ハイパフォーマンス
コンピューティング 57−13
（1995. 8. 24）

閾値関数の BDD の並列実装

丹羽 純平　　今井 浩

東京大学大学院理学系研究科情報科学専攻

Akers によって提案された二分決定グラフ（BDD）は、論理関数を表現する DAG である。Bryant による BDD 間の論理演算を効率良く行なうアルゴリズムにより、BDD は VLSI CAD 等の様々な分野で使われるようになった。近年では組合せ問題にも応用されるようになった。本稿では、閾値関数を表す BDD を構築する新しいアルゴリズムを提案する。動的計画法を用いた前処理を行なった後で BDD を出力サイズに比例した計算量でトップダウンに構築するものである。我々は更に上述のアルゴリズムを並列化してそれを AP1000+ 上で実装し、実験により既存のアルゴリズムで構築できなかったサイズの BDD を迅速に構築できることを示した。

PARALLEL CONSTRUCTION OF A BDD REPRESENTING A THRESHOLD FUNCTION

Junpei NIWA　　Hiroshi IMAI

Department of Information Science, Faculty of Science, the University of Tokyo

A Binary Decision Diagram (BDD), proposed by Akers, is a DAG representing a boolean function. Through the development of the efficient algorithms of boolean operations on BDDs by Bryant, it has been used in various fields, such as VLSI CAD. Recently it has been applied to combinatorics. We propose a new algorithm of constructing a BDD representing a threshold function. After a preprocessing stage using a dynamic programing, we construct the BDD in a top-down manner with time proportional to the output size. Furthermore we parallelize the algorithm and implement it on AP1000+, and the experiment shows that the BDD which could not be constructed by the existing algorithm is constructed quickly.

1 導入

論理関数をコンパクトに表現し操作することは重要である。BDDは論理関数をコンパクトに表現した有向無閉路グラフ(DAG)のことである[1], [2]。論理演算はDAGを操作することで効率良く行なえ、解の数え上げ、最大重みの解の探索等がDAGのサイズに比例した手間で行なえる。BDDは変数順によりサイズが変化し、サイズを小さくする変数順を求めることが重要となる。しかし一般の論理関数に対してBDDのサイズを最小にする変数順を見つける問題は難しい[3]。

我々は閾値関数を表すBDDを考える。閾値関数とは重み付けされた入力の和が閾値をこえるかどうかで出力が定まる論理関数である。閾値関数のネットワークにより任意の論理関数を実現する[4]等といった論理的研究もなされている。また最適化問題等の制約式によく現れ、解の数が数論上意味を持つ[5]ことが知られている。閾値関数を表すBDDは理論的にも研究されていて、どのような変数順に対してもBDDのサイズが指数サイズになる閾値関数が存在する[8]。

実際に従来のアルゴリズムを用いて閾値関数を表すBDDを構築するには2つの問題がある。

- 論理式への変換が伴う
  例えば"$2v+3w+x+3y+4z \le 7$"を表すBDDを構築する際、これを論理式に変換する必要があり、この場合は"$(\bar{v}\wedge\bar{w}\wedge\bar{x})\vee(\bar{v}\wedge\bar{x}\wedge\bar{y})\vee(\bar{v}\wedge\bar{z})\vee(\bar{w}\wedge\bar{y})\vee(\bar{w}\wedge\bar{z})\vee(\bar{y}\wedge\bar{z})$"となり複雑で冗長になってしまう。このため、数式も扱えるパッケージが開発された(BEM-II)[6]。

- 計算の途中で計算が爆発してしまう[2]
  既存のアルゴリズムでは、小さなBDDを作ってそれらに論理演算を適用していくために、最終的なBDDが小さくても計算の途中で計算が爆発してしまうことがある。

そこでこの欠点を克服するべく、我々は幅優先アルゴリズム[7]を使って閾値関数を表すBDDを構築する新しいアルゴリズムを提案する、つまり動的計画法を用いて前処理の表を計算した後、出力サイズに比例した計算量でBDDを構築するアルゴリズムを提案する。

