分割二分決定グラフによる有限状態機械の到達可能性解析のPCクラスタを用いた並列実装手法の提案

小島慶久[†] 藤田昌宏[††]

順序回路の形式的検証において，より大規模・複雑な回路を対象とするために，分割二分決定グラフ（POBDD: Partitioned-reduced Ordered Binary Decision Diagram）による有限状態機械（FSM: Finite State Machine）の到達可能性解析（Reachability Analysis）アルゴリズムが提案されている．この分割二分決定グラフの到達可能性解析では，いつ，どのように論理空間を分割するか決定するためのパラメータによって計算時間が大幅に変化することが経験的に知られており，適切なパラメータを選ぶことが重要となる．一方，分割二分決定グラフを用いた到達可能性解析アルゴリズムは，分割されたパーティション内の到達可能性解析とパーティション間での到達状態のやりとりに帰着でき，これらの計算は独立性が比較的高いため，並列化による高速化が期待できる．つまり，問題に適したパラメータを見つけることと，パーティション単位で計算の並列化をうまく組み合わせることが高速化に効果的であると期待できる．そこで本論文では，パーティション単位で到達可能性解析を並列化する際に，パーティション内の状態の変化の有無に着目して不要な計算や通信を防ぐ並列化手法を提案する．さらに，本研究では，パラメータ探索やパーティション単位での並列化の手法を実験するための環境を構築し，その上にこの並列化手法を実装した．

Parallelized Implementation of Reachability Analysis Using Partitioned-ROBDDs on PC-cluster

Yoshihisa Kojima[†] and Masahiro Fujita[††]

The reachability analysis algorithm of finite state machines (FSMs) using Partitioned-reduced Ordered Binary Decision Diagrams (POBDDs) has been proposed to target more large and complex circuits in sequential circuit verification. It is empirically known that the computation time of the algorithm drastically changes depending on the POBDD parameters which decide how and when to decompose the logical space, thus choosing good parameters is very important in the algorithm. On the other hand, it is expected that parallelization works well because the computations used in the algorithm (namely, reachability analysis in each partition and exchanging reached states between partitions) are highly independent. Consequently, it should be effective to combine both of i) to find and apply the good parameters for given target circuits and ii) to parallelize computations for each partition. In this paper, we propose a parallelized method which aims to avoid redundant computations or communications focusing on the changes of the states in the partitions. We built a PC cluster environment for experiments of parameter-searching and parallelization by partition, and we implemented this algorithm on this environment.

1. はじめに

モデルチェッキング[1)]をはじめとする順序回路の形式的検証手法では，検証対象の回路をFSMとして定式化し，初期状態からの到達可能な状態を探索することで，望ましくない状態に到達することがあるかを調べ，またその場合にそこに至るまでの入力シーケンスを計算することが基本となっている．この到達可能性解析をシンボリックに行うアルゴリズムでは，各状態を明示的に扱うことなく，状態を集合として扱い，状態遷移関係とともに論理関数で表現し，状態探索を論理関数どうしの演算によって行うことで，多くの状態を含む回路を扱うことが可能となる．これらの論理関数の表現には，二分決定グラフ（ROBDD: Reduced Ordered Binary Decision Diagram; 以下BDD）などが用いられるが，大規模・複雑な回路を対象としたときに，このBDDの爆発がしばしば問題となる．

† 東京大学大学院工学系研究科電子工学専攻
Electronic Engineering, The University of Tokyo
†† 東京大学大規模集積システム設計教育研究センター
VLSI Design and Education Center, The University of Tokyo

この問題を改善するために，分割二分決定グラフを用いた到達可能性解析アルゴリズムが Narayan らによって提案された[2]．このアルゴリズムは，分割前の論理空間の到達可能性解析を，分割後のパーティション内に閉じた到達可能性解析と，パーティション間の到達状態のやりとり（通信）に帰着することで，1 つの大きな問題を多数の小さい問題に置きかえている．ただし，この POBDD では，分割の際のパラメータによって，計算時間が大幅に変化し，良いパラメータをうまく選ばないと，計算の効率化が期待できない．また，POBDD を用いた到達可能性解析では，パーティション単位で計算の独立性が比較的高いため，パーティション単位で計算を並列化することによる高速化が期待できる．

そこで本論文では，POBDD を用いた到達可能性解析を，パーティション単位の並列計算として PC クラスタに実装する手法を提案する．パーティションの状態の変化に対して通し番号（バージョン）を振り，最後に参照したときからさらに変化があるかどうかを見ることによって，必要な計算を判断し，また計算全体の完了を検出できるようにしている．また，本研究では，その並列化手法の実装を行い，PC クラスタを用いて複数の異なる分割パラメータを並列に試すことで，対象の回路に適したパラメータを探索して計算全体の高速化を図るための実験環境を構築した．

以後，2 章では関連研究，3 章では予備知識としてシンボリック到達可能性解析アルゴリズム，論理関数の表現手法である BDD および POBDD，さらに POBDD を用いた到達可能性解析アルゴリズムについて説明し，POBDD を用いる際のパラメータの重要性について述べる．4 章では，本研究の提案手法として，パーティションを単位とした並列化手法について述べる．5 章で実装および実験結果を示し，最後に 6 章でまとめる．

2. 関 連 研 究

この章では，本研究の先行研究および類似研究に関して述べる．

多くの実用的な論理関数をコンパクトに表現するための手法として，BDD[3] があるが，論理関数が大規模・複雑になってくると，BDD のサイズが爆発してしまうという問題がある．Narayan らは，BDD の爆発を抑えるために，論理関数を 1 つのモノリシックな BDD で表現するかわりに，窓関数を用いて論理空間を複数のパーティションに分割し（**図 1**），論理関数をパーティションごとの BDD の集まりとして表現する POBDD[4] を提案した．

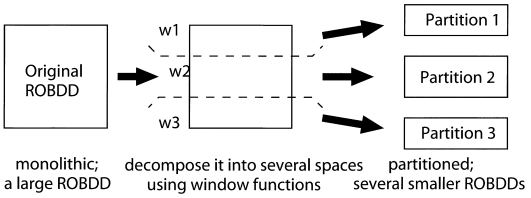


図 1 論理空間の分割の例（3 個）
Fig. 1 Decomposing a boolean space into 3 subspaces.

この POBDD を用いたシンボリックモデルチェッキングアルゴリズム[2] では，状態遷移関係や到達状態をパーティションとして分割し，メモリ使用量の削減に成功している．ただし，各パーティションの処理は 1 つずつ順に行っていた．また，計算のパフォーマンスはパーティショニングパラメータによって変化するが，最適なパラメータを探る指針については示されていない．

Heyman らは，32 台のワークステーション上にパーティションを分散させ，各ワークステーション上でパーティションごとの到達可能性解析を行い，またワークステーション間でパーティションの状態を交換するような並列実装を行っている[5]．パーティショニングのコスト関数のパラメータを動的に変化させること，また窓関数として 1 変数の単純なものではなく，やや複雑なものを探索することでパーティションごとのサイズのバランシングを行い，スケーラビリティを得ることに成功している．

さらに，Grumberg らは，coordinator と worker から構成されるアルゴリズムを提案し[6]，パーティションをプロセッサに対して固定的に割り当てないことで，使用する計算機資源を節約している．

3. 予 備 知 識

この章では，本研究に関連する既存の研究として，シンボリック到達可能性解析アルゴリズムと POBDD について述べる．

3.1 シンボリック到達可能性解析アルゴリズム

シンボリック到達可能性解析のアルゴリズム[2] を**図 2** に示す．到達可能性解析では，初期状態 $I(s_p)$ と許される状態遷移が与えられたときに，到達可能な状態 $R(s_p)$ を求める．特に，シンボリックな状態探索では，状態を集合として扱い，FSM への入力 i において，現状態 s_p から次状態 s_n への許される遷移を 1，それ以外を 0 として表現する状態遷移関係（TR: Transition Relation）$TR(s_p, s_n, i)$ を用いる．到達状態 $R(s_p)$ と状態遷移関係 $TR(s_p, s_n, i)$ の積をとるこ

```
FSM_TRAVERSAL(I(s_p), TR(s_p, s_n, i)){
    R(s_p) = F(s_p) = I(s_p)
    while(F(s_p) ≠ 0){
        N(s_n) = IMAGE(F(s_p), TR(s_p, s_n, i))
        F(s_p) = N(s_n ← s_p) ∧ ¬R(s_p)
        R(s_p) = R(s_p) ∨ F(s_p)
    }
}
```

