数にしたがって，指定された関数を起動しながら処理終了まで動作する．具体的には以下の動作を行う．
   1. 前処理関数 `pre_func` を実行する．
   2. 処理開始割り込み信号を待つ．
   3. ユーザ関数 `user_func` を実行し，1フレーム分のデータを処理する．
   4. `frame_num` フレーム分のデータを処理が済んでいなければ，(2) へ戻る．
   5. 後処理関数 `post_func` を実行し，処理を終了する．
3. すべてのノードPCで処理が終了したことを確認し，RPV全体の処理を終了する．

ユーザ関数 `user_func` の引数は，三つの入出力データへのポインタと一つのアプリケーションプログラマが自由に利用できる引数から成る．受信同期データ引数と受信非同期データ引数には，システムによって自動的に必要なデータが渡され，ユーザ関数はそれらのデータを読み込むことができる．ま

```
class RPV_Connection{
    int myPC_no;              自ノードPC番号
    char* keyword;            処理内容のキーワード
    int input_PC_num;         受信する同期データの数
    int* input_PC;            受信同期データの転送元ノードPC番号
    int* input_data_size;     受信する同期データのサイズ
    int input_frame_num;      同時に使用する受信同期データのフレーム数
    int input_buf_num;        受信同期データ用バッファの数
    int output_PC_num;        送信データの数
    int* output_PC;           送信データの転送先ノードPC番号
    int* output_data_size;    送信データのサイズ
    int output_buf_num;       送信バッファの数
    int ainput_PC_num;        受信非同期データの数
    int* ainput_PC;           受信非同期データの転送元ノードPC番号
    int* ainput_data_size;    受信非同期データのサイズ
    int ainput_data_num;      同時に使用する受信非同期データのフレーム数
    int connect_PC_num;       使用するノードPCの数
    int* connect_PC;          使用するノードPC番号列
};
```

図 2: クラス **RPV_Connection**

た，送信データ引数には，ユーザ関数から書き込み可能であり，書き込みが終了すると自動的にそれらのデータは送信される．このようにユーザ関数は，データの送受信や同期について考慮する必要がなく，1 枚の画像を入力とし，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記述可能である．そのため，実時間動画像処理ということをほとんど意識することなく，容易にユーザ関数をプログラミングできる．

3 RPV の実装

本節では，RPV におけるデータ転送機構，同期機構，エラー処理機構の実現方法について述べる．

3.1 実装の基本方式

並列動画像処理を効率的に実行するために，本システムの各ノード PC では，**図 5**に示す四つのモジュールが，UNIX プロセスとして並行動作している．これにより，データの受信，処理，送信を並行に行うことができ，外部からのデータの受信にも即座に対応できるようになっている．また，データの送受信のためのバッファは，複数のモジュールからアクセスできるように共有メモリ上に実現されている．以下，各モジュールの動作について説明する．

- データ受信モジュール（DRM）
  データ受信に関する情報のやりとりを行うモジュール．
  - 転送元ノード PC からの各フレーム分のデータ転送が終了した知らせを受け取り，データが書き込まれた受信バッファの領域を書き込み不可（読み込み可能）にする．

