汎用並列計算機上の並列論理シミュレータ*

–タイムワープに基づく方法とルーズな時刻同期による同期的方法–

石原 義勝[1]　　瀧 和男[2]†

[1]神戸大学大学院自然科学研究科 情報知能工学専攻‡

[2]神戸大学工学部情報知能工学科§

概要

論理シミュレーションを高速化する手法の一つにタイムワープ法を用いた並列論理シミュレーションがある．従来の報告でタイムワープ法の有効性は示されているが，実装環境が特殊であり汎用並列計算機上での評価は行なわれていなかった．そこで，本論文では3種類の汎用並列計算機上で実装，評価を行ない，いずれについても有効性を示すことができた．特にSPARCserver1000(6CPU)においては，最大で5.8倍の速度向上，295K[event/sec]の処理効率を得た．

更に，もう一つの試みとしてタイムワープ法に比べ処理オーバヘッドが小さくより高速な並列シミュレーション方式を提案する．この方式ではタイムホイル法の並列版を基礎とするが，ルーズに時刻同期をとることにより並列性を高めている．またイベント登録時の競合がなく，更に同期のためにロック操作を行なわないため，並列処理のオーバヘッドが小さい．現在この方法をコンパイルド・イベントドリブン方式と組み合わせて設計中である．

Abstract

"Time Warp" is one of the techniques to implement a high speed parallel logic simulator. Although an effectiveness of time warp has been reported, the execution machine was very limited and uncommon. Since efficiency of time warp logic simulator on general-purpose parallel computers has not been evaluated, we implemented it on three different parallel computer systems and took measurements. As a result, we found that the time warp simulator was efficient on each computer systems. Especially, it achieved 5.8-folds speedup and 295K[event/sec] on SPARCserver1000 (6CPU).

We propose another parallel simulation method aiming at lower overhead and faster speed than time warp. The method realizes higher parallelism taking the lose time synchronization, while it is based on the parallel timing wheel method. Since there is no contention of event registration among the processors and no data lock for time synchronization, the overhead of parallel processings is quite small in the method. A high speed parallel logic simulator based on the method is under development, coupled with the compiled event-driven method.

1　はじめに

近年の回路規模の増大により高速かつ柔軟な論理シミュレーションの要求が高まっている．その有効な手段の一つとして並列論理シミュレーションが挙げられる．

従来より論理シミュレータの並列化に関する報告がされている [3,6,8,9,12]．論理シミュレータの並列化を行なう際には，シミュレーション時刻の管理が重要な問題となる．時刻管理を行なう方法には集中時刻管理と分散時刻管理がある．集中時刻管理の代表的なアルゴリズムにタイムホイル法 [5,8] がある．タイムホイル法では単位時間ごとにすべてのプロセッサ間で同期をとる必要があるので，プロセッサ台数が増加するのに伴いそのオーバヘッドは無視できなくなる．一方，分散時刻管理では各プロセッサが独立して時刻を管理するため，並列処理に適している．

分散時刻管理の代表的なアルゴリズムにコンサーバティブ法 [2] とバーチャルタイムに基づくタイムワープ法 [1] がある．コンサーバティブ法に関しては従来から研究され問題点やその解決方法などが報告されている [2,9]．しかし，タイムワープ法は問題点であるロールバックのオーバヘッドが大きいと考えられ，並列論理シミュレータに適用された例は少ない．有効性が示されている報告もあるが，それは特殊な環境 (並列論理型言語専用マシン "Multi-PSI"

---

*Parallel Logic Simulators on General-Purpose Parallel Computers - An Asynchronous Method Based on Time Warp and A New Synchronous Method with Loose Synchronization -

†Yoshikatsu ISHIHARA[1] and Kazuo TAKI[2]

‡Division of Computer and Systems Engineering, Graduate School of Technology and Science, Kobe University

§Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University

上に並列論理型言語 KL1 で実装) であり汎用並列計算機上での有効性は示されていなかった [4]. そこで, 本研究では Unix OS を装備した汎用並列計算機上で手続き型言語である C 言語を用いて実装し, 評価を行なったので報告する.

