BDD を用いた順序回路の形式的検証における逆像計算の並列化手法*

永見 康一　　木村 晋二　　渡邊 勝正

奈良先端科学技術大学院大学 情報科学研究科†

順序回路に対する BDD を用いた形式的設計検証において用いられる逆像計算を、共有主記憶型並列計算機上で行なう手法を提案する。これは、従来提案されている BDD の深さ優先 traversal に基づく逐次計算手法を、複数プロセッサを用いて行なうものである。BDD の再収れん箇所で並列性が失われる状況を防ぐために、各プロセッサが動的に traversal 経路を決定する手法を用いる。

We propose a new parallel algorithm of inverse image computation used in BDD-based formal verification. The algorithm is suitable for a shared memory multi-processor system; processors execute the depth-first traversal on a BDD as a graph. For balancing the load, processors dynamically decides their traversal path to avoid the situation that some processors do the same computation at a re-convergent subgraph of the BDD which they are traversing on.

1　はじめに

論理回路の形式的検証とは、設計した回路の論理的機能が仕様を満たしているかどうかを厳密に判定する作業であり、設計回路の大規模・複雑化に伴って重要な技術になってきている。BDD (Binary Decision Diagram; 二分決定グラフ) [1,10] の登場により、処理過程で必要になる論理関数処理や集合操作を計算機上で効率的に行なうことが可能となり、計算機支援による検証の自動化は実用的になりつつある。検証対象の設計規模の拡大や処理の高速化を狙って、効率化手法が盛んに研究されている [3,5,7,8]。

順序回路の検証手法として代表的な、時相論理を仕様記述に用いる記号モデル検査 [4] においては、順序回路に対して Kripke 構造とよばれるグラフを構成し、このグラフを探索してある性質を持った節点の集合の不動点を求めることで検証を行なう。この探索においては、遷移関数の逆像計算の反復を行なう [2]。

中江らが提案した逆像計算手法 [9] は、1. 次状態変数を導入する必要がない、2. 節点集合を表す BDD を 1 度たどるだけで計算が可能、などの点から効率的である。以後、この手法をボトムアップ逆像計算手法と呼ぶ。

並列計算環境の充実により、VLSI-CAD の分野においても並列計算の応用が期待されている。本稿では、このボトムアップ逆像計算を共有主記憶型並列計算機上で行なう手法を提案する。以下 2 節で本稿で用いる用語や概念を導入し、3節で提案手法の基礎となっている逐次アルゴリズムを説明し、4 節で並列化手法について述べる。

2　準備

2.1　論理関数と有限集合の特徴関数表現

n 変数論理関数とは、関数 $\{0,1\}^n \to \{0,1\}$ である。また、論理関数ベクトル $\{0,1\}^n \to \{0,1\}^m$ は、$\boldsymbol{f}$ のように表記する。論理関数 f, g について、$\overline{f}$, $f \cdot g$, $f + g$, $f \equiv g$は、それぞれ論理否定、論理積、論理和、論理一致の各演算を表す。f の入力変数 x に 0,1 を代入して得られる関数(コファクタ) をそれぞれ $f_{\overline{x}}$, f_x と書く。$\exists x.f$ は $f_{\overline{x}} + f_x$ で定義される smoothing 演算を表す。$\boldsymbol{x} = [x_1, x_2, \cdots, x_l]$ とするとき、$\exists \boldsymbol{x}.f = \exists x_1.(\exists x_2.(\cdots(\exists x_l.f)\cdots))$ である。

有限集合 S の部分集合 A に対し、$\chi_A(a) = 1 \Leftrightarrow$

*Parallel Inverse Image Computation for BDD-based Formal Verification of Sequential Circuits

†Kouichi NAGAMI, Shinji KIMURA and Katsumasa WATANABE, Graduate School of Information Science, Nara Institute of Science and Technology

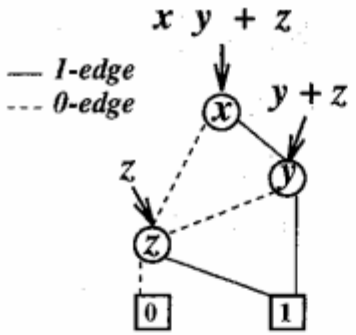

Fig. 1: 論理関数 $x \cdot y + z$ の BDD

$a \in A$ で定義される関数$\chi_A : S \to \{0,1\}$ を A の**特徴関数**と定義する。S の要素を二進符号化し、A に属する要素の符号をオンセットに対応させることにより、χ_A を論理関数として実現できる。このとき、$\chi_{\overline{A}} = \overline{\chi_A}, \chi_{A \cup B} = \chi_A + \chi_B, \chi_{A \cap B} = \chi_A \cdot \chi_B$ などが成り立ち、集合演算を論理関数処理によって行なうことができる。論理関数表現として BDD を用いれば、集合の要素を明示的に表現しないため集合の大きさとグラフサイズが直接関係せず、大規模な集合を効率的に処理することが可能な場合がある。以降では、混乱の恐れのない限り集合をその特徴関数の形で表記することがある。