我々は更にこのアルゴリズムの分散並列化を試みる。前処理の部分はパイプライン化してその表を分散させ、BDDもなるべく分散させて構築することを試みた。新しいアルゴリズムと並列処理により、高速化が可能となり、これまで構築できなかったサイズのBDDを構築することができた。今回はSPMD+MP型の並列計算機AP1000+上でアルゴリズムを実装し実験を行ない、その結果を示す。

2 準備

2.1 OBDD

全順序付き二部決定グラフ(OBDD)とは論理関数を表す有向無閉路グラフのことである[2]。論理関数$f(x_1,\cdots,x_n)$を表すOBDDは5個組$(X, N, root, label, edge)$からなる。

$X = \{x_1, x_2, \cdots, x_n\}$ (全順序付き変数の集合)
$x_1 < x_2 < \ldots < x_n$
$N = N_v \cup N_c$ $(N_v \cap N_c = \emptyset)$ (ノードの集合)
N_v: 変数ノード　N_c: 定数ノード $\{c_0, c_1\}$
$root \in N$ (ルートノード)
$label : N \longrightarrow (X \cup \{0,1\})$
$edge : N_v \times \{0,1\} \longrightarrow (N_v \cup N_c)$ (枝の集合)
$\forall v \in N_v$ $(b \in \{0,1\})$
$label(edge(v,b)) \in \{0,1\}$ または
$label(v) < label(edge(v,b))$
(変数の全順序に矛盾しない)

$root$から定数ノードに至る全てのパス上で同じ変数がラベル付けされたノードは高々1個しかない。また各パス上の変数順はXの順序に矛盾しない。Xのi番目の変数を$\pi[i]$と表し、$\pi = (\pi[1], \cdots, \pi[n])$を$X$の変数順と定義する(つまり$x_i = \pi[i]$)。もし$label(v) = \pi[i]$ $(v \in N_v)$ならばvの$level$はiであると定義する($level(v) = i$と表す)。定数ノードの$level$は$n+1$とする。

ルートノードがvであるOBDDは次のような論理関数Fを表す。

- vが定数ノードの時
  $F(v) = label(v)$
- vが変数ノードの時
  $F(v) = \overline{label(v)} \cdot F(edge(v,0)) + label(v) \cdot F(edge(v,1))$

OBDDのサイズは$|N_v|$で与えられる。

変数の割り当てが与えられた時、OBDDが表す関数の値はルートノードから定数ノードまで次のようにグラフを探索することで得られる。変数ノードvにおいては、ラベル付けされている変数の値0,1に応じて、$edge(v,0)$、$edge(v,1)$をたどっていく。このようにしてたどり着いた定数ノードのラベル0または1が関数の値となる。

次に冗長なノードと等価なノードを定義する。$edge(v,0) = edge(v,1)$なる変数ノードvを冗長なノードという。2つのノードu、vが次の2つの条件のいずれかを満たす時等価なノードであるという。

1. uもvも定数ノードで
   $label(u) = label(v)$
2. uもvも変数ノードで
   $label(u) = label(v)$, $edge(u,0) = edge(v,0)$, $edge(u,1) = edge(v,1)$

等価なノードがないOBDDをQOBDDといい、等価なノードと冗長なノードがないOBDDをROBDDと言う。

2.2 閾値関数

n 個の重さを持った品物の集合 (各品物の重さは w_j(正整数)) と閾値 t(正整数) が与えられた時、適当に品物を選択して、その重さの和が閾値以下なら真を返す論理関数を閾値関数と呼ぶ。式で書けば次のようになる。
(x_j は品物 j を選ぶ時のみ 1 になる論理変数)

$$\tau(x_1,\ldots,x_n) = \begin{cases} 1 & (\sum_{j=1}^{n} w_j x_j \le t) \\ 0 & (\sum_{j=1}^{n} w_j x_j > t) \end{cases}$$