図 2 標準的な到達可能性解析アルゴリズム
Fig. 2 Standard reachability algorithm.

s_p は現状態, s_n は次状態, i は入力, $I(s_p)$ は初期状態の集合, $TR(s_p, s_n, i)$ は現状態 s_p から次状態 s_n への遷移が入力 i において許されるときに 1 となる状態遷移関係, $R(s_p)$ は到達状態集合, $N(s_n)$ は次状態集合, $F(s_p)$ はあるステップにおいて初めて到達できたというフロンティア状態の集合である. $IMAGE$ は状態遷移関係を用いて現状態から次状態を求めるためのイメージ計算と呼ばれるものである.

とによって（これをイメージ計算と呼ぶ）次の到達状態 $N(s_n)$ を求めていく.

$$IMAGE(R(s_p), TR(s_p, s_n, i)):$$
$$N(s_n) = \exists s_p, i(TR(s_p, s_n, i) \wedge R(s_p))$$

ただし，$\wedge$ は論理積を意味する．以後，$\vee$ は論理和，$\neg$ は否定を意味する．つまり，状態 1 つ 1 つを明示的に扱うことなく，集合どうしの演算によって到達可能状態を求める．あるイメージ計算のステップにおいて，初めて到達した状態をフロンティア状態 $F(s_p)$ と呼び，このフロンティア状態がもう発生しなくなった段階 $F(s_p) = 0$ が定点（fixed-point）であり，初期状態 $I(s_p)$ からの到達可能状態 $R(s_p)$ がすべて求まったことを意味する．

到達可能状態の集合 $R(s_p)$ や状態遷移関係 $TR(s_p, s_n, i)$ を論理関数として表現する際，BDD を用いることが多いが，大規模な到達可能状態集合や状態遷移関係などを扱う際，BDD の爆発が問題となる．

3.2 POBDD

そこで，図 1 のように 1 つの大きな論理空間を場合分けにより複数の小さな空間に分割し，複数の小さな BDD を用いて 1 つの大きな論理関数を表現する POBDD という手法が Narayan らにより提案された[4]．なお，この POBDD に対して，従来の BDD をモノリシックな BDD と記述して区別する．

以下に関数 f の POBDD$\{(w_i, f_i)\}$ の定義を示す.

- w_i と f_i は変数順 π_i の BDD（ただし $1 \leq i \leq k$）
- $w_1 \vee w_2 \vee \cdots \vee w_k = 1$
- $f_i = w_i \wedge f$（ただし $1 \leq i \leq k$）

ここで，x_i は入力変数である．w_i は窓関数であり，論理空間の分割を定義する．(w_i, f_i) の組合せは i 番目のパーティションを表す．すべてのパーティションを足し合わせると，元の f になる．

$$f = f_1 \vee f_2 \vee \cdots \vee f_k$$

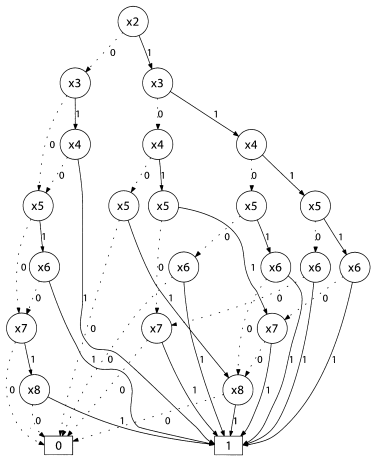
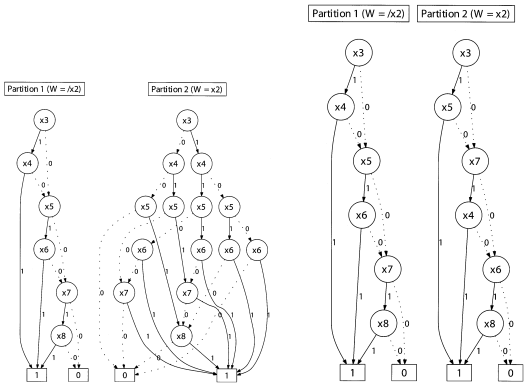


図 3 元のモノリシックな BDD
Fig. 3 Original (monolithic) BDD.



i) before opt.

$\pi_1 = \pi_2 =$
$(x_1, x_3, x_4, x_5, x_6, x_7, x_8)$

ii) after opt.

$\pi_1 =$
$(x_1, x_3, x_4, x_5, x_6, x_7, x_8)$
$\pi_2 =$
$(x_1, x_3, x_5, x_7, x_4, x_6, x_8)$

図 4 パーティションごとの変数順の最適化前と最適化後
Fig. 4 Before and after optimization of variable orders for each partition.

なお，$w_i \wedge w_j = 0$（ただし $i \neq j$）の条件下で，各パーティションは直交となる．

各パーティションを表現する BDD f_i はパーティションごとに独立に最適な変数順序を選ぶことができるため，よりコンパクトにできることがあるという特長を持つ．たとえば，**図 3** の BDD を変数 x_2 を用いて場合分けをして，2 つの BDD に分割する（**図 4** の i)）．この段階では，2 つのパーティションで変数順は同じなので，合計ノード数は元のモノリシックな BDD とほとんどかわらない．しかし，パーティションに分割したことで，右半分の変数順を独立に最適化すると，図 4 の ii) のように，劇的に小さくすること

ができる．このため，同じサイズのメモリを使用するとき，モノリシックな BDD を使う場合に比較して，表現可能な論理関数の範囲が広がることになる．

3.2.1 POBDD を利用するうえでのパラメータ

この項では，POBDD を利用する際のパラメータの一部を紹介する．

3.2.1.1 窓関数の選び方

POBDD を扱ううえでは，窓関数をどう選ぶか（論理空間をどのように場合分けして分割するか）が重要である．同じモノリシックな BDD（図 3）を 2 つのパーティションに分割するとしても，場合分けの変数として x_2 を選ぶとうまくいくが（窓関数 $w_1 = \neg x_2$，$w_2 = x_2$ を用いる．図 4 の ii)），x_1 を選ぶ（窓関数 $w_1 = \neg x_1$，$w_2 = x_1$ を用いる）と，単純に元の BDD とまったく同じものが 2 つできてしまうだけとなり，分割後のサイズにはかなりの差が生じる．前者は元のモノリシックな BDD に対して，各パーティションの BDD の最大サイズ，各パーティションの BDD の合計サイズともに小さくなっており，場合分けがうまくいくが，後者ではどちらも悪化する．したがって，単純に POBDD を利用して分割すればつねにうまくいくわけではなく，適切な窓関数を選択する必要がある．

ただし，入力変数が n 個であったとすると，窓関数は 2^{n^n} 個存在するため，窓関数を選ぶ操作自体にヒューリスティックが必要となる．選択基準を決定するコスト関数には，パーティショニング係数（partitioning factor）α と冗長係数（redundancy factor）β[2] というパラメータがある．また，窓関数をどの程度複雑なものまで探索するかを左右する，窓関数の変数の数などのパラメータがある．

3.2.1.2 分割のタイミング

POBDD では，BDD の分割を計算の過程でいつ行うかということも問題になる．BDD が大きくなる前に，計算の初期段階で分割する方法をスタティック・パーティショニングという．一方，計算の過程で BDD がある大きさよりも大きくなったら，オンデマンドで分割して計算を続行する方法をダイナミック・パーティショニングという．これらの分割には，いつ初期段階のパーティショニングを行うか，いくつのパーティションに分割するか，BDD が大きくなりすぎたと判断する BDD のノード数のしきい値などのパラメータが存在する．