2.2 BDD

BDD は、論理関数を表現する非巡回有向グラフである [1;10]。図 1 は論理関数 $x \cdot y + z$ を表す BDD である。BDD は真理値表の二分木表現である二分決定木を縮約したグラフと見ることができ、各節点は 0-edge および 1-edge の二つの枝を持つ。また、BDD を構成する各節点は，レベル付けされており、各レベルが入力変数に対応している。この対応は一対一対応であり、この対応を固定すると BDD は論理関数の標準形となる。各節点 (を根節点とする部分グラフ) は、それぞれがある論理関数を表現しており、0-edge および 1-edge が指す先はコファクタであるという関係がある。すなわち図 2 のように論理関数の Shannon 展開式をグラフ構造が再帰的に表現している。混乱の恐れのない限り節点 N が表す論理関数も N と表記する。また $i \in \{0,1\}$ とするとき、節点 N の i-edge が指している節点を N_i と表記する。

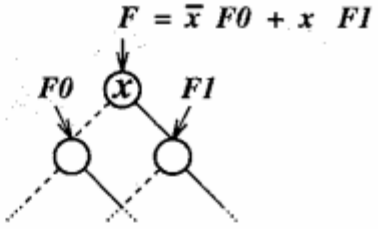

Fig. 2: Shannon 展開のグラフ構造表現

2.3 順序機械と Kripke 構造

順序機械 M を、入力 $\boldsymbol{x} = [x_1, x_2, \cdots, x_l]$, 出力 $\boldsymbol{z} = [z_1, z_2, \cdots, z_t]$, 状態変数 $\boldsymbol{y} = [y_1, y_2, \cdots, y_m]$, 遷移関数 $\boldsymbol{\delta} = [\delta_1(\boldsymbol{x}, \boldsymbol{y}), \delta_2(\boldsymbol{x}, \boldsymbol{y}), \cdots, \delta_m(\boldsymbol{x}, \boldsymbol{y})]$, 初期状態集合 $S_0(\boldsymbol{y})$ で表す。また、$I(\boldsymbol{x}) : \{0,1\}^l \to \{0,1\}$ を入力の定義域 (の特徴関数) とし、$S(\boldsymbol{y}) : \{0,1\}^m \to \{0,1\}$ を状態集合 (の特徴関数) とする。M の Kripke 構造 $K(M)$ とは、以下で定義される節点集合 V および枝集合 E を持った有向グラフである。

- $V \stackrel{def}{=} \{(\boldsymbol{x}, \boldsymbol{y}) |\ I(\boldsymbol{x}) \cdot S(\boldsymbol{y}) = 1\}$
- $E \stackrel{def}{=} \{< (\boldsymbol{x}, \boldsymbol{y}),\ (\boldsymbol{x}', \boldsymbol{y}') > |\ (\boldsymbol{x}, \boldsymbol{y}), (\boldsymbol{x}', \boldsymbol{y}') \in V$ かつ $\boldsymbol{y}' = \boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{y})\}$

すなわち、$K(M)$ の各節点は M の遷移に対応しており、$K(M)$ の各枝は 2 つの遷移間の隣接関係 (続けて生じることがあるかどうか) を表している。また、$K(M)$ に対し、$Init_{K(M)}(\boldsymbol{x}, \boldsymbol{y}) \stackrel{def}{=} I(\boldsymbol{x}) \cdot S_0(\boldsymbol{y})$ で初期節点集合を定義する。これは M の初期状態からのすべての遷移に対応する節点の集合を意味する。図 3 は、ある順序機械の状態遷移図 (上) と、その Kripke 構造 (下) を例示している。

2.4 逆像計算

Kripke 構造 $K(M)$ の節点集合 (の特徴関数) $G(\boldsymbol{x}, \boldsymbol{y})$ の、遷移関数 $\boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{y})$ による逆像計算は、次式で定義される。なお、簡単のため M の入力の定義域 $= \{0,1\}^l$ (すなわち $I(\boldsymbol{x}) = 1$) とする。

$$Prev\,(G\,(\boldsymbol{x}, \boldsymbol{y})) \stackrel{def}{=}$$

$$\exists \boldsymbol{x}', \boldsymbol{y}'. \left(\left(\prod_{i=1}^{m} y_i' \equiv \delta_i\,(\boldsymbol{x}, \boldsymbol{y}) \right) \cdot G\,(\boldsymbol{x}', \boldsymbol{y}') \right) \quad (1)$$