閾値関数の QOBDD と ROBDD の例が図 1にある。

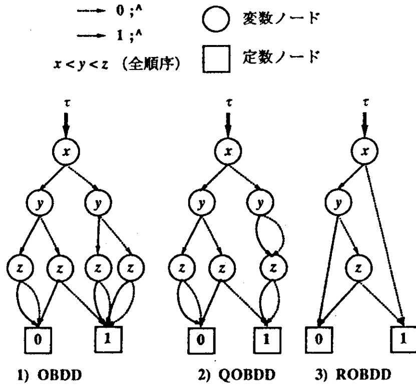

図 1: $\tau = (5x + 4y + 3z \le 7)$ の OBDD の例

3 閾値関数を表す QOBDD の構築

3.1 幅優先アルゴリズム

我々はまず QOBDD をトップダウンに構築するアルゴリズム (幅優先アルゴリズム) を紹介する [7]。これは与えられた論理関数 f を表す QOBDD を与えられた変数順 π で幅優先に構築するアルゴリズムである。以下では真を T、偽を F で表す。

procedure CONSTRUCT(f,π)
step1 ルートノード v を作る ($F(v) = f$)
step2 $table \leftarrow \emptyset$ $i \leftarrow 2$
step3 $level(u) = i-1$ なる全てのノード u に対して以下の操作を繰り返す。
3.1 $g \leftarrow RESTRICT(F(u), \pi[i-1], 0)$
3.2 if ($MEMBER(g) = T$)
then{$edge(u,0) = u'$ ($F(u') = g$)}
else{ 新しいノード u' を作り、
$edge(u,0) = u'$, $table \leftarrow table \cup \{g\}$}
3.3 $g \leftarrow RESTRICT(F(u), \pi[i-1], 1)$
3.4 if ($MEMBER(g) = T$)
then{$edge(u,1) = u'$ ($F(u') = g$)}
else{ 新しいノード u' を作り、
$edge(u,1) = u'$, $table \leftarrow table \cup \{g\}$}
step4 if ($table = \{0,1\}$)
then end
else{$i \leftarrow i+1$, $table \leftarrow \emptyset$, goto **step3**}
(ただし、$RESTRICT(g,x,b)$ は $g|_{x=b}(b \in \{0,1\})$ を返す関数、$MEMBER(g)$ は $g \in table$ なら T を返す関数とする。)

幅優先アルゴリズムは上からレベル毎に進んでいく。新しいノードが作られる時はそのノードが表す関数が $table$ に登録され、新しい枝を引く時はそれが指すノードが $table$ に存在するか調べられる。よって等価なノードは生じない、つまり QOBDD が構築される。

このアルゴリズムは RESTRICT と MEMBER が正しく行なわれる限り、論理関数がどのような形で表現されていても正しく動く。しかし、実行時間は RESTRICT と MEMBER に依存する。つまり RESTRICT と MEMBER が迅速に行なわれれば、output-size sensitive な計算量が保証される。

3.2 閾値関数への適用

閾値関数 τ と変数順 π が与えられた時、この幅優先アルゴリズムを用いて τ を表す QOBDD を構築する。

3.2.1 閾値関数の表現

閾値関数を表す QOBDD を考える。ルートノードからノード v へ至るパスの 1 個を P_v とする。$edge(u,1)$ が P_v 上にあるようなノードの集合を s_v とする。
$t_v = t - \sum_{u \in s_v} w_{level(u)}$ と定義すれば、各ノード v が表す論理関数は

$$F(v) = \begin{cases} 1 & (\sum_{j=level(v)}^{n} w_j x_j \le t_v) \\ 0 & (\sum_{j=level(v)}^{n} w_j x_j > t_v) \end{cases}$$

の形で記述できる。つまり ($level(v), t_v$) の組が閾値関数の表現であると考えることができ、このことを $F(v) = (level(v), t_v)$ で表す。

3.2.2 RESTRICT

$level(v) = i$ を満たすノード v が表す論理関数は $F(v) = (i, t_v)$ で記述できるから、その RESTRICT は次のようになる。

$$RESTRICT(F(v), \pi[i], 0) = (i+1, t_v)$$
$$RESTRICT(F(v), \pi[i], 1) = (i+1, t_v - w_i)$$

3.2.3 MEMBER

MEMBER を効率良く実行するには関数の等価性判定を高速に実行しなければならない。QOBDD を構成する前に次のような前処理を行なう。

$A(p,q)$ を「$\{w_p, w_{p+1}, \ldots, w_n\}$ の適当な部分集合を選んだ時その和がちょうど q になる」命題と定義する。すると次の漸化式が成立する。

$$A(n,q) = T \text{ iff } q = 0 \vee q = w_n$$
$$A(p,q) = A(p+1,q) \vee (w_p \leq q \wedge A(p+1, q-w_p))$$
$$(1 \leq p < n,\ 0 \leq q \leq t)$$