更にタイムホイル法とタイムワープ法の比較結果より, タイムホイル法ではプロセッサ 1 台当たりの性能は高いが並列化した時の速度向上は低く, タイムワープ法ではプロセッサ 1 台当たりの性能は低いが速度向上は大きいことが分かった. そこで, プロセッサ当たりのシミュレーション性能も高く, かつタイムホイル法よりも速度向上の得られる新しい並列シミュレーション方式を提案する. この方式はタイムホイル法の並列版 [10] を基本に置き, 時刻の同期を "緩く" とることで, 高い速度向上を得ることを目的としている.

以下 2節ではタイムワープ法の概要について述べる. 3節ではタイムワープ法の実装方式について説明し, 4節で評価を行なう. 5節では今回提案する新しい並列シミュレーション方式について述べる.

2 タイムワープ法

タイムワープ法はイベントドリブン方式の一つで, ゲートが独自で時刻 (ローカルバーチャルタイム, 以下 LVT) を管理しながらシミュレーションを実行する. タイムワープ法ではシミュレーションを実行する際に, ゲートに到着するメッセージの到着時刻 (以下, タイムスタンプ) が正しい順序 (タイムスタンプが増加する順序) で到着すると仮定し, メッセージが到着するたびに評価を行なうことで, イベント処理の効率化を図っている.

しかし, シミュレーションの並列実行時にはメッセージが必ずしも正しい順序で到着するとは限らない. そこでもし上記の仮定に矛盾が起きた場合には, ゲートの状態を巻き戻して正しい順序で再評価を行なう. これをロールバックという. タイムワープ法ではロールバックのオーバヘッドが問題点となる.

また, ゲートはロールバックが発生した時のためにゲートに届いたメッセージを履歴として保存しておく必要がある. そのため, 回路規模が大きくなったり, シミュレーション時間が長くなるとメモリが不足するという問題点がある. この問題点を解消するためにタイムワープ法では以下のような方法を用いている. まず, すべてのメッセージのタイムスタンプの最小値をとるグローバルバーチャルタイム (以下, GVT) を設定する. そして, メモリの不足が生じた時に GVT よりも小さいタイムスタンプをもつ履歴を解放する.

3 シミュレータの実装

3.1 シミュレータの仕様

本シミュレータで扱う信号値は, High,Low,X (不定値) の 3 値とする. また, 遅延は各ゲートに単位時間の整数倍を割り当てるようなノンユニット遅延モデルとした.

3.2 タイムワープ法の実装

時刻管理機構 2節で述べたようにタイムワープ法では 2 種類 (LVT, GVT) の時刻を扱う必要がある. LVT は各ゲートに割り当てられるが, 実装時においてこの割り当ては効率が悪い. そこで, 本シミュレータではプロセッサごとに LVT を割り当て, ゲートは自分が割り当てられているプロセッサの LVT に従ってシミュレーションを実行するようにする.

また, GVT は 2節で述べたようにすべてのメッセージのタイムスタンプの最小値をとる. そこで, GVT 更新の際には一旦シミュレーションを停止することにする. つまり, 新しいメッセージが発行されないようにするわけである. そして通信経路上にメッセージが存在しないことを確認し, GVT の更新を行なう. また, GVT は履歴解放処理に必要となるので, 履歴解放処理を実行する直前に更新することにする.

プロセッサ間通信 プロセッサ間通信は, 共有メモリ型並列計算機では共有メモリ上のメッセージキューを介して行なう. 分散環境では本研究室で開発された並列オブジェクト指向言語「mosaic」[7,11] の通信ライブラリを用いて通信を行なう.

ゲートのプロセッサへの割り当て 本研究では, 各プロセッサにゲートを割り当てる際になるべくプロセッサ間通信が少なくなるように縦割指向戦略を用いた. 縦割指向戦略を図 1に示す.