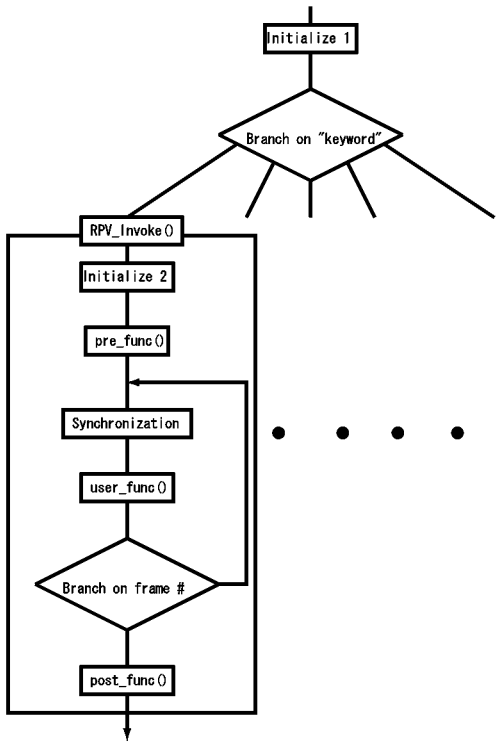


図 3: 処理の流れ

  - 処理が済んでデータ書き込み可能になった受信バッファの領域をチェックし，転送元ノード PC に伝える．
- データ処理モジュール（DPM）
  実際の画像処理を行うモジュール．
  - 最初に関数 **pre_func** を呼び出す．
  - 最後に関数 **post_func** を呼び出す．

```
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_Ainput*, void*),     ループ中に実行される関数
    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 4: 関数 `RPV_Invoke`

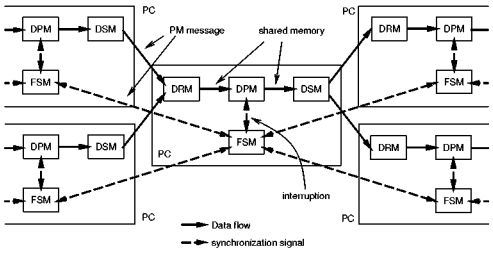


図 5: 各ノード PC のモジュール構成

- 処理開始の割り込み信号を受けとると，受信バッファの読み込み可能領域からデータを読み込んで画像処理を行う（実際は関数 **user_func** を呼び出す）．処理結果は送信バッファの書き込み可能領域に書き込み，その領域を書き込み不可とする．
- 処理が終了すると，当該の受信バッファ領域を書き込み可能にする．
- 処理結果の書き込みが終了すると，書き込みの終了を通知する割り込み信号を DSM に送る．

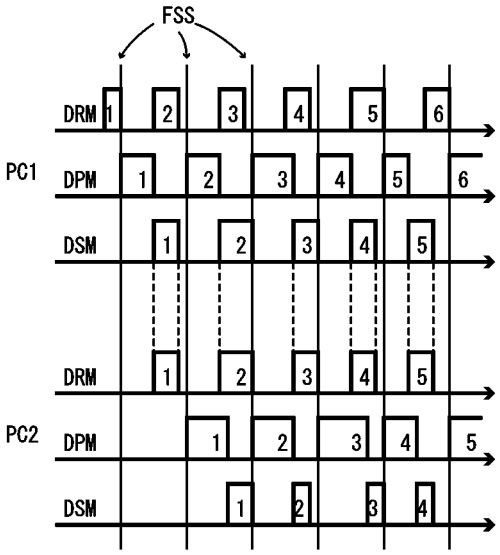
- データ送信モジュール（DSM）
  データ送信に関する情報のやりとりを行うモジュール．
  - 処理モジュールからの各フレーム分のデータ書き込みが終了した知らせを割り込み信号として受け取り，そのデータを転送先ノード PC に送信する．
  - 送信が終了したデータの格納されていた領域を書き込み可能とする．

- フレーム同期モジュール（FSM）
  ノード PC 間の同期をとるためのモジュール．
  - FSM どうしの通信によって，**3.3**で述べるフレーム同期信号（Frame Synchronization Signal:FSS）をデータの流れにそって上流から下流へ向けて送受信する．
  - FSS を受け取ると，処理開始を通知する割り込み信号を DPM に送る．

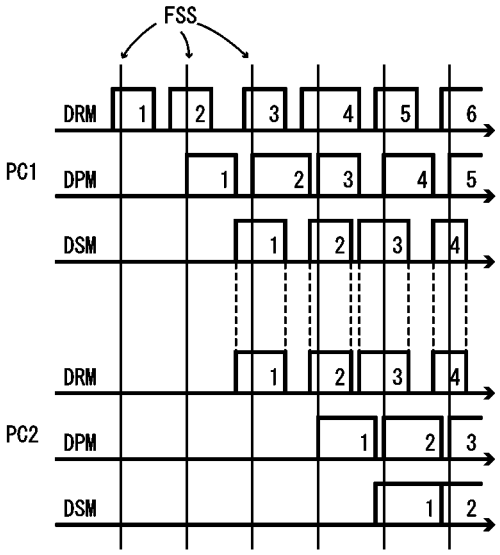
3.2 ノード PC における処理の並列化

システム全体としての性能を向上させるためには，各ノード PC での性能を向上させる必要がある．そこで，ここでは，ノード PC 内のモジュールの並列化について考察する．**3.1**で述べたモジュールのうち CPU を多く利用するものは，DPM 以外に DSM がある．これは PM 通信ライブラリではデータ転送の単位が最大 8K バイトのため，画像のような大きなデータは分割して送信する必要があり，その制御に CPU が必要なためである．その他のモジュールはほとんど CPU を利用しない．また，Myrinet は全二重通信可能であるため，データ受信とデータ送信が競合することもない．したがって，データの転送，処理に直接関与するモジュールである DRM, DPM, DSM での処理は，一般に**図 6（a）**のようになる．この図で，上半分と下半分がそれぞれ一つのノード PC の動作を表しており，それぞれのノード PC の中では，上から順に DRM におけるデータ受信時間，DPM におけるデータ処理時間，DSM におけるデータ送信時間を表している．また，上段のノード PC から下段のノード PC へと同期データが転送されている．この図のように，DPM によるデータ処理の終了後，DSM によるデータ送信が行われるという排他的な処理になる．

