

同期型タプル通信を用いたマルチユーザ PlayStation ゲームシステム

仲宗根 雅臣 河野 真治
琉球大学工学部情報工学科

マルチユーザゲームではリアルタイム性とネットワーク上の分散処理が同時に要求される。この性質の違う2つの要求をタプル通信を画面書き換えに同期した形で行なう同期型タプル通信を用いて実現する手法を提案する。本研究では、PlayStationのネットやろうぜシステムとUNIX BSD/OS上に同期型タプル通信を実装しネットワークゲームの枠組を作成した。いくつかのネットワークゲームを実装し、その通信時間について測定を行なった。

Multiuser PlayStation Game System with Synchronous Tuple Communication

Masaomi Nakasone, Shinji Kono
Faculty of Information Engineering, University of the Ryukyus

Both real-time and distributed programming are required for multi-user game. Linda type tuple communication is extended with synchronus communication based on video refresh period. This synchronous tuple communication is implemented on "Net-Yarouze" PlayStation development system and BSD/OS as a framework for network game programming. We implement several network games and evaluate its communication time.

1 はじめに

本論文ではまず、PlayStationの概要を説明し、PlayStationによる通信に割り当てられる時間を調べる。次に同期型タプル通信によるリアルタイム処理を提案し、同期型タプル通信のAPIの詳細を記述する。次にPlayStationとUNIX(BSD/OS)上の実装について述べ、最後に、通信時間についての評価を行なう。

サ(GPU)がCPUに直結されている。PlayStationには2Mバイトのメインメモリ搭載されており、このアプリケーション領域にプログラムコードとデータ、画像データ(2次元・3次元)、サウンドデータの全てを収める必要がある。

PlayStationには、ルートカウンタと呼ばれるR3000-33Mhzのクロックの8分周をカウントする1本の16ビットカウンタが存在する。これにより、割り込みなどのソフトウェア操作とは無関係に、時間制限やタイミングの調整など、ゲームプログラミングに必要なカウンタ機能を実現している。本研究の通信時間測定にも、このカウンタを用いている。

2 PlayStationの概要

2.1 PlayStationのシステム構成

PlayStationは、R3000のCPUを中心にグラフィック・サウンドプロセッサとそのバッファや通信デバイス、CD-ROM・周辺デバイス、OS ROMがバスにつながれており、グラフィックデータ生成プロセッ

2.2 PlayStationのプログラミング方法

この章では、一般的なPlayStationでのプログラミングの方法やその手段であるネットやろうぜシステ

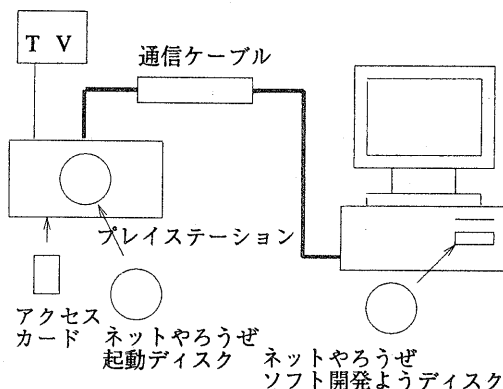


図 1: ネットやろうぜシステム図

ムについて説明する。

2.2.1 ネットやろうぜシステム

「ネットやろうぜ」は、図 1 のようにホストマシン (PC/AT 互換機) と PlayStation を専用ケーブルでシリアル接続することにより、プログラムの開発からデバッグ、テストランまでの一貫してホストマシン上で行えるシステムである。また、「ネットやろうぜ」システムでは、アプリケーションに必要な各種データはすべて既存のフォーマットからコンバートすることにより作成でき、またそれらのデータはすべてネットやろうぜライブラリを通じて直接扱えるようになっている。

2.2.2 ネットやろうぜ PlayStation OS

ネットやろうぜ PlayStation OS は、PlayStation の CPU として搭載されている R3000 をターゲットとしたオペレーティングシステムで、その上で C 言語によりゲーム開発を行なうことができる。ネットやろうぜ PlayStation OS では、専用ハードウェア (GPU) による処理と CPU によるプログラム実行を並列におこなうことができる。