A が $n \times (t+1)$ の2次元の表を表していると考えれば、動的計画法で $O(nt)$ 時間 $O(nt)$ 領域で表を作成できる。その後もう1個の $n \times (t+1)$ の2次元の表 B を作成する。各要素は次の式を満たす、$B(p,q) = \max\{j \mid j \leq q \wedge A(p,j) = T\}$。表A、Bの例が表1にある。ここで $element(p,q)$ を

表 1: 表 A(上) 表 B(下) の例
品物の重さの集合 $=\{w_1, w_2, w_3, w_4\}$ で各重さが
$w_1 = 6, w_2 = 3, w_3 = 2, w_4 = 4$ で閾値 $t = 8$ の場合

p \ q	0	1	2	3	4	5	6	7	8
1	T	F	T	T	T	T	T	T	T
2	T	F	T	T	T	T	T	T	F
3	T	F	T	F	T	F	T	F	F
4	T	F	F	F	T	F	F	F	F

p \ q	0	1	2	3	4	5	6	7	8
1	0	0	2	3	4	5	6	7	8
2	0	0	2	3	4	5	6	7	7
3	0	0	2	2	4	4	6	6	6
4	0	0	0	0	4	4	4	4	4

もし、$q < 0$ なら -1 を返し、そうでなければ $B(p,q)$ を返す関数と定義する。すると以下の命題が成立する [9]。

命題 1 同じ $level\ i$ にある2つのノード u, v が等価なノードであるための必要十分条件は
$element(i, t_u) = element(i, t_v)$ である。

これにより、関数の等価性判定が定数時間で行なえる。

4 並列化と実験

今回は変数順 π として降順を採用した。
$(w_1 > w_2 > \ldots > w_n)$

4.1 前処理の表

3.2.3の漸化式から逐次の場合には2重のforループで表を作成する訳だが、

```
for (p = n ; p > 0 ; p = p - 1)
  for(q = 0 ; q ≤ t ; q = q + 1){
    A(p,q) = ...
    B(p,q) = ...
  }
```

各イテレーション間に依存関係があり、単純に並列化はできない。そこで表を横方向にプロセッサー台数でブロック分割して各プロセッサーが各ブロックを所有することにした。各プロセッサーが2番目のループをパイプライン化して計算することで並列化可能となる (図2)。各プロセッサー

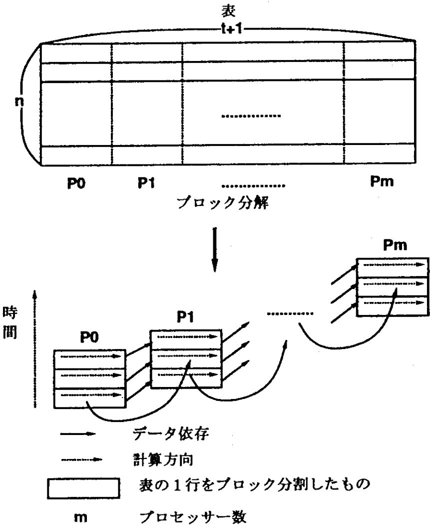

図 2: 計算の流れ

が表を分散して所有するから、大きいサイズの表を作成できるメリットがある。しかし、実際QOBDDの作成時にノードの等価性判定を行なう際、つまり $element(i, t_v)$ を計算する際、他のプロセッサーに属する表の値を引かなければならない場合が生じてくる。この部分は各プロセッサーが表全体を所有した場合より遅くなる (表2)。

またブロックのサイズが小さくても通信と計算がオーバーラップさせられない。そこで小さい問題に対しては仮想プロセッサーを考え、表を仮想プロセッサー数 (VP) でブロック分割した。仮想プロセッサー (vp) を実際のプロセッサー (p) に $p \bmod VP = vp$ を満たすようにマッピングした。これにより複数のプロセッサーが表の同じブロックを共有することになる。実験の結果が表3にある。