この計算の目的は、G に含まれる節点への枝を持つ節点の集合を求めることである。例として図 3 に、

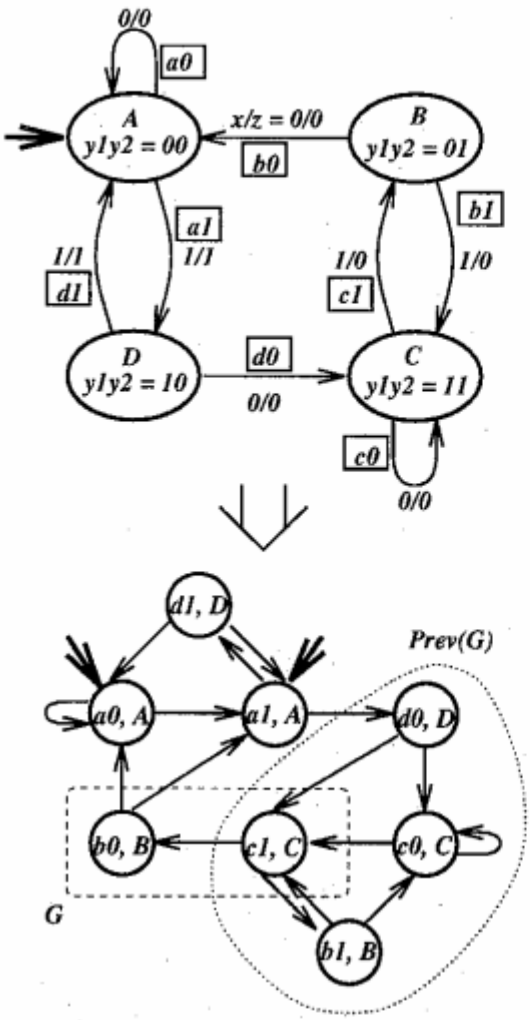

Fig. 3: 順序機械と Kripke 構造

G(破線で囲まれた節点集合) と $Prev(G)$(点線) の例を示す。

3 ボトムアップ逆像計算手法

ボトムアップ逆像計算手法 [9] の概略を説明する。

まず、$i = 1, 2, \cdots, m+1$ に対し、

$D(i, G(\boldsymbol{x}, \boldsymbol{y})) \stackrel{def}{=}$

$$\begin{cases} \exists \boldsymbol{x}', \boldsymbol{y}'.((\prod_{j=i}^{m} g_j) \cdot G(\boldsymbol{x}', \boldsymbol{y}')) & i = 1, 2, \cdots, m \\ \exists \boldsymbol{x}', \boldsymbol{y}'.(G(\boldsymbol{x}', \boldsymbol{y}')) & i = m+1 \end{cases}$$

で $D(i, G(\boldsymbol{x}, \boldsymbol{y}))$ を定義する。ただし、簡単のため $y'_j \equiv \delta_j$ を g_j とおいた。

1. 定義式の右辺を $x'_k (1 \leq k \leq l)$ で Shannon 展開して整理すると

$$D(i, G(\boldsymbol{x}, \boldsymbol{y})) = D(i, G_{\overline{x_k}}(\boldsymbol{x}, \boldsymbol{y})) + D(i, G_{x_k}(\boldsymbol{x}, \boldsymbol{y}))$$

2. 定義式の右辺を y'_i で Shannon 展開し、整理すると

$$\begin{aligned} D(i, G(\boldsymbol{x}, \boldsymbol{y})) &= D(i+1, G_{\overline{y_i}}(\boldsymbol{x}, \boldsymbol{y})) \cdot \overline{\delta_i(\boldsymbol{x}, \boldsymbol{y})} \\ &+ D(i+1, G_{y_i}(\boldsymbol{x}, \boldsymbol{y})) \cdot \delta_i(\boldsymbol{x}, \boldsymbol{y}) \end{aligned}$$