3.3 POBDD を用いた到達可能性解析の実装

到達可能性解析では，通常，到達した状態の集合や状態遷移関係を BDD で表現するが，大規模な回路に対しては，BDD が爆発して計算が続行不能になるということがしばしばある．そこで，状態の集合や遷移関係の表現に，従来のモノリシックな BDD のかわりに POBDD を用いるという方法が提案された[2]．モノリシックな BDD では状態探索の途中で BDD が爆発し，計算が完了しなかった一方，POBDD では BDD の爆発を抑えこみ，探索空間の拡大に成功している．

```
1:POBDD_REACH(I(s_p), TR(s_p, s_n), k){
2:    {W_j(s_p)} = COMPUTE_WINDOWS(TR(s_p, s_n))
3:    foreach(j ∈ k){
4:        TR_jj(s_p, s_n) = W_j(s_p) ∧ W_j(s_n) ∧ TR(s_p, s_n)
5:        TR_jj'(s_p, s_n) = W_j(s_p) ∧ (¬W_j(s_n)) ∧ TR(s_p, s_n)
6:    }
7:    Queue = INITIALIZE_QUEUE({w_j(s_p), I(s_p), TR(s_p, s_n)})
8:    do{
9:        foreach(j ∈ k){
10:           reached_new = false
11:           foreach(l ∈ k, l ≠ j)
12:               reached_new  ∨=  communicate(R_j, R_l)
13:           if(reached_new)
14:               Queue+ = j
15:       }
16:       reached_new = false
17:       foreach(j ∈ Queue){
18:           reached_new  ∨=  computeLFP(R_j, TR_jj')
19:       }
20:       Queue = φ
21:   }while(reached_new = true)
22:}
23:computeLFP(R_j, TR_jj'){
24:   N_j(s_p) = FSM_TRAVERSAL(R_j(s_p), TR_jj)
25:   F_j(s_p) = (¬R_j(s_p)) ∧ N_j(s_p)
26:   R_j(s_p) = R_j(s_p) ∨ F_j(s_p)
27:   return  (F_j ≠ φ)
28:}
29:
30:communicate(R_j, R_l){
31:   R_old_j = R_j
32:   C_l' = updateCommState(R_l)
33:   R_j = R_j ∨ (C_l' ∧ W_j)
34:   return  (R_old_j ≠ R_j)
35:}
36:
37:updateCommState(R_l){
38:   return  IMAGE(R_l, TR_ll')
39:}
```

図 5 POBDD_REACH アルゴリズム[2]
Fig. 5 POBDD_REACH algorithm[2].

POBDD を用いた到達可能性解析のアルゴリズムを**図 5** に示す．ただし，この擬似コードは，並列化手法の説明のため，計算内容が変わらないように書きかえてある．$I(s_p)$ は初期状態の集合，$TR(s_p, s_n)$ は状態遷移関係，k はパーティション数である．最初に，パーティション，状態遷移関係を作成し（2–7 行目），ループに入る．ループ前半（9–15 行目）では，すべてのパーティションについて，パーティションどうしの到達状態のやりとり（*communicate* 計算と呼ぶ）を行っている（12 行目）．*communicate* 計算（30 行目）では，パーティション l の到達状態から l の外 l' への

通信状態 $C_{l'} = IMAGE(R_l, TR_{ll'})$ を計算し（38 行目．これを，*updateCommState* 計算と呼ぶ），パーティション j に取り込んでいる（12 行目）．ループ後半（17–19 行目）では，通信（あるいは初期化段階）によって変化が発生したパーティション j について，パーティション j 内での到達可能性解析を行っている（18 行目）．これを，LFP（Least Fixed Point）計算☆と呼ぶ．これらの操作により，いずれかのパーティションで新しい状態に到達した（$F_j \neq \phi$）場合には，ループを繰り返し（$reached\_new = true$），新たに通信状態を取り込んで再び到達可能性解析を行っている．

4. 提 案 手 法

この章では，本研究で提案するパーティション単位の並列化手法を提案する．

POBDD を用いた到達可能性解析アルゴリズムでは，分割前の論理空間での到達可能性解析は，分割後のパーティション内に閉じた到達可能性解析（LFP 計算）と，パーティション間での到達状態のやりとり（*communicate* 計算）に帰着できる．これらの計算は互いに独立性が高いため，並列化による高速化が期待できる．

この並列実装をするための手法として，バージョンというものを用いる．到達状態や通信状態の変化の有無をバージョンという通し番号で管理し，最後に行った LFP 計算，*communicate* 計算，*updateCommState* 計算のときから到達状態や通信状態が変化したかを比較することによって，再計算の必要の有無，計算全体が定点に達したかどうかを判断する．

以後，用語の定義，POBDD を用いた到達可能性解析の並列化の順に説明する．

4.1 用語の定義

この節では，以後使用する用語の定義を行う．

4.1.1 到達可能性解析に関するもの

- 初期状態 $I(s_p)$：到達可能性解析を開始する際に起点となる状態の集合．
- 到達状態 $R(s_p)$：初期状態から開始して，到達できた状態の集合．
- 状態遷移関係 $TR(s_p, s_n, i)$：入力 i において，現状態 s_p から次状態 s_n への遷移を許すときに 1，そうでないときに 0 として，状態遷移を定める．
- イメージ計算：状態遷移関係 $TR(s_p, s_n, i)$ を用いて，現状態 $R(s_p)$ から次状態 $N(s_n)$ を求めること．$IMAGE(R(s_p), TR(s_p, s_n, i)) : N(s_n) = \exists s_p, i(TR(s_p, s_n, i) \wedge R(s_p))$．状態遷移関係と現状態の積をとる．
- 到達可能性解析：$R(s_p) = FSM\_TRAVERSAL(I(s_p), TR(s_p, s_n, i))$．$I(s_p)$ から $TR(s_p, s_n, i)$ を用いて到達可能な状態 $R(s_p)$ をすべて求めること．
- フロンティア状態 $F(s_p)$：$FSM\_TRAVERSAL$ における，新たに到達した状態．
- 定点：$R(s_p) = N(s_n \leftarrow s_p)$ すなわち $F(s_p) = 0$ が成立している段階．これ以上イメージ計算を繰り返しても $R(s_p)$ は変化しない．

4.1.2 POBDD による到達可能性解析に関するもの

- 状態遷移関係 TR_{jj}：パーティション j 内から j 内への遷移．TR_{jj} と次の $TR_{jj'}$ はグローバルな状態遷移関係 TR から，パーティションごとの窓関数をかけることで導出される．$TR_{jj}(s_p, s_n, i) = W_j(s_p) \wedge W_j(s_n) \wedge TR(s_p, s_n, i)$．
- 状態遷移関係 $TR_{jj'}$：パーティション j 内から j 外への遷移．$TR_{jj'}(s_p, s_n, i) = W_j(s_p) \wedge (\neg W_j(s_n)) \wedge TR(s_p, s_n, i)$．
- 到達状態 R_j：パーティション j におけるすでに到達した状態．
- 通信状態 $C_{j'}$：パーティション j から外に行く状態．
- *computeLFP*：$LFP(j)\{R(j) = FSM\_TRAVERSAL(R_j, TR_{jj})\}$．パーティション j 内に閉じた到達可能性解析．LFP 計算のこと．
- *updateCommState*：$UPC(j)\{C_{j'} = IMAGE(R_j, TR_{jj'})\}$．パーティション j の到達状態 R_j から通信状態 $C_{j'}$ を求めること．
- *communicate*：$COMM(j, l)\{R_j = R_j \vee (C_{l'} \wedge W_j)\}$．パーティション j がパーティション l の通信状態 $C_{l'}$ のうち，パーティション j に入る状態を取り込むこと．*communicate* 計算，あるいは単に通信するという．