2.2.3 グラフィックシステム

PlayStation は、GPU (Graphic Process Unit) と呼ばれる、グラフィック描画処理プロセッサが搭載されていて、CPU とは、非同期で動作する事ができる。しかし、ゲームプログラムでは同期を合わせないと

いけなくなる状況のために同期機構も用意されている。この GPU は、画面表示のための CRTC 機能、3D 座標変換機能、フレームバッファに対するポリゴンの高速描画機能を備えている。

GPU には、メインメモリとは別にフレームバッファと呼ばれる 1M バイトのメモリを持っており、このフレームバッファは 16 ビットピクセルを単位とした 1024x512 の 2 次元のアドレス空間を構成しており、CPU からの直接アクセスはできない。フレームバッファとメインメモリ間は DMA (Direct Memory Access) によりデータ転送が行なわれる。

GPU は、フレームバッファの任意の領域の内容を、そのまま CRT ディスプレイ上にする。この領域を表示エリアと呼び、10 種の画面モードをサポートしている。

2.3 PlayStation での通信時間

まず、はじめに PlayStation での通信時間のハードウェアやプログラミング上での制限と可能性について説明を行う。

三次元グラフィックスを用いて画像を書く問題点は時間がかかることである。これを解決するために画像の生成を、描画命令リスト生成とポリゴン描画に分割・並列化する。

1. メモリ上に置かれたデータから、描画命令リスト (Z ソートリスト) 構造を生成する。
2. 描画命令リストが GPU に送られ、GPU はフレームバッファ上にポリゴンを描画する。
3. フレームバッファ上の描画イメージが画面に表示される。

以上が PlayStation におけるグラフィック処理の基本的流れで、PlayStation では、上記の 3 つが同時に行われる。

1. メインメモリ上に格納されたプログラムに従って実行する CPU
2. DMA コントローラと呼ばれるデータ転送専用ハードウェア
3. 画像表示を担当する GPU

の 3 つがそれぞれ並列的に動作する。CPU (プログラム) は、DMA と GPU の状態には関係なく、データ転送開始アドレスや表示位置などのデータを専用ハードウェアに転送できる (ノンブロック関数を呼ぶ)。その結果、それぞれのハードウェアが高度に並列に実

行でき、CPU のほとんどの時間を描画命令リスト生成に費すことができる。

メインメモリ上に生成されるリスト構造とフレームバッファ上の画像イメージは、それぞれ2組用意されており、ダブルバッファを構成する。前者の方はCPU が生成中のものと転送中のもの、後者は、GPU が描画中のものと表示中のものとなる。1枚の画像を表示している時間次の画像の用意にあてておき、万一、1枚分の表示時間で次の画像が用意できなくても、前と同じ画像が続くので、画面表示としては継続できる。

TV 受信機は1秒間に30または60枚の画像を表示するが、時間的制限が存在し、表示画像の切替えはいつでも自由に切替えることはできない。画面の表示中のデータを他のデータに切替えられると、画像の違いやノイズとしてでてきてしまうので、切替えは画面への表示が行われていない状態で実行しなくてはならない。画面には見えていてさらに、画面への表示が行われていない状態が必要となる。この条件を満たしている時間を垂直帰線期間と呼ばれており、1秒に60回、一定間隔でやってきて、その始まりは垂直同期割り込みとしてCPU に通知される。

垂直同期割り込みが発生すると、コントローラ(ユーザが手にするパッド)の状態がチェックされる。また、垂直同期コールバック関数が定義されている場合、このコールバック関数も実行される。