これにより、$D(i, G(\boldsymbol{x}, \boldsymbol{y}))$ に関する 2 通りの漸化式が与えられる。この漸化式が、BDD の Shannon 展開構造に則したものとなっている。$G(\boldsymbol{x}, \boldsymbol{y})$ の BDD は一般に入力変数 $[x_1, x_2, \cdots, x_l]$, 状態変数 $[y_1, y_2, \cdots, y_m]$ に対応するレベルの節点を持つが、$[y_1, y_2, \cdots, y_m]$ に関する変数順序が $y_1 \prec y_2 \prec \cdots \prec y_m$ (左辺が上位) であるとすると、$G(\boldsymbol{x}, \boldsymbol{y})$ の BDD をたどりながら、入力変数 ($[x_1, \cdots]$) に対応するレベルにおいては 1. の漸化式によって、状態変数 ($[y_1, \cdots]$) に対応するレベルにおいては 2. の漸化式によって再帰的に計算することにより、$D(1, G(\boldsymbol{x}, \boldsymbol{y}))$ を求めることができる。また、求める $Prev(G(\boldsymbol{x}, \boldsymbol{y}))$ は $D(1, G(\boldsymbol{x}, \boldsymbol{y}))$ に等しいので、G の根節点における計算結果が逆像となる。式 2.4 を式通りに計算すると、$\boldsymbol{x}', \boldsymbol{y}'$ に対応する $(l+m)$ 個の変数を BDD で陽に扱わなければならないが、ボトムアップ計算手法ではその必要がない。

なお、ハッシュテーブル構造に基づく演算キャッシュの利用により、G の BDD において、共有されている部分グラフにおける再計算を避けることができる。

4 一斉 traversal による並列化

前節で説明したアルゴリズムを、複数のプロセッサで実行する手法を考える。

まず、前節のアルゴリズムを逐次的に実行する様子を示したのが、図 4 である。矢線で示される順序の深さ優先 traversal を行ない、中間結果 D_a, D_b, D_c, D_d を生成し、最終結果の D を得る。ここで各中間結果および最終結果は、$D_a \rightarrow D_b \rightarrow D_c \rightarrow D_d \rightarrow D$ の順で生成されるが、このような全順序的な逐次生成は本質的ではなく、例えば D_b と D_c は、D_a が生成された後に並列に生成することが可能である。BDD 上の深さ優先 traversal における並列性を抽出する手法として、動的 Shannon 展開に基づく手法

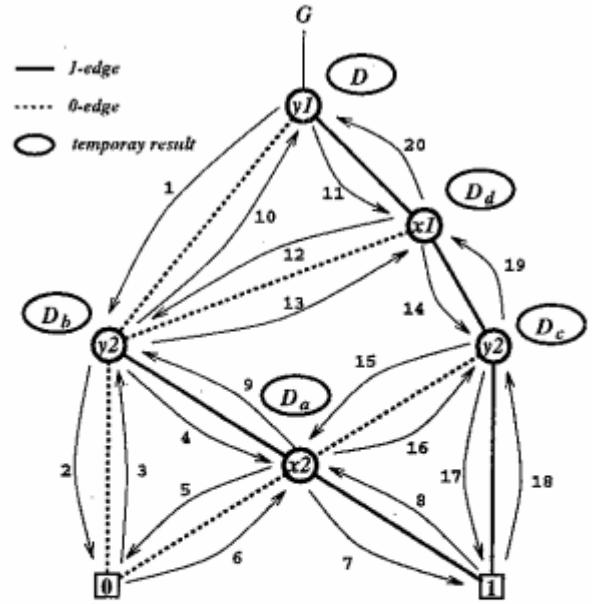

Fig. 4: 逐次 traversal

がある [6]。しかしながらこの手法では、BDD のように一般に再収れんを持つようなグラフの traversal に関しては、再収れん箇所で処理の並列性が失われる可能性がある。これは、各プロセッサの traversal 経路が静的に決定されていることに起因する。ところで、ボトムアップ逆像計算手法における一連の再帰的計算について、BDD G はその計算のデータ依存グラフとみなすことができる (例えば、Dd を計算するためには Db と Dc の値が必要であることが図 4 のグラフから見てとれる)。そこで、次のようにして各プロセッサが traversal 経路を動的に決定しながら計算を行なう手法を提案する。

4.1 BDD 節点における計算処理の並列性

G のある BDD 節点 N における計算は、N が入力変数 (x_k) の節点か状態変数 (y_i) の節点かによって

$$
\begin{aligned}
(x_k\text{節点}) \quad & D(i, N_{\overline{x_k}}(\boldsymbol{x},\boldsymbol{y})) + D(i, N_{x_k}(\boldsymbol{x},\boldsymbol{y})) \\
(y_i\text{節点}) \quad & D(i, N_{\overline{y_i}}(\boldsymbol{x},\boldsymbol{y})) \cdot \overline{\delta_i(\boldsymbol{x},\boldsymbol{y})} + \\
& D(i, N_{y_i}(\boldsymbol{x},\boldsymbol{y})) \cdot \delta_i(\boldsymbol{x},\boldsymbol{y})
\end{aligned}
$$

である。x_k 節点の場合の $D(i, N_{\overline{x_k}}(\boldsymbol{x},\boldsymbol{y}))$ と y_i 節点の場合の $D(i, N_{\overline{y_i}}(\boldsymbol{x},\boldsymbol{y}))$ を **0-edge オペランド** と呼び、同様に $D(i, N_{x_k}(\boldsymbol{x},\boldsymbol{y}))$ と $D(i, N_{y_i}(\boldsymbol{x},\boldsymbol{y}))$ を **1-edge オペランド** と呼ぶことにする。上記の計算を次のように 5 step に分けて考える。