4.1.3 タスク，バージョンに関するもの

- タスク：LFP 計算，*updateCommState* 計算や *communicate* 計算など，パーティションに関する計算の単位．
- 状態の更新：初期化操作，LFP 計算，*communicate* 計算によって，到達状態 R_j が増大すること，もしくは *updateCommState* 計算によって通信状態 $C_{j'}$ が変化すること．

☆ 本来，Least Fixed Point という表現自体は定点という意味であるが，本論文ではパーティション内での LFP までの到達可能性解析という意味で使う．

- バージョン V_{R_j}，V_{C_j}：それぞれ，パーティション j の到達状態 R_j の更新の通し番号，通信状態 $C_{j'}$ の更新の通し番号を示す．計算の最初に，状態があれば 1，なければ 0 で初期化し，更新のたびに 1 ずつ増やす．
- 記録 L_{L_j}，L_{U_j}，$L_{C_{j,l}}$：それぞれ，パーティション j における最後の LFP 計算での到達状態 R_j のバージョン V_{R_j} の値，最後の $updateCommState$ 計算での到達状態 R_j のバージョン V_{R_j} の値，最後の $communicate$ 計算での通信状態 $C_{l'}$ のバージョン V_{C_l} の値を保持する．計算の初期段階に 0 で初期化する．

4.1.4 計算環境に関するもの

- PC クラスタ：多数の PC をネットワークで接続することにより構築した並列計算環境．
- プロセッサ：PC クラスタの構成要素．1 台の PC を指す．
- コントローラ（Controller）：多数のプロセシングエレメントを管理し，能動的にタスク処理を要求する．計算中のパーティションのロック，必要な計算の判断，計算の終了検出などを行う．コントローラ自身はもっぱら調停を行い，基本的にはタスクを行わない．1 つだけ存在する．
- プロセシングエレメント（PE）：コントローラからの要請により，受動的に一度に 1 つのタスクをこなす．PE プロセスは通常 1 つのプロセッサあたり 1 つ動かす．

4.2 POBDD を用いた到達可能性解析の並列化

POBDD を用いる際，各パーティションが直交するように窓関数を選んでおくことで，計算をパーティションごとに独立に行うことができるため，並列性が高いと期待できる．この性質を活かし，POBDD による到達可能性解析を，PC クラスタへ実装する．

到達可能性解析を PC クラスタ上で並列実装するためには，計算をどのように分散させ，進行を管理するか，また，どのタスクをどのプロセッサに割り付けるかなどの点を考慮する必要がある．具体的には以下の項目のようになる．

- パーティションに関連するものの切り分け
- パーティションの転送
- タスクの発生条件，計算の終了条件
- プロセシングエレメントへの割付け

これらについて，順に説明する．

4.2.1 パーティションに関連するものの切り分け

並列計算を効率的に行うためには，通信によるオーバヘッドを小さくするために，タスクと変数の依存関係を明確にする必要がある．

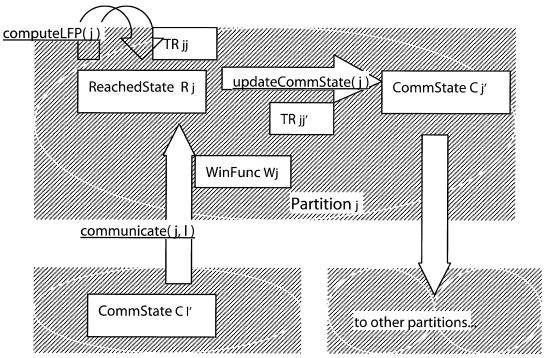


図 6 パーティションのタスクと依存関係
Fig. 6 Tasks of partitions and their dependencies.

$computeLFP$ は，TR_{jj} を用いてパーティション j 内に閉じた到達状態 R_j からの到達可能解析を行い，到達状態 R_j を更新する．$updateCommState$ は，$TR_{jj'}$ 到達状態 R_j から，パーティション j の外に行く通信状態 $C_{j'}$ を求める．$communicate$ は，外のパーティション l からの通信状態 $C_{l'}$ をパーティション j の到達状態 R_j に取り込む．

POBDD による到達可能性解析で用いる変数とタスクは，図 5 で示した擬似コードから以下のようになる．

パーティションに関連する主な変数は，

- W_j：パーティション j の窓関数．
- TR_{jj}，$TR_{jj'}$：パーティション j 内からパーティション j 内への状態遷移関係，パーティション j 内からパーティション j 外への状態遷移関係．
- R_j：パーティション j 内の到達状態 $R \wedge W_j$．
- $C_{j'}$：パーティション j 外への到達状態（ここでは通信状態と呼ぶ）$IMAGE(R_j, TR_{jj'})$．

である．

同様に，パーティションに関連するタスクは，

- $computeLFP(j)\{R_j = FSM\_TRAVERSAL(R_j, TR_{jj})\}$：（以下 LFP 計算）パーティション j 内に閉じた到達可能性解析．
- $updateCommState(j)\{C_{j'} = IMAGE(R_j, TR_{jj'})\}$：パーティション j 内からパーティション j 外への通信状態 $C_{j'}$ の導出．
- $communicate(j,l)\{R_j = R_j \vee (C_{l'} \vee W_j)\}$：パーティション l からの通信状態 $C_{l'}$ の取り込み．

である．

よって，**図 6** に示すように，計算を分散させるためには，パーティション j を単位として，窓関数 W_j，状態遷移関係 TR_{jj}，$TR_{jj'}$，到達状態 R_j，通信状態 $C_{j'}$ をひとまとめにして扱うと都合がよいことが期待される．

4.2.2 パーティションの転送

PC クラスタ上の各プロセッサはネットワークを介して接続されているため，パーティションはネットワークを経由してやりとりできる必要がある．このため，BDD の構造を変数順も含めてネットリストとして送信側でダンプ・受信側でリストアする．

4.2.3 タスクの発生条件

到達可能性解析では，通常，全体が定点に達するまで LFP 計算を繰り返すことになるが，POBDD を使った到達可能性解析では，どのパーティションの LFP 計算をしなければならないか，どのパーティションとどのパーティションが通信しなければならないかを決定する必要がある．

各タスクの発生タイミングは，

- $computeLFP$：到達状態 R_j が更新されたとき
- $updateCommState$：到達状態 R_j が更新されたとき
- $communicate$：通信状態 $C_{l'}$ が更新されたとき

となる．

そして，どのタスクも実行不要なことが，計算全体の終了条件となる．これらを管理するために，R_j や $C_{j'}$ に対してバージョン番号を振り，更新が起きたときにバージョンを増加させる操作を行う．各タスク完了時に，参照した R_j や $C_{j'}$ のバージョンをパーティションに保存しておき，最後に参照したバージョンよりも新しいバージョンの R_j や $C_{j'}$ を発見したら，対応するタスクを発生させる．**図 7** に，各タスクにおけるバージョン管理の方法を，**図 8** に，バージョンテーブルを参照して必要なタスクを判断するためのアルゴリズムをそれぞれ示す．**表 2** に，各タスクの発生条件，タスクが影響を与えうる対象の状態，タスク処理後の記録の関係をまとめる．

4.2.4 実 行 例

バージョン管理による到達可能性解析の実行例を以下に示す．**図 9** に，到達状態と通信状態のトレース結果を図示したものを示す．**表 1** に，3 つのパーティションで行った計算過程のバージョンの変化とタスクの進行の様子を示す．

以下，表 1 について説明する．$current$ の V_{R_i} はパーティション i の到達状態 R_i のバージョン，V_{C_i} はパーティション i からパーティション i 以外への通信状態 $C_{i'}$ のバージョンを示す．$lastReferred$ はパーティションが最後に行ったタスクで参照した状態のバージョンの記録を示す．パーティション i において，L_{L_i} は最後の LFP 計算において参照したパーティション i の到達状態 R_i のバージョンの値，$L_{C_{i,j}}$ はパーティション j との最後の $communicate$ 計算において参照したパーティション j の通信状態 $C_{j'}$ のバージョン V_{C_j} の値，L_{U_i} はパーティション i か

```
1: computeLFP(part){
2:     (reached_new, success) = FSM_TRAVERSAL(R_part, TR_part,part)
3:     if(reached_new)
4:         V_Rpart ++
5:     if(success)
6:         L_Lpart = V_Rpart
7: }
8:
9: communicate(part, peer){
10:    (reached_new, success) = COMMUNICATE(part, C_peer)
11:    if(reached_new)
12:        V_Rpart ++
13:    if(success)
14:        L_Cpart,peer = V_Cpeer
15: }
16:
17: updateCommState(part){
18:    oldCommState = part.commState
19:    newCommState = IMAGE(part, TR_part,part')
20:    if(oldCommState ≠ newCommState)
21:        V_Cpart ++
22:    L_Upart = V_Rpart
23: }
```

図 7 $computeLFP$，$communicate$，$updateCommState$ ルーチン

Fig. 7 $computeLFP$, $communicate$, $updateCommState$ routines.