このとき画像の実時間処理を実現するためには，DPM によるデータ処理時間と DSM によるデータ送信時間の合計が 1 フレーム時間以内となることが必要となる．しかし，1 台のノード PC に 2 個の CPU を搭載すれば，DPM と DSM を並列動作させることが可能となり，**図 6（b）**のように，t 番目のフレームのデータ処理と $t-1$ 番目のフレーム



(a) Without overlaping DPM and DSM



(b) With overlaping DPM and DSM

図 6: モジュールのタイムチャート

のデータ送信をオーバラップさせることが可能となる．つまり，1 フレーム時間内で画像データの処理に CPU を多く割り当てられるようになる．このような点を考慮し，本研究の PC クラスタでは各ノード PC に 2 個の CPU を搭載している．ただし，データ処理とデータ送信をオーバラップさせない場合に比べてオーバラップさせる場合は，1 フレーム時間あたりの処理可能量が増加する（スループットが増大する）かわりにレイテンシが 2 倍になってしまう．したがって，どちらの方式を採用するかをアプリケーションの計算量やノード PC の台数などを考慮してアプリケーションプログラマが選択できるようになっている．

3.3 時刻管理

分散並列計算機環境において実時間画像処理を行う場合，時刻の管理が重要である．これは，複数のカメラによって撮影された画像が同一時刻のものでなければならず，また各計算機でのデータの処理と計算機間の同期データの転送を同じ時間間隔で行わなければならないからである．このために，以下のことを行っている．

- すべてのカメラに外部同期信号を与えることによって，すべてのカメラを完全に同期させる．これにより，まったく同じ時刻の画像を撮影することができる．
- カメラが接続された各ノード PC で同時刻に画像獲得を開始する．時刻を合わせるために NTP[4] を導入し，ノード PC の内部時計を一致させている．NTP を使うことによって，数ミリ秒以下の精度でノード PC の時刻を合わせることができるので，33 ミリ秒ごとの画像獲得に対しては十分な精度である．また，データにはフレーム番号が付けられており，これを比較することにより，異なるカメラからの画像の獲得タイミングが同じであるかどうかを判断できる．
- カメラが接続されたノード PC は，カメラからの垂直同期信号のタイミングに合わせて，FSS を発生する．FSS は同期データの流れに沿って上流のノード PC から下流のノード PC へと転送される．この FSS によって各ノード PC 間のデータ処理とデータ転送の同期をとることができる（同期機構の詳細は **3.4**を参照）．

3.4 同期機構

本システムでは，実時間でデータを受信，処理，送信することが要求される．これらのデータ転送，処理を滞りなく実行するために，ノード PC 間の同期をとることが必要である．そこで，本システムでは，

- データ転送同期
- データ処理同期

の 2 種類の同期機構を提供し，同期の実現のために FSS を利用している．

データ転送同期は，DPM に同期データが到着しているかどうかをチェックする同期である．この同期により，前段のノード PC からの同期データ転送の遅れや前段までのノード PC における同期データの

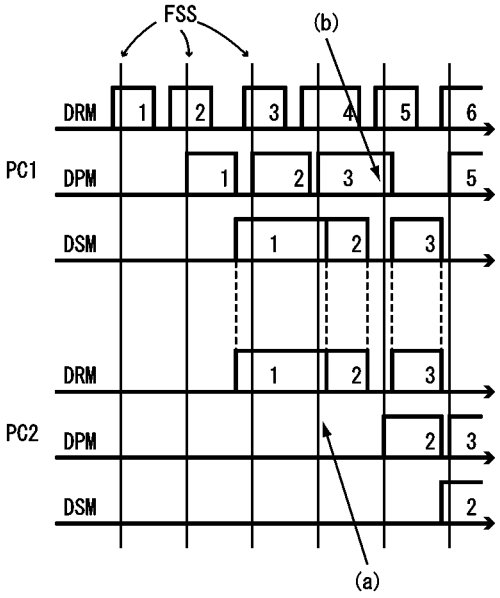


図 7: データ転送同期とデータ処理同期