3.3 使用データ

本研究では ISCAS'89 のベンチマークデータから 8 種類の順序回路を選んでシミュレーションを行なった. 使用したデータのゲート数, 信号線数, ゲート

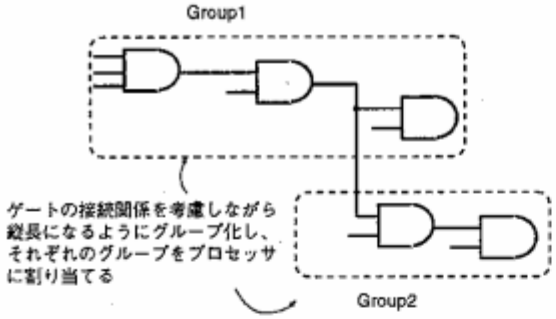

Fig.1:Cascading-Oriented Partitioning
図 1: 縦割指向戦略

当たりの平均入力線数，平均ファンアウト数を表1に示す．

Tab.1:Benchmark Data
表 1: ベンチマークデータ

回路	ゲート数	平均入力線数	平均ファンアウト数
s1494	683	2.15	2.08
s5378	385	1.70	1.61
s9234	6965	1.57	1.50
s13207	11965	1.66	1.55
s15850	13354	1.60	1.62
s35932	26433	1.92	1.92
s38417	31955	1.67	1.67
s38584	27695	1.83	1.85

また測定においては，各ゲートに1単位時間の遅延値を与えた．入力信号としてクロックは40単位時間を1周期とし，クロック線以外の入力端子はクロックの立ち上がりに合わせてランダムに信号値が変化するようにした．

3.4 実装環境

以下の3種類の汎用並列計算機上にシミュレータの実装を行なう．ただし，環境1では履歴解放処理とGVTは未実装である．

1. Symmetry S81 * (i80386 16MHz 28台, 120MB) ··· 環境1
2. SPARCstation2 互換機 (SPARC 40MHz, 28MB) 8台をVMEバスで結合 ··· 環境2 [11]
3. SPARCserver1000 (superSPARC+ 50MHz 6台, 192MB) ··· 環境3

*米 Sequent Computer Systems 社，共有メモリ型汎用並列処理計算機

4 実験結果

3種類の汎用並列計算機上で実装したときの速度向上の結果を図2，図3，図4に示す．また，それぞれの環境での単体性能，最大速度向上を表2に示す．図，表からも分かるように本実装において良好な結果を得ることができた．

Tab.2: Event/sec on 1CPU and Maximum Speedup
表 2: 単体性能と最大速度向上

環境	単体性能	最大速度向上 [倍]	最大性能
1	8	9 (16PE)	70
2	30	4.5(6PE)	127
3	77	5.8(6PE)	295

性能の単位は K[event/sec]

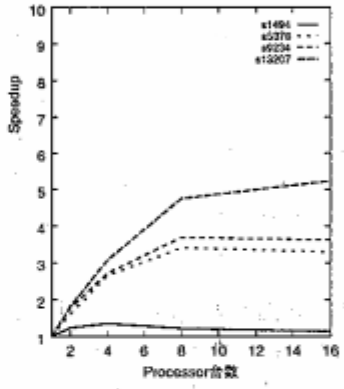

Fig.5: Speedup by Parallel Timing Wheel Method
図 5: タイムホイル法による速度向上

KL1 実装との比較 本実装では実装言語に手続き型言語C言語を使用した．C言語による実装がどのような影響を与えているかを確認するために文献 [4] の KL1 で実装されたシミュレータと比較を行なう．比較にはマシンの MIPS 値がほぼ同等 (3MIPS) の環境1の実験データを用いる．KL1 実装のシミュレータは分散メモリ型計算機上で実装されているので，プロセッサ間通信が発生しない単体性能で比較を行なう．また，環境1では履歴解放処理と GVT が未実装であるので，回路規模が最も小さい s1494 のデータを使用する．文献 [4] より KL1 実装の単体性能は 2.5K[event/sec] であった．よって，本実装の方が約3倍処理性能が良いことが分かった．これは以下のような理由によるもの