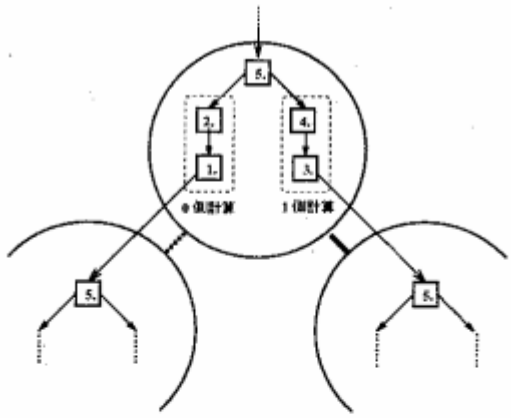

Fig. 5: 各 step のデータ依存

1. 0-edge オペランドの計算
2. $\overline{\delta_i}$ と 1. との論理積
3. 1-edge オペランドの計算
4. δ_i と 3. との論理積
5. 2. と 4. との論理和

x_k 節点の場合には step 2, 4 が不要であるが他は y_i 節点の場合と共通なので、以下では y_i 節点の場合と区別しないことにする。この各ステップのデータ依存関係は図 5 のグラフで示される。step 1.,2. を併せて **0 側計算**、step 3.,4. を併せて **1 側計算** と呼ぶことにする。図 5 からわかるように、0 側計算と 1 側計算の間の実行順序は規定されない。したがって、ある BDD 節点に複数のプロセッサが訪れた場合、

- それらが 0-edge オペランドの計算と 1-edge オペランドの計算とに適当に分かれ
- i-edge オペランド ($i \in \{0,1\}$) を持って返ってきたプロセッサが i 側計算の残りを実行し
- 後に完了した方の i 側計算を実行したプロセッサが step 5. を行なう

という方針により、個々の BDD 節点における計算の並列性を抽出することができる。図 6 では、節点 N を訪れたプロセッサのうち、P_0, P_1 がそれぞれ 0-edge オペランド、1-edge オペランドを持ち返って 0 側計算、1 側計算を行ない、P_0 が step 5. を計算して N における最終結果を持って N から離脱する様子を示している。このように 0-edge オペランド (1-edge オペランド) を持ち返ってくるプロセッ

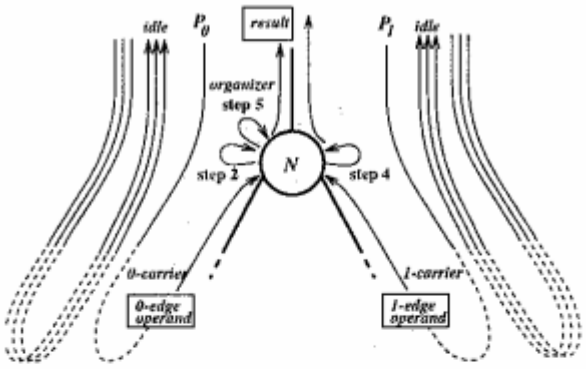

Fig. 6: 節点における処理

サ $P_0(P_1)$ を **0-carrier(1-carrier)** と呼び、それらのうち step 5. を行なって最終結果を計算するプロセッサ P_0 を **organizer** と呼ぶことにする。また、N を経由して下向きに traverse したプロセッサのうち、0-carrier にも 1-carrier にもならなかったプロセッサを **idle** と呼ぶ。

4.2 traversal 経路の動的決定手法

各プロセッサは、G の BDD の各節点における $\{0,1\}$ 側計算および step 5. に携わるべく、BDD 上を深さ優先で traverse する。本節では、この traversal における各プロセッサの経路決定手法について述べる。

4.2.1 経路決定のためのデータ

G のある BDD 節点を訪れたプロセッサが、次の traversal 方向を判断するためのデータ構造として、G の各 BDD 節点に、2 つのタグ T_0, T_1 および計算領域 tmp を付加する。各々のタグ T_0, T_1 は、それぞれ 0 側計算、1 側計算に対応しており、$\{init, opd, done\}$ の 3 値によってそれぞれの計算状況を表現する。タグのそれぞれの値の意味は次のようになる (T_0 の場合で説明する)。

init 0-carrier が未到着であることを示す。

opd 0-carrier P が到着済みで、P が 0 側計算を実行中であることを示す。1 側計算が既に終っている場合には step 5. も P によって実行される。すなわち P が organizer になる。

done 0 側計算の実行が完了していることを示す。T_1 も *done* である場合には、step 5. の実行も完了し、この節点における計算処理がすべて完了していることを表す。