4.2 QOBDDの構築

QOBDDの各ノード v を $(level(v), element(level(v), t_v))$ の組によって、どのプロセッサーに割り振るか決める。これにより等価なノードが複数のプロセッサーで生成されることはなくなる。QOBDDを利用して最適化問題を解く (QOBDDの枝に重みを付けて最大重みの解の探索を行なう)

表 2: 表を分散させた場合(下)とそうでない場合(上)との比較(各品物の重さは10bitの乱数、$t=$全品物の重さの和の半分、プロセッサーの台数は256台)

品物の数	前処理の表(秒)	QOBDDの構築(秒)	|QOBDD|
100	2.187e+00	3.347e-01	707597
200	計測不能	計測不能	計測不能

品物の数	前処理の表(秒)	QOBDDの構築(秒)	|QOBDD|
100	7.998e-01	7.686e-01	707597
200	1.534e+00	3.857e+00	3734718
300	2.153e+00	8.587e+00	8126630
400	2.854e+00	1.647e+01	15015746
500	3.446e+00	2.487e+01	23157763
600	4.441e+00	3.914e+01	34700876
700	5.496e+00	5.345e+01	48183785
800	6.560e+00	7.285e+01	62952029

表 3: 仮想プロセッサーの導入の効果(各品物の数は100、品物の重さは10bitの乱数、$t=$全品物の重さの和の半分、プロセッサーの台数は256台)

仮想プロセッサー数	前処理の表(秒)	QOBDDの構築(秒)
256	8.114e-01	7.671e-01
128	7.554e-01	7.933e-01
64	7.172e-01	9.259e-01

表 4: ブロック分解とサイクリック分解のQOBDD構築時間の比較(品物の重さは10bitの乱数、$t=$全品物の重さの和の半分、プロセッサーの台数は256台、単位は秒。)

品物の数	ブロック分解(秒)	サイクリック分解(秒)
100	9.743e-01	7.686e-01
200	4.106e+00	3.857e+00
300	8.728e+00	8.587e+00
400	1.552e+01	1.647e+01
500	2.330e+01	2.487e+01
600	3.442e+01	3.914e+01
700	計測不能	5.345e+01
800	計測不能	7.285e+01

ことを考えると、QOBDDを構築した後に探索するよりは、構築しながら探索を行なう方が速い。そのためにQOBDDを作る際、各レベル毎に同期を取る必要が生じる。

具体的には組(i,t_v)が通信によって送られた時、それを表すノードが存在するか調べる。もし、存在しなければノードを作り、次のレベルの変数に0,1を代入した部分問題$(i+1, element(i+1,t_v))$、$(i+1, element(i+1,t_v-w_i))$をそれぞれ作成し[1]、然るべきプロセッサーに送る。これを各レベル毎に同期を取って実行していく。

今回は組(i,t_v)からプロセッサーへの写像関数として次の2つを実験してみた。(以下mはプロセッサー数、pはプロセッサー番号)

1. ブロック分解
   プロセッサー番号pのプロセッサーはt_vが連続した範囲$(\lfloor\frac{t+1}{m}\rfloor p \leq t_v < \lfloor\frac{t+1}{m}\rfloor(p+1))$にあるノード$v$を管理する。

2. サイクリック分解
   プロセッサー番号pのプロセッサーは
   $t_v \bmod m = p$を満たすノードvを管理する。

この2つの違いを調べたものが表4にある。この時のロードバランスを表しているのが図3である。ノードの数が少ない時はロードバランスのいいサイクリック分解の方が速い。しかし、ノードの等価性判定の際、サイクリック分解はブロック分解より他のプロセッサーが管理する表を引く可能性が高くなる、つまりそれだけ時間がかかる可能性が高くなる。従ってノードの数が多くなると、この差が効いてきてブロック分解の方が速くなる。しかし、ブロック分解はバランスが悪いのでサイクリック分解よりも先に計算不能になってしまう。

最後に台数効果を示したのが図4であるが、これは前処理の部分は除いて実際にQOBDDを構築する部分のみである。組(i,t_v)からプロセッサーへの写像関数として、サイクリック分解を用いた。図の"ideal"は通信のオーバーヘッドもなくノードは完全に分散させた場合の台数効果である。"OURS"は我々の実験による実際の台数効果である。