$computeLFP$ は，パーティション内に閉じた到達可能性解析（1 行目），$communicate$ は，パーティション外からの通信状態の取り込み（9 行目），$updateCommState$ は，パーティション外への状態（通信状態）の計算（17 行目）をそれぞれ行っている．$computeLFP$ では，到達状態 R_{part} が新しい状態に到達した（変化した）ときに，到達状態のバージョン $V_{R_{part}}$ を増やす（4 行目）．逆に状態が変化しなければ，バージョンはそのままである．LFP 計算が完了したら，最後に参照した到達状態のバージョン $V_{R_{part}}$ を $L_{L_{part}}$ に記録する（6 行目）．$communicate$，$updateCommState$ のタスクについても，タスク発生の元となる状態のバージョンがそれぞれ $V_{C_{peer}}$，$V_{R_{part}}$ であること，計算によって更新されうる状態がそれぞれ R_{part}，C_{part} であること，計算が完了したときのバージョンの記録先が $L_{C_{part,peer}}$，$L_{U_{part}}$ であることが違うだけで，バージョン管理に関する振舞いは基本的に $computeLFP$ と同様である．

```
1: LookupVersionTable(){
2:     taskCandidate = φ
3:     foreach(part ∈ Partitions){
4:         if(V_Rpart > L_Lpart){
5:             taskCandidate ← new Task(LFP, part)
6:         }
7:         if(V_Rpart > L_Upart){
8:             taskCandidate ← new Task(UPDATECOMM, part)
9:         }
10:        foreach(peer ∈ Partitions){
11:            if(V_Cpeer > L_Cpart,peer){
12:                taskCandidate ←
13:                    new Task(COMMUNICATE, part, peer.commState)
14:            }
15:        }
16:    }
17: }
```

図 8 バージョンテーブルの参照

Fig. 8 Looking up the version table.

状態のバージョン（到達状態 $V_{R_{part}}$，通信状態 $V_{C_{peer}}$）が，最近の計算の際に参照したときの値（$L_{L_{part}}$，$L_{U_{part}}$，$L_{C_{part,peer}}$）から変化していたら，タスクが必要と判断する．タスクがまったく必要ないとき，全体の計算は完了している．

表 1 バージョン管理による計算の進行例
Table 1 An example of computation progress by version management.

		P_1						P_2						P_3						
		current		last referred				current		last referred				current		last referred				
step	task	V_{R_1}	V_{C_1}	L_{L_1}	$L_{C_{1,2}}$	$L_{C_{1,3}}$	L_{U_1}	V_{R_2}	V_{C_2}	$L_{C_{2,1}}$	L_{L_2}	$L_{C_{2,3}}$	L_{U_2}	V_{R_3}	V_{C_3}	$L_{C_{3,1}}$	$L_{C_{3,2}}$	L_{L_3}	L_{U_3}	result
0	Init	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Only R_1 has states.
1	LFP(1)	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	R_1 got new states.
2	UPC(1)	2	1	2	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	$C_{1'}$ changed.
3	Comm 2-1, 3-1	2	1	2	0	0	2	1	0	1	0	0	0	0	0	1	0	0	0	Only R_2 got new states.
4	LFP(2)	2	1	2	0	0	2	2	0	1	2	0	0	0	0	1	0	0	0	R_2 got new states.
5	UPC(2)	2	1	2	0	0	2	2	1	1	2	0	2	0	0	1	0	0	0	$C_{2'}$ changed.
6	Comm 1-2	2	1	2	1	0	2	2	1	1	2	0	2	0	0	1	0	0	0	No new states.
7	Comm 3-2	2	1	2	1	0	2	2	1	1	2	0	2	0	0	1	1	0	0	No new states.

表 2 タスクとバージョンの依存関係
Table 2 Dependencies between tasks and versions.

task	condition	may affect	post-process
$LFP(j)$	$V_{R_j} > L_{L_j}$	R_j	$L_{L_j} \leftarrow V_{R_j}$
$UPC(j)$	$V_{R_j} > L_{U_j}$	$C_{j'}$	$L_{R_j} \leftarrow V_{U_j}$
$COMM(j,l)$	$V_{C_l} > L_{C_{j,l}}$	R_j	$L_{C_{j,l}} \leftarrow V_{C_l}$

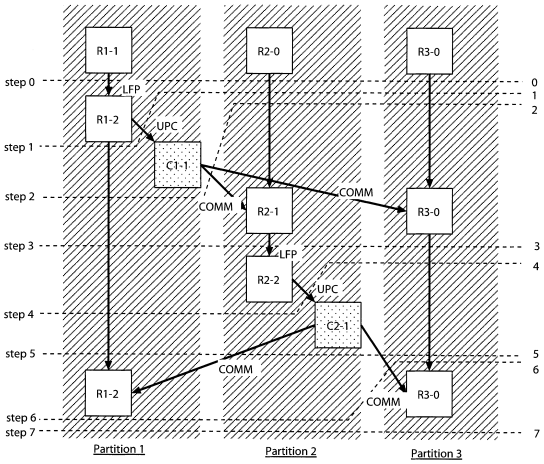


図 9 トレース結果の例
Fig. 9 An example for the trace result.

ら外への通信状態を求める *updateCommState* において参照したパーティション i の到達状態 R_i のバージョン V_{R_i} の値をそれぞれ示す．

- ステップ 0：テーブル *lastReferred* の要素はすべて 0 で初期化される．各状態のバージョンは初期状態が存在するパーティション 1 を除いて 0 になっている．パーティション 1 において，$V_{R_1} > L_{L_1}$ は最後に行った LFP 計算のときから，到達状態 R_1 が変化したことを示し，LFP 計算が必要であることが分かる．また，$V_{R_1} > L_{U_1}$ は最後に *updateCommState* を行ったときから到達状態が変化していることを示すので，*updateCommState* が必要である．
- ステップ 1：パーティション 1 で LFP 計算を行ったとする．その結果，到達状態 R_1 が変化したので，バージョン V_{R_1} を増やす．また，V_{R_1} を L_{L_1} に記録する．$V_{R_1} > L_{U_1}$ なので *updateCommState* が必要である．
- ステップ 2：パーティション 1 で *updateCommState* を行った．その結果，通信状態 $C_{1'}$ が変化したので，バージョン V_{C_1} を増やす．また，V_{R_1} を L_{U_1} に記録する．$V_{C_1} > L_{C_{2,1}}$ は，最後にパーティション 2 がパーティション 1 と通信したときからパーティション 1 の通信状態 $C_{1'}$ が変化したことを示すので，パーティション 2 はパーティション 1 と通信する必要がある．同様に，$V_{C_1} > L_{C_{3,1}}$ なのでパーティション 3 はパーティション 1 と通信する必要がある．なお，この実行例ではパーティション 1 の LFP 計算をパーティション 1 の *updateCommState* よりも先に行ったが，バージョン管理によるアルゴリズム自体はスケジューリングに対して自由度を残しているため，スケジューラはほかの指標を基準としてタスクの順序を決定してよい．
- ステップ 3：パーティション 2 が通信状態 $C_{1'}$ を取り込んだ．その結果，パーティション 2 の到達状態 R_2 が変化したので，V_{R_2} を増やす．また，V_{C_1} を $L_{C_{2,1}}$ に記録する．$V_{R_1} > L_{L_1}$ なので LFP 計算が必要，$V_{R_2} > L_{U_2}$ なので *updateCommState* が必要である．また同時に，パーティション 3 も通信状態 $C_{1'}$ を取り込んだ．到達状態 R_3 は変化しなかったので，V_{R_3} はそのままで，V_{C_1} を $L_{C_{3,1}}$ に記録する．
- ステップ 4：パーティション 2 で LFP 計算を行った．その結果，到達状態 R_2 が更新されたので，V_{R_2} を増やし，V_{R_2} を L_{L_2} に記録する．$V_{R_2} > L_{U_2}$ なので *updateCommState* が必要である．
- ステップ 5： パーティション 2 で *updateCommState* を行った．その結果，通信状態 $C_{2'}$ が更新されたので，V_{C_2} を増やし，V_{R_2} を L_{U_2} に記録する．$V_{C_2} > L_{C_{1,2}}$ なのでパーティション 1 は