よって、PlayStation の一般的な処理の流れは以下のようになる。まず初期化の後に、描画命令リスト生成を開始する。CPU は、画面切替えの前に、自分が行っているリスト生成の終了のほかに転送の終了と垂直同期割り込みの3つを待つ。その後、GPU に新しい表示描画と DMA コントローラに新しいリストのアドレスをそれぞれ設定して画面を切替える。そして、これらが完了したならばコントローラの状態を読み、それを元に CPU は次の画像のための描画命令リスト生成を開始する。図2に PlayStation での描画時間の仕組みの一例を示す。プログラムでの計算時間とグラフィック転送時間とグラフィック描画時間とで3重化されているが、画面のちらつきを防ぐために描画の終了は待たなくてはならない。この描画の終了を待つための時間(空いている時間)を通信に割り当てることにより、PlayStation のゲーム上では遅延をなくすことができる。

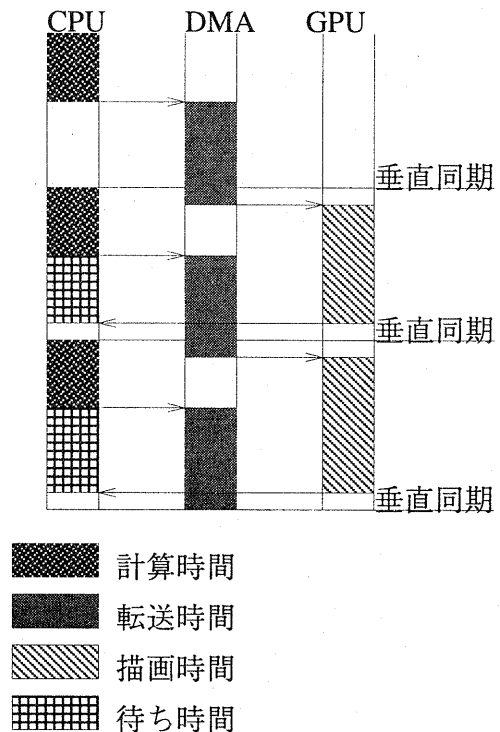


図 2: PlayStation での描画時間の関係

2.4 通常の PlayStation のプログラムの構造

初期化ルーチン

```
while(1){  
    パッド等、入力データ受け取り  
    foreach ゲームオブジェクト {  
        ゲーム上のオブジェクトの状態計算  
        描画登録処理  
    }  
    画面切替え操作  
    登録描画処理実行開始  
}
```

}

以上が、通常の PlayStation のプログラムの構造となる。ゲームオブジェクトの数の分オブジェクトの計算しその結果によって描画登録処理を行なう。その後画面切替え操作と登録描画処理実行開始を行ない while loop で戻りパッド等の入力を受け付ける。しかし、while(1) でのループは 1 秒間に 60 回まわるように計算量を抑える必要がある。

3 同期型タプル通信によるリアルタイム処理の提案

しかし、通信を含むゲームの場合は、通信要求したデータが 1/60 秒で返る保証がない。そこでデータ読み込みそこで要求とデータ読み込みを分離して 1/60 秒以内でなくても通信データを次の垂直同期以降でも受け取れるようにする。これを Linda 型のタプル通信を組み合わせ、PlayStation のゲーム処理(垂直同期)と同期した同期型タプル通信を PlayStation に適したネットワークゲーム開発環境として提案する。

3.1 Linda について

Linda とは、並列プログラミングライブラリで C 言語をベースにした C-Linda や Fortran をベースにした Fortran-Linda などがある。これにより C や Fortran などの言語から容易に Linda を用いてコーディングすることが出来る。これによって幅広いプラットフォームの上で、簡単に並列プログラムを作成することが出来る。

Linda では、サーバにタプルと呼ばれるデータを読み書きすることによって分散並列処理を実現している。タプルとは、id によってデータをアクセスでき

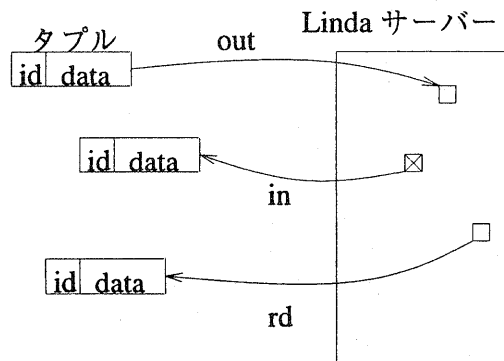


図 3: Linda の概要図

る単位であり、これをサーバに出し入れすることにより通信を行なう。Linda の概要図を図 3 に示す。ここでの in, out, rd は Linda のコマンドで、以下のような動作を行なう。