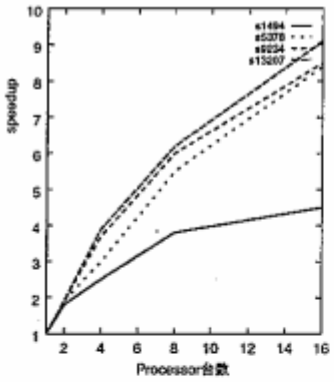

Fig.2: Speedup in S81(i80386)
図 2: 環境 1 での速度向上

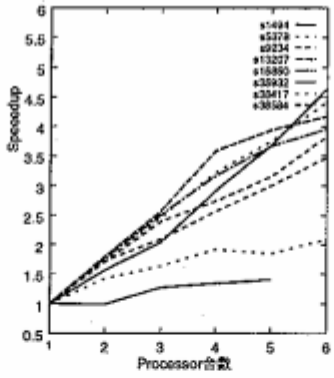

Fig.3: Speedup in SPARCstation2
図 3: 環境 2 での速度向上

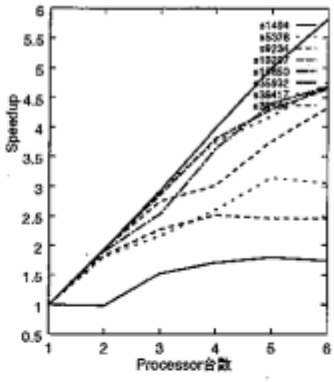

Fig.4: Speedup in SPARCserver1000
図 4: 環境 3 での速度向上

と思われる．KL1 は並列処理や同期に関する記述は容易であるが，単純な逐次計算におけるオーバヘッドが大きい．そのため，論理シミュレーションのように一つ一つの処理が単純なものにはそのオーバヘッドが影響したものと思われる．一方，C 言語は並列処理や同期に関する記述はプログラマに負担がかかるが，単純な処理には有効である．この違いが上記の性能差の原因であると思われる．

タイムホイル法との比較 文献 [4] によると共有メモリ型並列計算機上ではタイムワープ法よりもタイムホイル法の並列版 (各プロセッサにタイムホイルを設定し単位時刻ごとに同期をとりながらシミュレーションを実行，詳細は 5.1節参照) の方が有効であると予想されている．そこで，タイムホイル法を環境 1 で実装し比較を行なった．タイムホイル法の速度向上を図 5に示す．タイムホイル法の単体性能は 10K[evnet/sec] であり，タイムワープ法 (8K[event/sec]) よりも良好な結果が得られた．しかし，図 2，図 5からも明らかなように並列化した際にはタイムワープ法の方が速度向上が大きいことが分かる．回路を分割し各プロセッサに静的に割り当てる場合，タイムホイル法ではプロセッサ台数の増加に伴ってプロセッサ当たりの処理イベント数が減少する．そのためタイムホイル法の並列版では，処理イベントの少ないプロセッサは他のプロセッサの処理が終了するのを待つ必要がある．これにより速度向上が低下していると考えられる．

プロセッサ間通信 3.2節で述べた縦割指向戦略を用いることでプロセッサ間通信の頻度 (総メッセージ数に対するプロセッサ間通信となったメッセージの数) を 10%程度に抑えることができた．しかし，プロセッサ間通信の頻度が 30%を越えるデータ (s1494) については有効な速度向上を得ることができなかった．更に，全体としてはプロセッサ間通信の頻度が 10%程度であるがプロセッサごとで見てみると，他のプロセッサへ送出したメッセージの数が 0 のプロセッサもあれば 5000 以上送出しているようなプロセッサが存在するデータもあった．このような場合では他のプロセッサに頻繁にメッセージを送出しているプロセッサがボトルネックとなり速度向上は得ることができなかった．

負荷バランス タイムワープ法はイベントドリブン方式であるため，実行時にプロセッサの負荷は動的に変化する．本実装では 3.2節で述べた縦割指向戦略を用いて回路の分割を行なった．しかし，分割の際には “プロセッサ間通信を少なくすること” しか考慮に入れていなかった．そのため，イベントが多く発生するプロセッサと発生しないプロセッサの負荷のバランスが大きく違ってくるものが現れてきた．それにより速度向上が抑えられるものもあった．環境 3 において 6 プロセッサで 4 倍以上の速度向上が得られているものは，プロセッサ内で発生したイベント数 (プロセッサ外のゲートへのイベントを含む) の最大のものが最小のものの 1.1〜1.3 倍程度であった．それ以外のデータについては約 2 倍程度であった．

5 ルーズな時刻同期をとる並列論理シミュレーション

5.1 従来方式の問題点

1節で述べたように並列論理シミュレーションの時刻管理方式には集中時刻管理と分散時刻管理がある．分散時刻管理は大きな並列性を内在しているが，イベント処理1個当たりの時刻管理オーバヘッドは，集中時刻管理方式よりも大きい．そのため，プロセッサ当たりの性能は低くなることが問題となる．

一方，タイムホイル法は，集中時刻管理であるために逐次計算においては明らかにオーバヘッドが小さくプロセッサ当たりの処理性能は高い．しかし，並列化した際には次のような問題がある．