5 結論

閾値関数を表すBDDをトップダウンに構築するアルゴリズムを提案し、その並列化を行なってAP1000+上で実装した。これにより、これまで構築できなかったサイズのBDDを構築できるようになった。ゆえに、変数順と閾値関数を表すBDDのサイズとの関係を実際に解析することが可能となり、閾値関数を表すBDDの良い変数順を発見する道が開かれたと思われる。

しかし問題点がまだいくつかある。組(i,t_v)からプロセッサーへの写像関数として、今回実験したものよりもっとふさわしいものがあるかも知れない。それによりもっと高速にBDDを作成できる可能性がある。また出来上がったBDDと他のBDDとのANDやORを取る方法が今の段階では

[1]ここで他のプロセッサーに属する表を引く可能性が生じる

不明な部分がある。本稿で等価性判定をトップダウンに行なっている部分に対応してボトムアップに等価性判定ができれば、AND 等の演算を高速に並列実行できると思われる。これらの問題点を克服するのが今後の課題である。

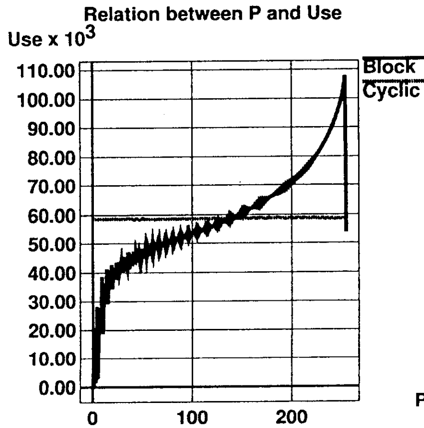

図3: ロードバランス (各品物の数は 400、品物の重さは 10bit の乱数、t = 全品物の重さの和の半分、横軸はプロセッサー番号、縦軸はプロセッサーの使用回数)

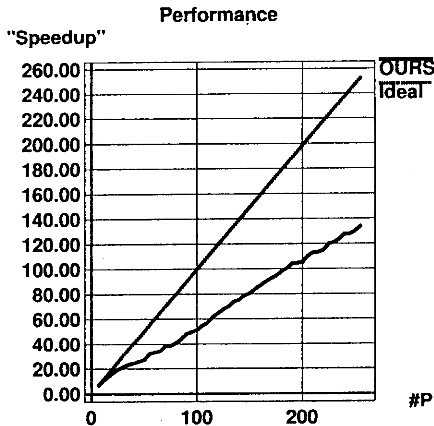

図 4: 台数効果 (各品物の数は 100、品物の重さは 10bit の乱数、t = 全品物の重さの和の半分)

参考文献

[1] S. B. Aker "Binary Decision Diagrams", *IEEE Trans. on Computers*, Vol.C-27 (1978), pp.509-516.

[2] R. E. Bryant "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, Vol.C-35 (1986), pp.677-691.

[3] S.J.Friedman and K.J.Supowit "Finding the Optimal Variable Ordering for Binary Decision Diagrams", *IEEE Trans. on Computers*, Vol.C-39 (1990), pp.710-713.

[4] S. Muroga "Threshold Logic and Its Application" *Wily-Interscience* (1971).

[5] Yanase, H.H., Soma, N.Y. "A New Enumeration Scheme for The Knapsack Problem", *Discrete Appl. Math.*, Vol.18, No.2, pp.235-45 (1987).

[6] S. Minato "BEM-II: An Arithmetic Boolean Expression Manipulator Using Binary Decision Diagrams", *IEICE Trans. Fundamentals*, Vol.E.76-A (1993), pp.1721-1729.

[7] S. Tani and H. Imai "A Reordering Operation for an Ordered Binary Decision Diagram and an Extended Framework for Combinatorics of Graphs", *ISAAC'94, LNCS*, Vol.834, pp.575-583 (1994).

[8] K. Hosaka, Y. Takenaga and S. Yajima "On the Size of Ordered Binary Decision Diagrams Representing Threshold Functions", *ISAAC'94, LNCS*, Vol.834, pp.584-592 (1994).

[9] J. Niwa "Constructing A Binary Decision Diagram of a Threshold Function", *Senior Thesis*, University of Tokyo (1995).