データ落ちを検出することができる．**図 7**は連続した二つのノードPCのタイムチャートである．DRMが受け取る同期データは，正常時は二つのFSSの間に受信される．そのため，FSSを受信したときにDPMは処理を開始することができる．しかし，(a)でデータ転送の遅れが起こっており，処理開始時に同期データが到着していない．このため，DPMは処理を開始することができず，プログラマによって指定されたエラー処理が実行されることになる．

一方，データ処理同期は，DPMの処理が1フレーム時間以内に終了しているかどうかをチェックする同期である．**図 7**において，(b) でデータ処理の遅れが起こっており，処理開始時になっても前のフレームのデータ処理が終了していない．このため，そのままではDPMは処理を開始できず，また，DSMはデータを送信することができない．この場合もプログラマが指定したエラー処理が実行されることになる．

なお，本システムでは，ここで述べた同期機構に基づいた同期データ転送以外に，非同期データ転送もサポートしている．非同期データ転送は，フレームタイミングとは関係なくデータを転送するものであり，データは1フレームに何回も転送されても，あるいはまったく転送されなくてもよい．通常は，フィードバックや協調処理のための制御信号などに利用される．アプリケーションプログラマは関数 `user_func` 中の任意のタイミングで最新の非同期データを転送することができる．

3.5 エラー処理機構

2種類の同期機構におけるエラー処理として，以下の方法が考えられる．

(a) データ転送の遅れ

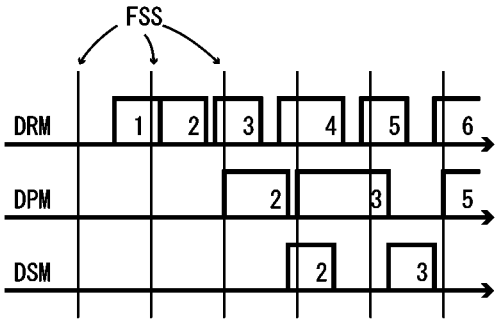
1. データが到着するのを待って，処理を開始する．
2. 次のFSSまで処理を行わない．次の処理では最新のデータを処理する．

(b) データ処理の遅れ

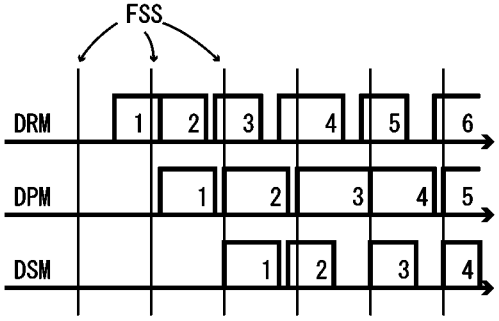
1. 処理を打ち切り，次のフレームのデータに対する処理を開始する．
2. 終了するまで処理を継続する．次の処理では最新のデータを処理する．

これらのエラー処理の組み合わせにより，以下の3種類のエラー処理モードを用意し，アプリケーションプログラマはその中から適切なものを選択することができる．なお，(a) の (2) は次のフレームのデータに対する処理を行うために現在のフレームのデータを落とすエラー処理であり，(b) の (1) は次のフレームのデータに対する処理を行うために現在のフレームのデータに対する処理を途中で打ち切るエラー処理であるから，これらのエラー処理の組み合わせは効果が重複しており，無意味なので考えない．

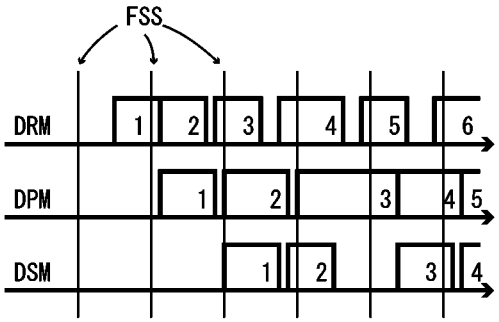
- データ落ち型（Data missing）
  (a) は (2)，(b) は (2) の組み合わせ（**図 8 (a)** 参照）．完全なデータだけを出力するが，エラーのときはデータ落ちが起こる．この方式は，処理が遅れたときに決められた時間以内に出力が得られないことになるので，厳密には実時間システムとは言えなくなる．しかし，データを落とすことによってすぐに遅れの状態から正常な状態に復帰できるので，実際の動画像処理としては最も利用される方式である．
- 不完全データ転送型（Incomplete data）
  (a) は (1)，(b) は (1) の組み合わせ（**図 8 (b)** 参照）．データ落ちは起こらないが，データ処理が遅れたときは不完全なデータが出力される．この方式のみが厳密な意味での実時間処理となる．ただし，計算機の処理能力と比べて処理量が多くなりすぎると，完全な出力がまったく得られなくなってしまうので，プログラマが負荷分散を慎重に行う必要がある．なお，このとき出力されるデータは，以下の中からアプリケーションプログラマが選択する．