- out タプルをサーバに書き込む。
- in id で指定されたタプルをサーバから取り込む。もし、タプルにデータが存在しなければ(一致する id が存在しなければ)、指定された id を持つタプルがサーバに書き込まれるまで、待ち合わせを行い、取り込まれたタプルは、サーバからは削除される。
- rd id で指定されたタプルをサーバから読み込む。もし、タプルにデータが存在しなければ、指定された id を持つタプルがサーバに書き込まれるまで、待ち合わせを行い、読み込まれたタプルは、そのままサーバの中に残る。

id は Linda では任意の文字列で指定できるが、PlayStation 用のこのシステムでは 0 ~ 65535 までの整数を用いる。データとしては 249 バイトまでのバイナリを送る事ができる。

このタプル通信を、PlayStation の垂直同期と同期しながら入出力が行なうように拡張する。

3.2 なぜ同期型なのか

PlayStation のプログラムの状態遷移が画面の垂直同期と同期しているため、画計算後の残りの CPU 時間を通信に割り当てることにより、ゲーム画面上では遅延をなくすることができる。

通常の Linda では、in(rd) コマンドが来てそのタ

ブルが存在しなかった場合待ち状態になり、ゲームが停止してしまう。そこでin(rd) をコマンドの登録と答えの受け取りの2つに分けることにより、ゲームの流れが止まらないようになる。

4 同期型 Linda API の概要

4.1 同期型 Linda API の使い方

同期型 Linda API の説明を以下に示す。

- void start_linda();
 - 通信を初期化して、同期型 Linda API を使用可能にする。
- void psx_sync();
 - これを、PlayStation の描画の切替え時に呼び出す。これによって、Linda コマンドの処理を同期時間内で可能な量だけ、まとめて行なわれる。
- void psx_sync_n();
 - psx_sync() と、同じだがこれは描画の同期に関係なく全ての同期型 Linda コマンドの処理を行う。
- int psx_out(int id, char *data, int size);
 - データをサーバに書き出すのに使用する。PlayStation のゲームのオブジェクトの状態遷移時にいつでも使って良い。ただし、すぐに実行されるのではなくコマンドは queue にためこまれ、psx_sync() でまとめて実行される。queue がいっぱいになると失敗する。帰値はこのコマンドのシーケンス番号となる。
- int seq = psx_in(int id);
- int seq = psx_rd(int id);
- int seq = psx_ck(int id);
- char *data = psx_reply(int seq);
 - これらはサーバのデータを受け取る時に使用する。まず、psx_in/psx_rd/psx_ck のどれかで、どの id のデータを受け取るかを指示する。そして、psx_reply(seq) で、データを(次の状態で)チェックする。データが来ていれば psx_reply はデータへのポインタを返す。そのポインタのアドレスには、データの長さ(1 バイト)とそれに引続きデータそのものが格納される。psx_ck の場合はデータが来て、それはタブルがないことを示す空文字列の場合があるため、空文字のデータを out することは避けることが望ましい。このデータを使用後

は free により領域を解放しなければならない。

4.2 同期型 Linda のデータ構造

同期型 Linda で使用したデータ構造(C 言語)を以下に記述する。

```
typedef struct command_queue{
    struct command_queue *next;
    char *command;
    unsigned char size;
} COMMAND;

typedef struct psx_reply {
    struct reply *next;
    char *answer;
    int seq;
    char mode;
} REPLY;
```

psx_in/psx_rd/psx_ck/psx_out の各関数で、command キューを作成し、psx_sync(_n) で、そのコマンドキューを同期型 Linda サーバーに転送する。また、psx_in/psx_rd/psx_ck の各関数ではreply キューを作成して、サーバーから応答が返されたものとそうでないものを区別している。そして、psx_reply でサーバーから返されたデータをデータポインタとして受け取る。