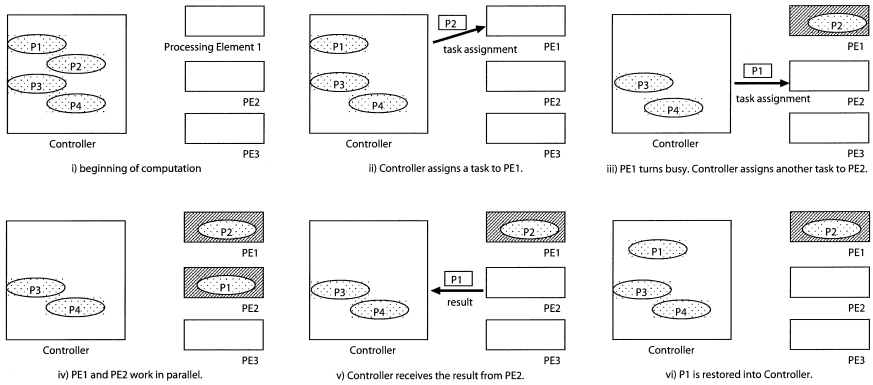


図 10 コントローラとプロセシングエレメント
Fig. 10 Controller and PEs

i) 計算開始前は，プロセシングエレメント PE1–3 はフリーで，コントローラがパーティション P1–P4 をすべて持っている．
ii) コントローラはフリーな PE1 に P2 を転送し，タスクを割り当てる．
iii) PE1 はビジーになる．コントローラはさらにフリーな PE2 に P1 を転送し，タスクを割り当てる．
iv) PE2 もビジーになる．PE1 と PE2 は並列に動作する．
v) PE2 でのタスクが完了し，PE2 はフリーになる．
vi) コントローラは計算結果の P1 を受け取る．
このように計算は進行する．計算の並列度は最大で PE の数とパーティションの数のいずれか小さい方の値になる．

パーティション 2 と通信する必要がある．同様に，$V_{C_2} > L_{C_{3,2}}$ なのでパーティション 3 はパーティション 2 と通信する必要がある．

- ステップ 6：パーティション 1 が通信状態 $C_{2'}$ を取り込んだ．到達状態 R_1 は変化しなかったので，単純に V_{C_2} を $L_{C_{1,2}}$ に記録する．
- ステップ 7：パーティション 3 も通信状態 $C_{2'}$ を取り込んだが，やはり到達状態 R_3 は変化せず，単純に V_{C_2} を $L_{C_{3,2}}$ に記録する．タスクを発生させる条件はなく，定点に収束したので，計算は完了である．なお，このトレースは一例にすぎず，異なるスケジュールによる進行もありうる．

4.2.5 PE への割付け

本研究での実装では，**図 10** に示すように，1 つのコントローラプロセスと複数の PE（Processing Element）プロセスから構成する．**図 11**，**図 12** のように，コントローラはパーティションのバージョンテーブルを参照してタスクを発生させ，PE に対してタスク実行を要求する．タスクの実行は原則 PE で行い，コントローラはスケジューリングに専念する．パーティションは，タスク実行時のみ，コントローラから PE に転送されたうえで処理されるので，PE に特定のパーティションを結び付けていない．このため，PE の数がパーティションより少なくても動作可能であり，また，スケジューラは PE に対するパーティションの割当てを動的に調整できる．

コントローラは，PE でタスクを実行しているパーティションにロックをかけるため，さらにそのパーティションに対してほかのタスクを発生させることはない．このため，計算の並列度は最大でパーティションの数となる．また，コントローラがスケジュールを集中管理するため，並列プロセスどうしの同期は自明なものとなる．

```
1: Controller_main(){
2:     PE = initializePEs()
3:     terminate = false
4:     while(¬terminate){
5:         if(isAnyTaskNotInvokedYet() ∧ isAnyVacantPE()){
6:             pe = takeOneVacantPE()
7:             task = takeOneTask()
8:             sendTask(pe, task)
9:         }
10:        if(isAnyTaskInProgress())
11:            receiveResult()
12:        if(¬isAnyTaskInProgress() ∧ ¬isAnyTaskNotInvoked())
13:            terminate = true
14:            sendTerminate(PE)
15:        }
16:    }
17: }
```

図 11 コントローラの動作
Fig. 11 The behavior of Controller.

コントローラはタスクの管理，パーティションのロック，プロセシングエレメントへのタスクの割当てを集中的に行っている．起動可能なタスクが残っていたら，空き PE がある限り割り当て，PE にタスク処理を要求する（5 行目）．実行中のタスクがあれば，計算結果が返るのを待つ（10 行目）．実行中のタスクもなく，さらに行うべきタスクもなければ，終了する（12 行目）．

5. 実装および実験結果

この章では，本研究で構築した PC クラスタシステムおよび提案した並列化手法の動作確認のため，ISCAS89 ベンチマーク[7] の例題（**表 3**）を用いた．その実験結果を示す．

```
1: PE_main(){
2:     terminate = false
3:     while(¬terminate){
4:         task = receiveTaskFromController()
5:         switch(task.type){
6:           case LFP :
7:             computeLFP(task.partition); break
8:           case UPDATECOMM :
9:             updateCommState(task.partition); break
10:          case COMMUNICATE :
11:            communicate(task.partition, task.commState); break
12:          case TERMINATE :
13:            terminate = true; break
14:        }
15:    }
16: }
```

図 12 プロセシングエレメント（PE）の動作
Fig. 12 The behavior of PE.

PE は，コントローラから送られてくるタスクの要求に応じて，実際にタスクを行い，その結果をコントローラに返す．タスクには，$computeLFP$（6 行目），$updateCommState$（8 行目），$communicate$（10 行目）があり，PE 1 つあたり，同時に 1 つまでのタスクを処理する．terminate 要求が送られてきたら，終了する（12 行目）．

表 3 例題の回路の規模
Table 3 The sizes of the example circuits.