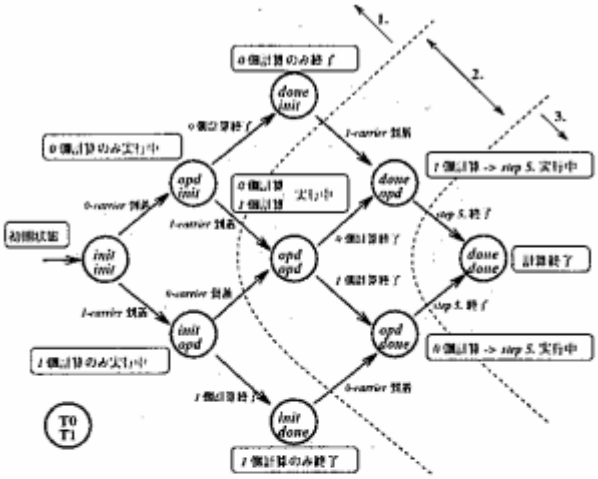

Fig. 7: タグの値の遷移および計算状況

2 つのタグはどちらも $init \rightarrow opd \rightarrow done$ という一方向の値の遷移を行なう。また、BDD 節点における計算の着手・完了にしたがって、一般には異なるプロセッサによって独立に T_0, T_1 の値の変更が行なわれる。値変更のタイミングによって、図 7に示されるような幾通りかの値の組合せを経由し、それぞれが図中に記した計算状況を表現する。

また、tmp は 2 通りの役割を担う。一つは、0 側計算あるいは 1 側計算を行なったプロセッサのうち先に終了した方が、その結果を (反対側の計算を行なうプロセッサが、続いて step 5. を行なうために) 一時的に格納するためであり、もう一つは、最終結果を格納するためである。後者は、2 で述べた演算キャッシュの役目も果たす。すなわち、再収れん箇所の再計算を避けるためにも用いられる。

以降では、$N.T_0$ という表記は節点 N の タグ T_0 を表す。$N.T_1, N.tmp$ についても同様である。

4.2.2 経路決定

各プロセッサは、G の根節点から深さ優先で一斉にグラフを traverse する。前述の付加データを用いた、動的な traversal 経路の決定について、下降時 (根節点から葉節点へ向かう方向) と上昇時 (葉節点から根節点へ向かう方向) とに分けて述べる。

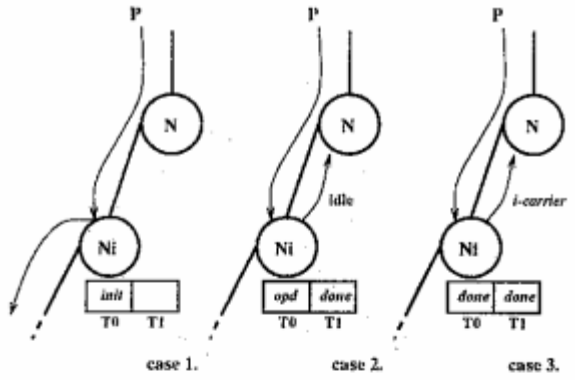

Fig. 8: 下降時の経路決定

下降時 いま、プロセッサ P が、ある節点 N の i-edge 側を traverse して節点 N_i を訪れたとする。このときの P の traversal 方向および動作は、N_i の計算状況によって次の 3 通りに分かれる (図 8)。

1. $N_i.T_0, N_i.T_1$ の少なくとも一方が *init* のとき: これは、N_i における計算のオペランドがまだ揃っていないことを示している。したがって P は未計算のオペランドを計算しに行けばよい。すなわち、*init* であるタグ $N_i.T_j (j \in \{0,1\})$ を選んで、その節点における j 側計算に携わるため、N_i の j-edge 側を traverse する。
2. $N_i.T_0 \neq init$ かつ $N_i.T_1 \neq init$ かつ $N_i.T_0, N_i.T_1$ の少なくとも一方が *done* でないとき: これは、N_i における計算の両オペランドが既に揃っており、かつ step 5. の計算は他のプロセッサが行なうことになっていて、しかもまだ結果が得られていないことを示している。すなわち N_i を根節点とするサブグラフには有効な計算がもはや残されていない。したがって P は idle として N に帰り、ジョブの探索を続ける。
3. $N_i.T_0 = N_i.T_1 = done$ のとき: これは N_i における計算が step 5. まですべて完了しており、結果が得られていることを示している。したがって P は、$N_i.tmp$ の値を持って i-carrier として N に帰る。