  1. 処理途中のデータ
  2. 前フレームの処理結果
  3. あらかじめ指定されたデータ



(a) Data missing



(b) Incomplete data



(c) Complete queuing

図 8: 同期におけるエラー処理方式

- 完全保持型（Complete queuing）
  (a) は (1)，(b) は (2) の組み合わせ（**図 8 (c)** 参照）．データ落ちも起こらず，完全なデータだけを出力する．しかし，この方式は，一度エラーが起こるとそれ以降の実時間性の保証ができず，さらに最悪の場合は受信バッファが溢れてしまう．

4 おわりに

本論文では，実時間並列画像処理アプリケーションを作成するためのプログラミング環境である RPV について述べた．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実時間処理を行う際の問題に悩まされることなく，データフロー情報とデータ処理アルゴリズムを記述するだけでプログラミングを行うことができる．なお，RPV を利用したアプリケーションとして実時間モーションキャプチャシステム [5] を構築し，RPV の実用性も確認している．

今後の課題としては

- ノード PC とネットワークについての負荷分析を利用した半自動負荷分散ツールの構築
- オーバヘッドの解析によるシステム性能の向上
- RPV プログラミングツールを用いた様々なアプリケーションの開発

が挙げられる．

参考文献

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, “PVM: parallel virtual machine — A users’ guide and tutorial for networked parallel computing,” The MIT Press, 1994.

[2] Message Passing Interface Forum, “MPI: A message-passing interface standard”, Int. J. Supercomputer Applications and High Performance Computing, vol.8, no.3, pp.159–416, 1994.

[3] H. Tezuka, A. Hori, Y. Ishikawa, M. Sato, “PM: an operating system coordinated high performance communication library,” High-Performance Computing and Networking, P. Sloot and B. Hertzberger, Springer-Verlag, pp.708–717, 1997.

[4] David L. Mills, “Internet time synchronization: the network time protocol,” IEEE Trans. Communications, vol.39, no.10, pp.1482–1493, 1991.

[5] 米元 聡, 有田 大作, 谷口 倫一郎, “多視点動画像処理による実時間全身モーションキャプチャシステム －視覚に基づく仮想世界とのインタラクション－,” 映像情報メディア学会誌, vol.53, no.3, pp.409–416, 2000.