	# of					
circuit	inputs	outputs	D-FFs	inverters	other gates	reachable states
s1238	14	14	18	80	428	2616
s1269	18	10	37	132	437	1.1313e9
s1512	29	21	57	367	413	1.6573e12
s3271	26	14	116	537	1035	1.3176e31
s3330	40	73	132	974	815	7.2778e17
prolog	36	73	136	623	978	7.2778e17
s4863	49	16	104	742	1600	2.1904e19
s5378	35	49	179	1775	1004	–
s6669	83	55	239	1009	2071	–

5.1 PC クラスタシステムおよびプログラム実装

Intel Pentium-4（1.8 GHz），512 MB の SDRAM の PC を 32 台，100Base-TX のスイッチを用いて接続してクラスタシステムを構成した．OS には Linux-2.4.21 を利用した．

順序回路の形式的検証，合成，シミュレーションツールの VIS（Verification Interacting with Synthesis）[8),9)] を基本として，米国富士通研究所 Amit Narayan 氏らが POBDD を利用したアルゴリズムを実装したツールがある．これを元にして，さらに PC クラスタ上で並列処理ができるように改良した pcppvis というツールを実装した．並列化の実装にあたっては，米国オークリッジ国立研究所による明示的メッセージパッシングの並列計算ライブラリである PVM（Parallel Virtual Machine）[10),11)] を用いた．

実験では，以下の手順によってスタティック・パーティショニングとダイナミック・パーティショニングを用いた．

- 初期状態からのイメージ計算を I 回行う．ここでは VIS 同様モノリシックな BDD を用いる．
- スタティックパーティショニング．2^n 個のパーティションに分割する．ここで，n は窓関数に含む変数の数である．
- LFP や $updateCommState$ の計算の途中，BDD の大きさがしきい値 t を超えたら，分割を行う（ダイナミック・パーティショニング）．もしダイナミック・パーティショニングをしても BDD の大きさが t 以下にならなければ，t の値自体を 2 倍にして続行する．

実験では，パラメータのバリエーションとして，しきい値 $t = \{30000, 50000, 80000\}$，初期イメージ計算回数 $I = \{0, 1, 2\}$，窓関数に含む変数の数 $n = \{1, 2\}$ の組合せの 18 通りを網羅的に試した．

また，分割に用いる窓関数は以下のように作成した．

まず，BDD の変数 1 つずつに対して，コスト $cost_x(f)$ を評価する[4)]．

$$cost_x(f) = \alpha[p_x(f)] + \beta[r_x(f)]$$
$$p_x(f) = \max(\frac{|f_x|}{|f|}, \frac{|f_{\neg x}|}{|f|})$$
$$r_x(f) = \frac{|f_x| + |f_{\neg x}|}{|f|}$$

ここでは，$\alpha = 0.3$，$\beta = 0.7$ を用いた．ただし，コストの評価自体に時間がかかるため，最初の 1 回を除き，コスト最小の 1 個の変数のみについて再計算して，それ以外のコストの値は再利用している．次に，コストが小さい n 個の変数について，0，1 のすべての値を組み合わせて 2^n 個の窓関数を作成する．たとえば，窓関数に用いる変数が x と y の 2 個ならば，窓関数は $w_1 = x \wedge y$，$w_2 = x \wedge \neg y$，$w_3 = \neg x \wedge y$，$w_4 = \neg x \wedge \neg y$ の 4 個となる．

到達可能性解析の基本アルゴリズムなどの条件は，従来の VIS と pcppvis とで同じものを使用している．BDD 変数順序の最適化アルゴリズムは sifting を用いた．イメージ計算にはクラスタ化状態遷移関係を用いる IWLS95 アルゴリズムを用いた．その際，クラスタサイズも同じ値を用いた．

5.2 ISCAS89 の例題による実験結果

ISCAS89 ベンチマーク[7)] に含まれる順序回路である，s1269，s3330 および prolog を用いた実験結果を，表 4〜表 8 にそれぞれ示す．

表において，$\#ofprocs$ は計算に用いたプロセッサの数，n は分割で窓関数に含む変数の数，I は最初に分割する前に行うイメージ計算の回数，t はイメージ計算を中止させる（BDD が爆発したと判断してダイナミック・パーティショニングを行う）BDD のノード数のしきい値の初期値，$\#ofparts$ は最終的なパー

表 4 pcppvis の実験結果（1）s1269
Table 4 Result of pcppvis (1) s1269.

exp#	*#of procs*	*I*	*n*	*t*	*#of parts*	cputime (sec)
original vis	1	–	–	–	–	7735
1	1	1	2	50000	5	**114**
2	1	1	2	80000	5	125
3	1	1	1	30000	9	177
4	1	1	2	30000	10	209
5	1	0	2	80000	4	231
6	1	0	1	50000	5	272
7	1	0	2	30000	9	280
8	1	0	1	80000	4	296
9	1	0	1	30000	9	311
10	1	1	1	50000	9	413
11	1	0	2	50000	4	466
12	1	1	1	80000	7	499
13	1	2	1	30000	25	1188
14	1	2	1	50000	20	1257
15	1	2	1	80000	27	4357
16	1	2	2	80000	27	7109
17	1	2	2	50000	34	7256
18	1	2	2	30000	>56	>8000

表 5 pcppvis の実験結果（2）s3330
Table 5 Result of pcppvis (2) s3330.

exp#	*#of procs*	*I*	*n*	*t*	*#of parts*	cputime (sec)
original vis	1	–	–	–	–	4415
1	1	1	2	50000	4	**1080**
2	1	1	2	80000	4	**1080**
3	1	1	1	80000	2	1147
4	1	2	2	80000	5	1220
5	1	1	2	30000	8	1739
6	1	2	2	50000	8	2114
7	1	1	1	50000	7	4319
8	1	0	2	50000	7	5326
9	1	0	2	80000	7	7514
10	1	0	1	30000	>29	>7500
11	1	0	2	30000	>15	>7500
12	1	1	1	30000	>24	>7500
13	1	0	1	80000	>10	>7500
14	1	2	1	30000	>24	>7500
15	1	2	1	50000	>14	>7500
16	1	2	1	80000	>6	>7500
17	1	2	2	30000	>26	>7500
18	1	0	1	50000	>11	>7500

ティションの数，*cputime* は計算時間（実時間，単位は秒）である．計算時間に “>” とあるものを除き，定点まで，到達可能状態がすべて求まった．

5.2.1 最適パラメータの探索

表 4，**表 5** を見ると，同じ 1 台のプロセッサで計算しても，分割パラメータによって計算時間が大きく異なることが分かる．また，パラメータの与え方が悪い（表 4 の (18) や，表 5 の (8)–(18)）と，モノリシックな BDD を用いた従来の VIS（*originalvis*）よりも計算時間が遅くなってしまうことがある．以上のことから，POBDD による到達可能性解析では，パラメータを適切に与えることが計算時間短縮のうえで重要であることが分かる．

表 6 pcppvis の実験結果（3）s1269
Table 6 Result of pcppvis (3) s1269.

					pcppvis		*straightforward*	
exp#	*#of procs*	*I*	*n*	*t*	# of *parts*	cputime (sec)	*#of parts*	cputime (sec)
1	4	1	2	80000	11	427	9	503
2	3	1	2	80000	9	363	9	524
3	2	1	2	80000	5	**110**	5	118
4	1	1	2	80000	5	124	5	136
5	2	1	2	50000	7	115	7	155
6	1	1	2	50000	5	114	5	130

表 7 pcppvis の実験結果（4）s3330
Table 7 Result of pcppvis (4) s3330.

					pcppvis		*straightforward*	
exp#	*#of procs*	*I*	*n*	*t*	*#of parts*	cputime (sec)	*#of parts*	cputime (sec)
1	4	1	2	50000	4	**438**	4	482
2	3	1	2	50000	4	674	4	771
3	2	1	2	50000	7	1516	5	1778
4	1	1	2	50000	4	1074	4	1241

表 8 pcppvis の実験結果（5）prolog
Table 8 Result of pcppvis (5) prolog.

					pcppvis		*straightforward*	
exp#	*#of procs*	*I*	*n*	*t*	*#of parts*	cputime (sec)	*#of parts*	cputime (sec)
original vis	–	–	–	–	–	91811	–	–
1	4	1	2	80000	8	**2310**	>13	>7200
2	3	1	2	80000	10	3137	>17	>7200
3	2	1	2	80000	8	2827	7	3738
4	1	1	2	80000	8	5908	>8	> 7200

この実験は，自明な並列性を利用して，同じ計算を異なるパラメータを与えて PC クラスタ上で同時に計算させて最も計算が早く終わったものの結果を利用することで，外見上プロセッサ 18 台で，s1269 では 68 倍，s3330 では 4.1 倍程度の高速化に成功したという解釈ができる．また，いったん最適なパラメータが求まったら，再び同じ計算をする際に再利用することもできる．