まずイベントを集中管理するために分散メモリ型並列計算機よりも共有メモリ型並列計算機の方により適した方式である．また，タイムホイルの並列化手法には2種類ある．

(i). タイムホイルを1つだけ設定する方法

(ii). 各プロセッサにそれぞれ1つタイムホイルを設定する方法

(i)の方法では発生するすべてのイベントが一つのタイムホイルで管理され，イベントをダイナミックにプロセッサに割り当てながらシミュレーションを実行する．イベントの登録，取り出し時に競合が起きるので，排他制御を行なう必要がある．そのため，プロセッサ台数が増加するにつれて排他制御のオーバヘッドが大きくなるという問題点がある．

(ii)の方法ではゲートをあらかじめプロセッサに割り当てておき，プロセッサに割り当てられたゲートに対するイベントはそのプロセッサで処理する．また，発生したイベントの送出先が他のプロセッサのゲートであれば，対応するプロセッサのタイムホイルに登録を行なう．そして，すべてのプロセッサが単位時刻ごとに同期を行ないながらシミュレーションを実行する．この方法ではイベントの取り出し時の競合はなくなるが，登録時の競合は存在する．これを解消するためには，あらかじめ他のプロセッサへのイベント登録を少なくなるようにゲートを割り付ける必要があるが，簡単ではない．更に4節で述べたようにプロセッサ台数が増加すると同期に要する時間がイベント処理に要する時間よりも大きくなるという問題点がある．

5.2 提案する方式の概要と考え方

5.1節の問題点を考慮しながら，並列タイムホイル法 (5.1節の (ii)の手法) を改良し，

1. イベント登録時の競合をなくすこと
2. 単位時刻ごとに同期をとらずにルーズな同期をとることで並列処理性を向上させること
3. それとともに同期オーバヘッドを軽減すること

を実現する方式を提案する．具体的な内容は以下に示す通りである．

まずタイムホイルの1つのタイムスロットにプロセッサ数分だけの独立したイベントリストを設定する．そして，あるプロセッサから送られたイベントは到着プロセッサのタイムホイル中の送出元プロセッサに対応するイベントリストに登録する．これによりイベント登録時の競合がなくなる．したがって，他のプロセッサへのイベント登録を少なくするようなゲートの割り当ては，共有メモリ型計算機を対象とする限り不要となる．上記の方法はすでに報告例がある [10] が，本提案ではこの方式と "緩い時刻同期" を組み合わせる点に特徴を持つ．それにより，従来のバリヤ同期に関連したオーバヘッドや，処理イベント数のばらつきによる速度向上の低下を防ぐことができる．

5.3 緩い時刻同期

基本的な考え方

> 各プロセッサの時刻をバリヤ同期を行なわずに進める．すなわち，プロセッサ A から B へのイベント登録について，ある時刻 t までに到着するイベントの登録がすべて終了したことを保証できる時，A は B に "A が登録したイベントに関する処理は時刻 t までは進めても問題はない" ことを通知する

時刻 t の求め方については A の現在時刻 T，A から B への最小遅延 τ とすると $t = T + \tau$ で求めることができる．このとき B が時刻 t の処理を開始できるならば，B は時刻 t のタイムスロットの中の通知済みのプロセッサに対応するイベントリストに対して処理を行なう．

この方式の利点は以下の通りである．従来の方式では単位時刻ごとにすべてのプロセッサが同期をとる必要があった．それに対し本方式では，あるプロセッサが時刻 t に進みたい時，すべてのプロセッサが処理を終了するのを待つ必要はない．すなわち，一つでも上述の "通知" をしてきたプロセッサが存在すれば，そのプロセッサが登録したイベントについて処理を開始することができる．これにより並列処理性が向上する．