5 PlayStation と unix への実装

5.1 同期型タブル通信を用いた PlayStation のプログラムの構造

この章では、通常の PlayStation のプログラムの構造と同期型タブル通信を用いた PlayStation のプログラムの構造の違いを比較する。

```
start_linda() .. ここで、Linda 処理を初期化
その他の初期化ルーチン
while(1){
    パッド等、入力データ受け取り PadRead() 等
    foreach ゲームオブジェクト {
        ゲーム上のオブジェクトの状態計算
        psx_in(), psx_out(), psx_rd()
        .. コマンド登録
        if (psx_reply()){
            答えが戻って来た時の状態へ
        } else {
```

答えの待ち状態

```

}
描画処理登録
}
psx_sync()
.. 登録されたコマンドを実行
画面切替え操作
登録描画処理実行開始
}

```

まず、最初に start_linda() を最初に実行する。この処理は Linda API を使用するのに必要な初期化を行なう。もし、初期化ルーチンでは Linda API を使用しない場合には、この start_linda() を初期化ルーチンの後で実行しても構わない。

そして、パッド等、入力データ受け取り、それに応じてゲームオブジェクトを計算し必要に応じて psx_in(), psx_out(), psx_rd() などの Linda コマンドをキューに格納して、psx_reply() の結果に応じて状態を変更して描画登録を行う。

ためられた Linda コマンドは psx_sync(_n) でサーバーの方に転送されそして、実行される。あとの画面切替え操作、登録描画処理実行開始は通常のプログラムと同じとなる。

通常の PlayStation プログラムと同様に while(1) でのループは 1 秒間に 60 回まわるように計算量を抑える必要があり、Linda コマンドの登録やサーバーへの転送も含める。

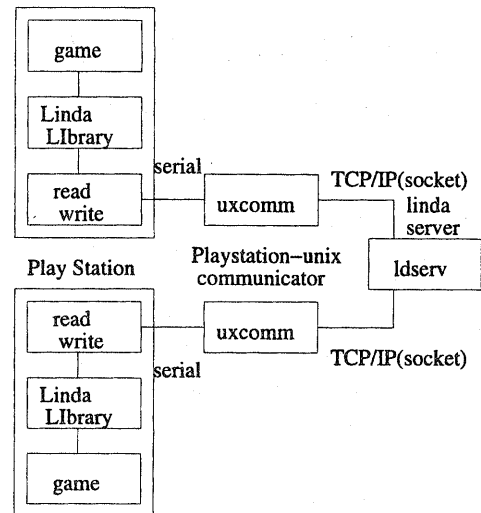


図 4: Linda のシステム構成図

5.2 同期型 Tuple 通信を実装したシステム構成

同期型 Tuple 通信を PlayStation に実装したシステム構成図を図 4 に示す。PlayStation のゲームからは Linda library を呼出を行ない、その Linda library が図 5 のパケット形式に変換し read/write のブロッキングを行なうルーチンを出し、そのルーチンが物理的に read/write を行なう。この read/write がシリアルをとおして、Unix 側にデータを転送する。Unix 側では、uxcomm(PlayStation-Unix Communicator) が、socket をとおして ldserv(Linda Server) へ転送し、そして Linda Server は、各 uxcomm から送られて来たデータ一括で処理し、それぞれの uxcomm に送り返す処理を行なう。そして、uxcomm はサーバから送られてきたデータをシリアルをとって PlayStation の中にいれて、Linda ライブラリがその意味のあるデータに変換してゲーム上で反映させる。