5.2.2 並列化による高速化

表 4 や表 5 で求まった最適なパラメータを元に，さらに台数を増やして並列化することによる計算時間の変化を，**表 6**，**表 7** にそれぞれ示す．なお，**表 8** については，プロセッサ 1 台ずつのパラメータの組合せでは短時間に求まるものが 1 つしか見つからなかったので，その 1 つについて，プロセッサの台数を増やしたときの結果を示している．

プロセッサ 1 台あたり PE プロセス 1 つずつを割り当て，コントローラプロセスは PE プロセスのうちの 1 つと同一のプロセッサで実行した．

比較対象として，本論文の提案手法であるバージョン管理を行わない，単純な方法で並列化した場合を考える．いずれかのパーティションにおいて，*LFP* 計算や *commnicate* 計算の結果フロンティア状態に達

したら，すべてのパーティションどうしで *communicate* 計算をやり直す必要があるものとする．これは，*LFP*，*commnicate* の計算のあとでフロンティア状態に達したとき，記録 $L_{C_{part,peer}}$ のすべてを 0 で初期化することで実現できる．この場合の実験結果を表中の *straightforward* として示した．提案手法の結果については，*pcppvis* として示してある．*straightforward* では，パーティション間の余計な通信によって，*pcppvis* よりも多くの計算時間がかかっている．

表 6 では，パラメータ $I = 1$，$n = 2$，$t = 80000$ について，プロセッサを 2 台にしたときに最善の結果が出ている．しかし，プロセッサを 3 台，4 台にすると，結果はかえって悪化してしまっている．これは，スケジューリングの変化のために，計算の過程が変わってしまい，パラメータの最適性が失われてしまったことに起因する．

また，パラメータ $I = 1$，$n = 2$，$t = 50000$ については，プロセッサを 1 台から 2 台にしてもあまり改善はみられない．これは，この問題の計算の並列性が低かったことに起因する．

pcppvis と *straightforward* を比較すると，7%から 44%の幅はあるものの，いずれも *pcppvis* の方が速い．モノリシック BDD を用いた *originalvis* と比較して，全体で 70 倍高速化している．適したパラメータで POBDD を用いることによる効果は 56.9 倍，プロセッサを 2 台にすることによる効果は 15%，バージョン管理による効果は 7%である．

表 7 では，パラメータ $I = 1$，$n = 2$，$t = 50000$ のときにプロセッサを 1 台から 4 台まで変化させている．2 台のとき，スケジューリングの変化により，計算時間が増大してしまっている．*pcppvis* は *straightforward* よりも 10%から 17%速い．*originalvis* と比較して，全体で 10 倍高速化している．適したパラメータで POBDD を用いることによる効果が 3.6 倍，プロセッサを 4 台にすることによる効果が 2.6 倍，バージョン管理による効果が 10%である．

表 8 では，パラメータ $I = 1$，$n = 2$，$t = 80000$ のときのみ比較的短時間に計算を行うことができた（他のパラメータでは，いずれも 2 時間を超えてしまった）．*originalvis* と比較して，全体で 40 倍高速化している．適したパラメータで POBDD を用いることによる効果が 16 倍，プロセッサを 4 台にすることによる効果は 2.6 倍である．この例では *straightforward* と *pcppvis* とでスケジューリングが変化してしまったために直接は比較できないが，強いていえば 30%から 2.2 倍強の効果があることになる．

このほかに，s1238，s1512，s3271，s4863，s5378 および s6669 についても試したが，s1238 は例題として小さすぎ，数秒で終了してしまうため，pcppvis の初期化オーバヘッドの方が大きく，目立った改善はなかった．s1512 については，適したパラメータを pcppvis で用いることにより，*originalvis* に比べて 4.2 倍高速に解くことができたが，並列性が不十分なため，プロセッサ数を増やしても改善はみられなかった．s3271，s4863 については，この実験では良い分割が見つからなかったため，モノリシック BDD を用いて処理する方が効果的であると考えられる．s5378，s6669 については，回路規模が大きすぎるため，VIS，pcppvis のいずれも数時間以内に解くことはできなかった．

6. 結　　論

本論文では，パーティション単位で到達可能性解析を並列化する手法としてパーティションの到達状態と通信状態の変化をバージョンとして管理することにより，必要な計算と計算の終了を判断するアルゴリズムを提案した．また，POBDD を用いた到達可能性解析の並列実装に対応できる実験環境を構築し，その上にこのアルゴリズムを実装し，実験によって，このアルゴリズムが動作することを確認した．

今後は，本研究で構築した実験環境と並列化手法を利用して，良い分割パラメータを並列に探索すること，到達可能性解析そのものを並列化することの 2 つを組み合わせた高速化手法を考える予定である．

謝辞　米国富士通研究所の Jawahar Jain 氏には，本研究の助言をいただいた．日本ヒューレット・パッカード社には，PC クラスタ作成にあたり，高性能な PC を 32 台も寄付していただいた．この場を借りて感謝の意を表す．

参 考 文 献

1) Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L. and Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond, *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*, Washington, D.C., pp.1–33, IEEE Computer Society Press (1990).
2) Narayan, A., Isles, A., Jain, J., Brayton, R. and Sangiovanni-Vincentelli, A.: Reachability Analysis Using Partitioned-ROBDDs, *Proc. ICCAD*, pp.388–393 (1997).
3) Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Computers*, Vol.C-35, No.8, pp.677–691 (1986).

4) Narayan, A., Jain, J., Fujita, M. and Sangiovanni-Vincentelli, A.: Partitioned ROBDDs — A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions, *Proc. 1996 IEEE/ACM Int'l Conf. on Computer Aided Design*, pp.547–554 (1996).
5) Heyman, T., Geist, D., Grumberg, O. and Schuster, A.: Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits,*Computer-Aided Verification, 12th International Conference*, Grumberg, O. (Ed.), pp.20–35, Springer (2000).
6) Grumberg, O., Heyman, T. and Schuster, A.: A Work-Efficient Distributed Algorithm for Reachability Analysis, *Proc. 12th International Conference on Computer Aided Verification CAV*, pp.54–66 (2003).
7) Collaborative Benchmarking Laboratory: ISCAS '89 Benchmarks.
http://www.cbl.ncsu.edu/CBL_Docs/iscas89.html
8) Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G. and Villa, T.: VIS: A System for Verification and Synthesis, *Proc. 8th International Conference on Computer Aided Verification CAV*, Alur, R. and Henzinger, T.A. (Eds.), New Brunswick, NJ, USA, pp.428–432, Springer Verlag (1996).
9) The VIS Group: The VIS Home Page.
http://vlsi.colorado.edu/~vis/
10) Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA. (1994).
11) Oak Ridge National Laboratory: PVM: Parallel Virtual Machine.
http://www.csm.ornl.gov/pvm/pvm_home.html

(平成 16 年 4 月 28 日受付)
(平成 17 年 1 月 7 日採録)

小島 慶久

2004 年東京大学大学院工学系研究科電子工学専攻修士課程修了．同年同博士課程進学．同年 10 月休学，米国 Zenasis Technologies, Inc. 勤務．VLSI 設計最適化ツールの研究開発に従事．

藤田 昌宏（正会員）

1985 年東京大学大学院工学系研究科情報工学専攻博士課程修了．工学博士．同年富士通入社．富士通研究所にて，VLSI CAD の研究に従事．1988〜1989 年イリノイ大学客員研究員．1993 年米国富士通研究所出向．VLSI CAD 研究グループ立ち上げ．2000 年より東京大学大学院工学系研究科電子工学専攻教授．2004 年同大学大規模集積システム設計教育研究センター教授．論理合成，論理検証，システムレベル設計支援技術等の研究に従事．SpecC コンソーシアム言語ワーキンググループ主査．IEEE，ACM 各会員．