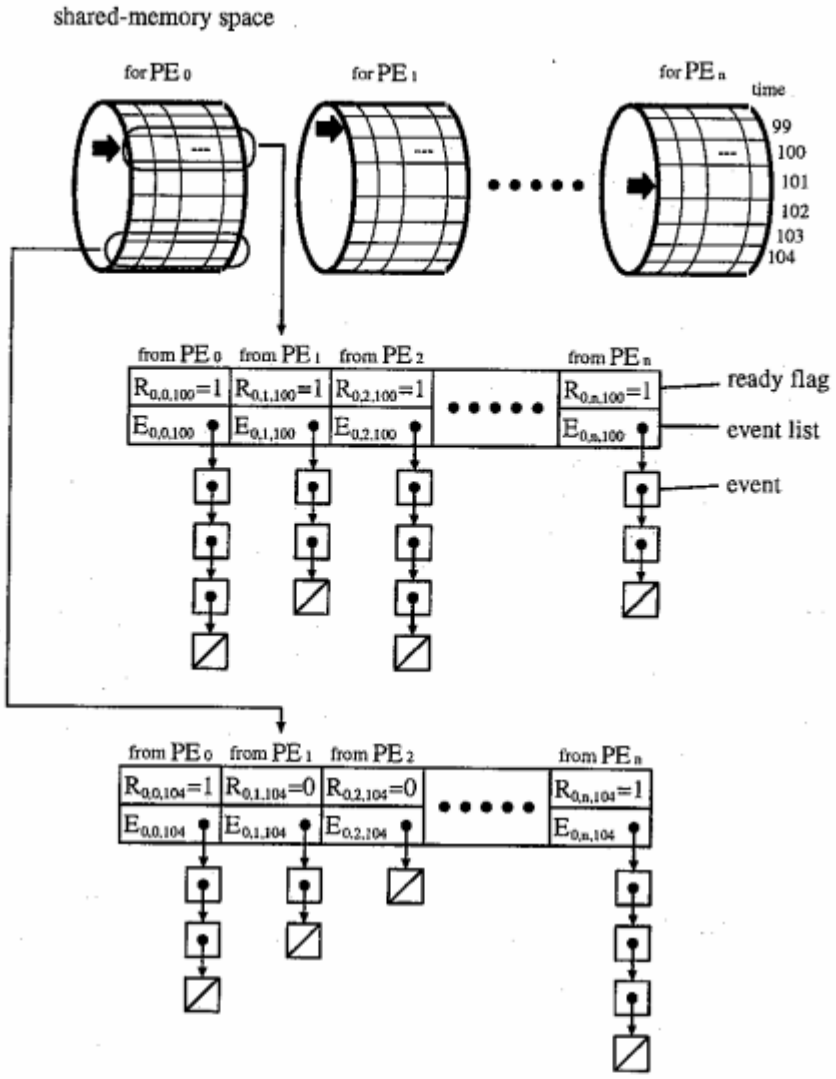

Fig.6: Data Structure Example for Loose Synchronization
図 6: 緩い時刻同期を実現するデータ構造の例

更に従来のバリヤ同期では，同時刻における各プロセッサのイベントのばらつきが直接性能低下に影響を及ぼしていた．それに対して本方式では

τ区間のタイムスロットに含まれる総イベント数

がプロセッサごとにどうばらつくかで性能低下が決まる．すなわち，τが長くとれるほどばらつき減少には効果が期待できるものである．

5.4 アルゴリズムとデータ構造

本方式で用いるデータ構造の例を図 6に示す．プロセッサ数だけのタイムホイルを設定する．各$PE(ProcessingElement)$ は現在時刻 (図 6の矢印の部分) をある約束の範囲内で独立に進めることができる．一つのタイムスロットにはPE数分だけ独立なフィールドを持たせ，PE_mが登録しに来た場合はタイムスロットの PE_mの部分に書き込みを行なう．このデータ構造により，イベント登録時のプロセッサ間の競合を防ぐことができる．

タイムスロットの 1 フィールドは，レディフラグ $R_{i,j,t}$とイベントリスト $E_{i,j,t}$からなる．ここで，i はタイムホイルの PE番号，j は書き込み PE 番号，t はタイムスロット番号 (登録を行なう時刻) である．i,j,t を指定することにより，任意のタイムホイル，任意のフィールド，任意のスロットを指定することができる．