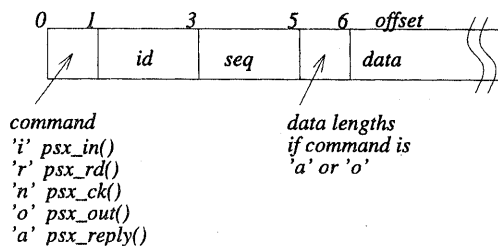


図 5: PlayStation から送られるパケット形式

5.3 同期型 Linda Server の実装

5.3.1 in command

```
'i' + id + seq + datasize + data
if(exists(tuple(id)) {
    answer(tuple(id));
    free(tuple(id));
} else {
    insert( tuple(id).waitq , seq );
}
```

5.3.2 read command

```
'r' + id + seq + datasize + data
if(exists(tuple(id)) {
    answer(tuple(id));
} else {
    insert( tuple(id).waitq , seq );
}
```

5.3.3 nonblock read command

```
'r' + id + seq + datasize + data
if(exists(tuple(id)) {
    answer(tuple(id));
} else {
    answer("");
}
```

5.3.4 out command

```
'o' + id + seq + datasize + data
if(exists(tuple(id).waitq) {
    answer(tuple(id));
    if(wait.mode = 'i')
        free(tuple(id));
} else {
    tuple(id).data に data を
    malloc/copy する。
}
```

6 実装・プログラミング例

この Linda ライブラリを利用する事により次のような状況やゲームが可能となる。

命令数	関数実行時間	ターンアラウンドタイム
0	13	*****
1	39	79
2	66	140
5	145	334
20	570	1339

表 1: 実行時間

1. 2 台以上の PlayStation で、パッドのデータを通じての対戦ゲーム。
2. とりあえず、データ全てをサーバに送り込み、PlayStation から必要なデータを取り出して更新し、必要に応じて相手の PlayStation の更新を取り出す形式を用いたパズルゲーム。

実際のゲーム例として

1. 2 台での協調 shooting ゲーム
2. 二人の人間による踊り
3. 2 台の戦車による対戦

しかし、パズルゲームを除いて、パッドのデータのみをやりとりしているために状態の誤差が大きい。

7 評価・まとめ

このライブラリを使い実際の通信時間を計測した。まず、PlayStation は 1 秒間に 60 回の垂直同期が発生する。この垂直同期が発生する前までに座標計算、描画処理がそれぞれ終了させる事が望ましい。この垂直同期が発生するまでには、TV の縦の解像度の分だけの水平同期が発生し、縦の解像度を 240 として、水平同期発生する回数を計測する。この時、1 水平同期は約 70us となる。

計測値は `psx_rd()` 命令の回数による、`psx_sync()` から帰ってくる時間と実際にデータが帰ってくるまでの時間 (水平同期数) を表 1 に示した。縦の解像度を 240 なので、描画時間も含めて 240 以内なら遅延は起こらない。

240 水平同期を超えると CPU 処理が描画に追いつかなくなり、ゲームの遅延が発生する。描画処理と通信処理を半々となるようなプログラムの場合、関数実行時間で 5 命令、命令を出してすべての結果が得られるまでの処理の場合 2 命令で遅延が発生してしまう。

現在は PlayStation <-> Unix 間の通信よりも、Unix <-> Unix 間の通信の方が遅い。これは小さいバケットで細かい同期を取っているためと思われる。これはバケットのブロッキングなどにより高速化できる余地がある。

この Linda を用いた PlayStation 開発環境は、今までのプレイステーションを超えたマルチユーザー・マルチマシンでプログラミング可能にする。

しかし、データ量が増えるに従って遅延時間が増加し PlayStation 間のデータの誤差 (矛盾) の原因となる。通信ルーチンの高速化とともに、これらの誤差を修正するアルゴリズムに関しても研究していく予定である。

参考文献

- [1] Margaret A. Dietz., "A Characterization of Communication Pattern in Distributed Applications.," May 1995.
- [2] Hideki Koike., A Framework for Visualizing Parallel Linda Programs.
- [3] Nicholas Carriero., Adaptive Parallelism and Piranha., Feb 1994.