なお、この 3 通りの場合分けは図 7 の、破線を境界線とする 3 つの領域 1., 2., 3. に対応している。

上に挙げた経路決定には、2 つの問題点がある。まず一つは 1. の場合で、$T_0 = T_1 = init$ であった場合に、どちらのオペランドを計算しに行くかという問題である。これについては

- 各節点に 1 bit のスイッチを付加し、プロセッサはスイッチを見て選択を決定した後、スイッチを反転させる方法。これによれば、プロセッサは訪れた順に、交互に 0-edge 側と 1-edge 側に振り分けられる。
- $\frac{1}{2}$ の確率で、random に選択する方法。
- プロセッサ番号の偶奇、あるいは番号の二進表現のビット列を利用して選択する方法。

などが考えられる。どれが効率的かは BDD の構造に依存し、一概には判断できないと思われる。

二つめは、2. の場合である。上に示した経路判定は、プロセッサがまず N を訪れ、N における計算のオペランドを取得するために N_i を訪れ、オペランドがまだ計算中であることを知ったとき、そこは置き去りにして他のジョブを探しに行くことに相当する。これに対して、オペランドの計算が終了するまで wait する方法も考えられる。この場合、同じ経路で traverse してきた複数のプロセッサが同一のオペランドを wait するという状況を避けるための操作が必要になる。

上昇時 次に、P が節点 N の i-edge に沿って上昇してきたときの経路判定および動作について述べる。上昇してきた P は、idle か carrier かのいずれかであるので、この 2 通りで場合分けする (図 9)。

1. P が **idle** のとき: $N.T_{\bar{i}}$ を調べ、*init* であれば $\bar{i}$ 側計算を手伝いに行く。*init* でなければ N より下にジョブは残っていないので、idle として上に上がってジョブを探しに行く。
2. P が **i-carrier** のとき: $N.T_i$ を調べ、*init* でなければ既に先着の i-carrier がいるということなので、それ以降は idle となって 1. に従う。$N.T_i = init$ であれば P は N に到着した最初の i-carrier ということなので、$N.T_i$ を *opd* に変更し i 側計算を行なう。i 側計算完了後に $N.T_{\bar{i}}$ を調べ、*done* であれば以降は N における organizer として step 5. の計算完了後に T_i を *done* に変更し、その後計算結果を *tmp* に格納して、その値を持って carrier として N の親ノードに向けて上昇する。$N.T_{\bar{i}} \neq done$ であれ

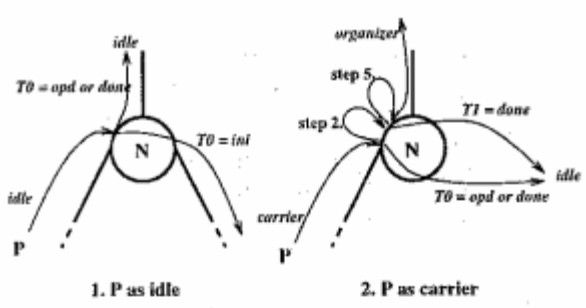

Fig. 9: 上昇時の経路決定

ば、$\tilde{i}$-carrier が organizer になってくれるので、P はそれ以降 idle となって 1. に従う。

4.3 アルゴリズム

以上の考えをまとめると次のようなアルゴリズム ParallelInverseImage が得られる (図 10)。これは、p 個のプロセッサの各々を アルゴリズム Dynamic-Traversal にしたがって traverse させ、頂点の節点の organizer が結果を返すのを待つものである。

5 おわりに

本稿では、順序回路の形式的設計検証に用いられるボトムアップ逆像計算処理の、共有主記憶型並列計算機向きの並列化手法を提案した。これは、BDD の再帰的グラフ構造を利用して定義されたような再帰計算においては、その BDD 自身がデータ依存グラフになっていることに着目し、各プロセッサが BDD をたどりながらジョブを探索、取得するものである。BDD の再収れん箇所において並列性が失われることを避けるため、各 BDD 節点に付加したデータによりその節点の計算状況を表現し、プロセッサは計算状況に応じて探索経路を動的に設定する。

記号モデル検査においては、逆像計算が処理時間に対して支配的であるため、この部分の高速化は検証処理の高速化に直接結び付く。現在、手法の有効性を確認すべく SGI Challenge XL (MIPS R4400 150 MHz × 36, 主記憶 2 Gbyte) を用いた実装を急いでいる。

本稿では提案手法の基本的な考えを示したが、今後は対処すべき問題点を明らかにし、改良を重ねていく必要がある。例えば、G の BDD が幅の大きい (fat な) グラフであった場合、逆像計算は並列性を多く含んだものとなり本手法が有効に働くが、逆に縦に長いグラフの場合逆像計算は本質的に逐次処理となってしまい、並列計算の意味が薄くなってしまうことが、直観的に予想される。また、実装に当たっては、プログラムの正当性を注意深く考慮しながらロック操作を削減することも課題となる。