ここで，$R_{i,j,t}=1$ は "PE_jは時刻 t までに登録す

```
/* n : NUMBER OF PE */
int   time;       /* CURRENT TIME OF PEi */
FIFO  queue;
int   delay[n];  /* MINIMUM DELAY FROM PEj
                    (j = 0 to n)          */
int   ready_flag[n][LENGTH OF TIME WHEEL];
EVENT *event_list[n][LENGTH OF TIME WHEEL];

scheduler()
{
  while (time != simulation_time){
    ENQUEUE ALL PE_ID;
    while (QUEUE IS NOT EMPTY) {
      DEQUEUE PE_ID;
      if (ready_flag[PE_ID][time] == 1){
        simulation(PE_ID);
        ready_flag[PE_ID][time] = 0;
      }
      else
        ENQUEUE PE_ID;
    }
    set_flag();
    time++;
  }
}

simulation(PE_ID)
{
  while(event_list[PE_ID][time] IS NOT EMPTY){
    ESTIMATION EVENT;
    if (NEW EVENT OCCUER AT TIME(t) TO PEj)
      ENTRY EVENT TO event_list[i][t] ON PEj;
  }
}

set_flag()
{
  for (k = 0; k < n; k++)
    ready_flag[i][time + delay[k]] = 1 ON PEk
}
```

Fig.7: Pseudo-code for the Basic Algorithm

図 7: 基本アルゴリズムの疑似コード

```
/* n : NUMBER OF PE */
int   clock;     /* CURRENT TIME OF PEi */
int   delay[n]; /* MINIMUM DELAY FROM PEj
                    (j = 0 to n)          */
FIFO  queue;
EVENT *event_list[n][LENGTH OF TIME WHEEL];

Scheduler()
{
  while (clock != simulation_time){
    ENQUEUE ALL PE_ID;
      while (QUEUE IS NOT EMPTY){
        DEQUEUE PE_ID;
        if (PE_ID'S CURRENT TIME
           + delay[PE_ID] > clock)
          simulation(PE_ID);
        else
          ENQUEUE PE_ID;
      }
      clock++;
  }
}

simulation(PE_ID)
{
  while(event_list[PE_ID][clock] IS NOT EMPTY){
    EVALUATE EVENT;
    if (NEW EVENT OCCUR AT TIME(t) TO PEj)
      ENTRY EVENT TO event_list[i][t] ON PEJ;
    }
}
```

Fig.8: Optimized Pseudo-code for Shared Memory Machines

図 8: 共有メモリ向きに最適化したアルゴリズムの疑似コード

べきイベントのうち，少なくとも PE_iのタイムホイルに登録すべきもののイベントはすべて終了した"ことを意味する．したがって，$R_{i,j,t}=1$ にセットされた後は，イベントリスト $E_{i,j,t}$は決して更新されないことが保証される．

PE_iは時刻 t に更新されると，時刻 t のタイムスロットのフィールドの中ですでに $R_{i,j,t}=1$ になっているものを探しだし，対応するイベントリストに存在するイベントを順に処理する．$j=m$ のイベントリストが空になると $R_{i,m,t}=0$ に更新し，同様の処理を繰り返す．ここで，タイムホイルのスロット数を"最大遅延 + 最小遅延の最大値" 以上に大きくとっておくことに注意する．この条件を守ればレディフラグやイベントリストの操作時に PE_mと PE_iが競合することはない．もし $E_{i,j,t}$が未処理でかつ $R_{i,j,t}=0$ であるフィールドが残っている場合は $R_{i,j,t}=1$ になるのを待つ．すべてのフィールドの処理が終了すると，PE_iは $R_{k,i,u}=1(k=0\ to\ n, u=t+\tau)$ の更新を実行し次の時刻に進む．この処理が 5.2節で述べた "通知" の処理に対応する．ここで，τは PE_iのゲートから PE_kのゲートへの最小遅延である．τの値は，i と kの組み合わせごとに異なる値を設定することができる．上記のアルゴリズムの疑似コード化したものを図 7に示す．図 7ではある一つの PE 上の処理を示している．またタイムホイルのスロット数が有限であって同領域が繰り返し利用されることについての記述は省略している．

5.5 共有メモリ型マシン，分散メモリ型マシンへの修正

本方式を共有メモリ型並列計算機で実装する場合には上記のデータ構造やアルゴリズムを簡単にすることができる．5.4節ではイベントの送出元がτの値を保持していた．しかし，τを登録先に持たせておいて"PE_iがPE_jの時刻をみて，自分の時刻との差がτ以下であることを確認"すれば5.4節の$R_{i,j,t}=1$と同じ効果が得られるので，レディフラグを省略することができる．共有メモリ型並列計算機用に最適化した疑似コードを図8に示す．

節5.4で述べたアルゴリズムは基本的には共有メモリ型計算機を対象としたものであるが，わずかの変更で分散メモリ型計算機上にも実装することができる．分散メモリでは他のPEからレディフラグを設定することは不可能である．そこで，PE_jがPE_iに登録するイベントを一旦PE_jの中で生成しておき，レディフラグをセットしていたタイミングでイベントをメッセージとしてPE_iに送信する．このメッセージがレディフラグをセットする意味を兼ねる．

5.6 実装設計

本方式を実現するために，コンパイルド・イベントドリブン方式と組み合わせて実装設計を進めている．コンパイルド・イベントドリブン方式はゲートの情報や接続関係などを実行コード中に埋め込んで処理を高速化する方法である．また，更に高速化のために図6のイベントリストの代わりにスタックを用いる．また，従来の方式は発生したイベントをすべてゲートの入力線に反映させてから，ゲートの評価を行なうという2パス方式であるが，これを1パスで行なう方式も考案し高速化を図っている．

6 おわりに

タイムワープ法を適用した並列論理シミュレータの例は少なく汎用並列計算機上での性能は明らかにされていない．そこで，本研究では3種類の計算機上で実装，評価を行なった．その結果それぞれの計算機上で有効な結果を得ることができた．また，タイムホイル法と比較することによって，集中時刻管理と分散時刻管理の双方の利点と問題点を明らかにすることができた．

次に，上記の評価結果を考慮してプロセッサ当たりの処理性能も高く，並列化した際にも速度向上が得易い新しい並列論理シミュレーションの方式を提案した．この方式では，従来のバリヤ同期と違ってルーズに同期をとることにより並列処理性の向上と同期オーバヘッドを軽減することを目的としている．この方式に関しては実装と評価が今後の課題である．

7 謝辞

5.5節の内容について御助言を頂いた神戸大学大学院自然科学研究科の沼昌宏先生をはじめ，並列LSI-CADタスクグループの諸氏に感謝します．

参考文献

[1] D.R.Jefferson. Virtual time. *ACM Trans. Prog.Lang. and Sys.*, Vol. 7, No. 3, pp. 404–425, July 1985.

[2] J.Misra. Distributed discrete-event simulation. *ACM Computing Servey*, Vol. 18, No. 1, pp. 39–65, March 1986.

[3] 工藤知宏, 木村哲郎, 寺沢卓也, 天野英晴. 共有メモリを想定した並列論理シミュレータ. 信学技法, Vol. CPSY91-23, pp. 151–158, 1991.

[4] 松本幸則, 瀧和男. バーチャルタイムによる並列論理シミュレーション. 情報処理学会論文誌, Vol. 33, No. 3, pp. 387–395, March 1992.

[5] Richard M.Fujimoto. PARALLEL DISCRETE EVENT SIMULATION. *Communications of the ACM*, Vol. 33, No. 10, pp. 30–53, October 1990.

[6] M.J.Chung and Y.Chung. Data parallel simulation using time-warp on the connection machine. *Des.Auto.Conf.*, pp. 98–103, 1989.

[7] 小倉毅, 瀧和男. 並列オブジェクト指向言語とマルチワークステーション上 の実装. JSPP'94 論文集, pp. 97–104, May 1994.

[8] P.Banerjee. *Parallel Algorithms for VLSI Computer-Aided Design*. Prentice-Hall,Inc., 1994.

[9] 下郡慎太郎, 鹿毛哲郎. メッセージドリブンによる並列論理シミュレーション. 信学技法, Vol. CAS88-110, pp. 23–30, 1988.

[10] Larry Soule' and Tom Blank. Parallel Logic Simulation on General Purpose Machines. *Proc.ACM/IEEE Des.Auto.Conf.(DAC)*, pp. 166–171, 1988.

[11] 瀧和男, 小倉毅, 小西健三. ワークステーション複合体による並列処理システム— 中・小粒度オブジェクト指向並列処理の実現 —. 情報処理学会 PRG13-7 研究報告, Vol. 93, No. 73, August 1993.

[12] Y.Matsumoto and K.Taki. Adaptive time-ceiling for efficient parallel discrete event simulation. *Proc. Object-Oriented Simulation Conf.(OOS'93)*, pp. 101–106, January 1993.