また、本手法の応用としては、apply 演算が目標に挙げられる。apply 演算とは BDD を用いた二項論理演算 (論理和、論理積 など) の総称であり、BDD を用いた論理関数処理の最も基本的な処理である。apply 演算は 2 つの BDD を同時にたどりながら実行されるため、本手法の拡張の検討が必要であると思われる。しかしながら apply 演算のレベルで効率的な並列化手法が適用できれば、BDD 処理系のユーザ・レベルから透過的な並列 BDD 処理系を実現することができる。これは既存の BDD アプリケーションをそのまま並列化することができることを意味するため、意欲を持って取り組むべき課題である。

謝辞

日頃から有益な御助言、御討論をいただく本学情報科学研究科渡邊研究室の諸氏に深謝いたします。特に、ボトムアップ逆像計算手法についてコメントをいただいた中江達哉氏に感謝いたします。

参考文献

[1] Randal E.Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comput.*, Vol. C-35, No. 8, pp. 677–691, Aug. 1986.

[2] 平石裕実, 濱口清治. 論理関数処理に基づく形式的検証手法. 情報処理, Vol. 35, No. 8, pp. 710–718, Aug. 1994.

[3] J.R.Burch, E.M.Clarke, and D.E.Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *Proc. of 28th ACM/IEEE Design Automat. Conf.*, pp. 403–407, June 1991.

[4] J.R.Burch, E.M.Clarke, and K.L.McMillan. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. of 27th ACM/IEEE Design Automat. Conf.*, pp. 46–51, June 1990.

[5] Hervé J.Touati, Hamid Savoj, Bill Lin, Robert K.Brayton, and Alberto Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. of IEEE Int. Conf. on Computer-Aided Design*, pp. 130–133, Nov. 1990.

[6] Shinji KIMURA, Tsutomu IGAKI, and Hiromasa HANEDA. Parallel Binary Decision Diagram Manipulation. *IEICE Trans. Fundamentals*, Vol. E75-A, No. 10, pp. 1255–1262, Oct. 1992.

[7] 小原隆司, 石浦菜岐佐, Hoyong CHOI, 白川功, 本原章. ブール単一化を用いた像計算. 電子情報通信学会春季大会 SA-2-4, pp. 1–(372–373), Mar. 1993.

[8] 松永裕介, Patrick C.McGeer, 藤田昌宏. 2 分決定グラフを用いた推移的閉包計算アルゴリズムと形式的検証への応用. Technical Report VLD92-76, 電子情報通信学会, Jan. 1992.

[9] 中江達哉, 平石裕実, 濱口清治. 順序機械の設計検証における BDD 処理に適した逆像計算法. 電子情報通信学会春季大会 SA-2-5, pp. 1–(374–375), Mar. 1993.

[10] S.Minato, N.Ishiura, and S.Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *Proc. of 27th ACM/IEEE Design Automat. Conf.*, pp. 52–57, June 1990.

```
[ParallelInverseImage]
  入力: 節点集合 G, 状態遷移関数 δ
  出力: G の δ による逆像 Prev(G)
begin
  for 1 ≤ i ≤ p pardo begin
    G'[i] ← DynamicTraversal(G, δ)
    if (G'[i] ≠ nil) then
      Prev(G) ← G'[i]
  end
end.

[DynamicTraversal]
  入力: 節点集合 G, 状態遷移関数 δ
  出力: D(1, G) または nil
begin
  if G = 0 or G = 1
    return G
  N ← top node of G
  // downward traversal
  while ∃i.(N.T_i = init) do
  begin
    choose i s.t. N.T_i = init
    D ← DynamicTraversal(N_i, δ)
    if D ≠ nil then
    begin
      C ← Carrier(N, i, D, δ)
      if C ≠ nil then
        return C
    end
  end
  if N.T_0 = N.T_1 = done then
    return N.tmp
  else
    return nil
end.

[Carrier]
  入力: BDD 節点 N, 枝種 i ∈ {0, 1}
        i-edge オペランド D
        状態遷移関数 δ
  出力: N における計算結果 (organizer)
        または nil(それ以外)
begin
  if N.T_i ≠ init then
    return nil
  lock(N.T_i)
  N.T_i ← opd
  N.tmp ← i 側計算の結果
  if N.T_ī = done then
  begin
    // act as organizer
    N.tmp ← N.tmp + D (step 5.)
    N.T_i ← done
    unlock(N.T_i)
    return N.tmp
  end else begin
    N.tmp ← D
    N.T_i ← done
    unlock(N.T_i)
    return nil
  end
end.
```

Fig. 10: 並列逆像計算