数理解析研究所講究録
第 790 巻 1992 年 134-140

二分決定グラフの並列構成アルゴリズムについて

木村 晋二、井垣 努、羽根田 博正　　神戸大学工学部

本稿では、論理設計支援ツールで広く用いられている二分決定グラフの並列構成アルゴリズムについて述べる。論理式からそれに対応する二分決定グラフを構成するときに、並列実行できる演算を論理演算の依存関係から求める方法と、並列実行できる演算が少ないときに、それらの演算をさらに並列実行できる部分に分割して並列度をあげる方法を示す。構成アルゴリズムのベースとしては、否定枝付きの共有二分決定グラフを用い、25 CPU で 20 倍強の高速化を達成した。

1　はじめに

論理関数を効率良く表す方法として、近年二分決定グラフ (Binary Decision Diagram, BDD) が注目され、広く研究されている ([4], [3], [8])。多くの実用的な論理関数に関して、非常に効率良く関数を表現できるので、二分決定グラフの処理に関する時間は少ないが、問題によっては、問題のサイズの指数に比例する記憶量と処理時間を必要とする ([5])。

そこでここでは、二分決定グラフの処理の並列化による処理の高速化について考察する。ここでは、共有記憶型の並列計算機向きのアルゴリズムと、その実現について述べる。以下ではとくに、論理式により表された論理関数に対応する二分決定グラフを求める問題について考察する。なお、ここでは、文献 [8] の否定枝付き共有二分決定グラフの構成法をもとに、並列アルゴリズムを考える。

二分決定グラフの並列化手法については、文献 [6] に基本的なアイデアが述べられている。ここでは、そのアイデアのもとに並列化を行なった。それは、論理演算の並びの中の各演算を異なるプロセッサに割り当てることと、同時に実行できる論理演算が少ない時にそれらの論理演算を分割して異なるプロセッサに割り当てて並列度を上げるという手法である。ここではさらに、共有変数へのロックを最小限にとどめることにより並列実行の効果を高めている。

以下 2 節で二分決定グラフの直列構成アルゴリズムについて述べ、3 節で並列構成アルゴリズムについて述べる。

2　二分決定グラフの構成

2.1　論理式と論理関数

n 変数論理関数は、$\{0,1\}^n$ から $\{0,1\}$ への関数である。論理関数を表すには、種々の方法があるが、以下に示すような論理式表現が用いられることが多い。

定義 1 (論理式)

1. 入力論理変数 $x_1, x_2, \ldots, x_n$ は n 変数論理式である。
2. f_1, f_2 が n 変数論理式であるとき、$(\neg f_1)$, $(f_1 \cdot f_2)$, $(f_1 + f_2)$ は n 変数論理式である。
3. 上記で生成されるもののみが n 変数論理式である。

$\neg$ は論理の否定を、$\cdot$ は論理積を、$+$ は論理和を表す。

論理式に対応する論理関数は、以下のように定義される。

定義 2 (論理式の表す論理関数)

1. 入力論理変数 x_i の表す n 変数論理関数は、$f(x_1, x_2, ..., x_n) = x_i$ であるような論理関数である。

2. $(\neg f_1)(x_1, x_2, ..., x_n)$ は $f_1(x_1, x_2, ..., x_n)$ が 0 であるとき 1、$f_1(x_1, x_2, ..., x_n)$ が 1 であるとき 0 であるような関数である。
$(f_1 \cdot f_2)(x_1, x_2, ..., x_n)$ は $f_1(x_1, ..., x_n) = f_2(x_1, ..., x_n) = 1$ であるとき 1、それ以外のときには 0 であるような関数である。
$(f_1 + f_2)(x_1, x_2, ..., x_n)$ は $f_1(x_1, ..., x_n) = f_2(x_1, ..., x_n) = 0$ であるとき 0、それ以外のときには 1 であるような関数である。

2 変数論理式 $(((\neg x_1) \cdot x_2) + (x_1 \cdot (\neg x_2)))$ は上記の定義から、以下の論理関数を表す。

x_1	x_2	$(((\neg x_1) \cdot x_2) + (x_1 \cdot (\neg x_2)))$
0	0	0
0	1	1
1	0	1
1	1	0

演算の優先順位として $\neg > \cdot > +$ を仮定し、意味が曖昧にならない限り括弧は省略する。また $\cdot$ も可能であれば省略する。なお、以下では、論理式とそれが表す論理関数を同一視する。また、function_name = 論理式 ; という代入により、右辺の論理式を左辺で参照できるようにする。よって、

$f = (((\neg x_1) \cdot x_2) + (x_1 \cdot (\neg x_2)))$;

で定義された f と、

$h = (\neg x_1)$; $g = ((h \cdot x_2) + (x_1 \cdot (\neg x_2)))$;

で定義された g は同じ論理関数である。

2.2 共有二分決定グラフと否定枝

二分決定グラフは、論理関数を閉路を含まない有向グラフで表す方法である。これの基本となるのは、図 1 (a) に示す二分決定木である。二分決定木の各接点は変数でラベル付けされ、各接点から出る 2 本の枝は 0 と 1 でラベル付けされている。枝の向きは上から下とする。最上位の接点から下へ下がって行く時にどちらの枝を選んだかによって、各変数の値が決まる。深さ n の二分決定木の葉の数は 2^n 個あるので、左から関数の値 $f(0, ..., 0, 0)$, $f(0, ..., 0, 1)$, ..., $f(1, ..., 1, 1)$ の値を表すと、枝の選び方による変数への値の決め方と f の引数が一致し、論理関数 f を一意に表すことができる。

二分決定グラフは、この二分決定木の論理的に等価な接点を一つにまとめた図 1 (b) に示すようなものである。なお、ある接点から出る二つの枝が同じ接点を指している場合には、その接点無しでも情報は失われないので削除し、その接点に入る枝をその接点から出る枝に直接結合する。これを行なった結果を図 1 (c) に示す。ある論理関数を表す二分決定グラフは、変数の順序を決めれば一意である。なお、変数の順序により、二分決定グラフのサイズ (接点数) が大きく異なることが知られている。

共有二分決定グラフは、等価な接点を一つにまとめるという操作を複数の論理関数に対して行なったものであり ([8])、記憶量の低減などの利点がある。共有二分決定グラフでは、論理関数は一つの接点 (接点へのポインタ) に対応している。以下では、二分決定グラフで表された論理関数と、接点へのポインタを同一視する。

否定枝は、枝に否定演算を表す属性を付化させるもので、その枝を通った後は、論理値をすべて反転させる ([1], [7], [8])。これも、二分決定グラフの接点数を減らす上で非常に有効である。二分決定グラフの一意性を保つため、上記の文献に従い、(1) グラフの葉としては 0 だけを用い、(2) 各接点の 0 枝には否定枝を用いない、とする。否定枝を用いた二分決定グラフの例 $(((\neg x_1) \cdot x_2) + (x_1 \cdot (\neg x_2)))$ を図 2 に示す。

2.3 論理演算

二つの論理関数上の論理演算も、二分決定グラフを用いて行なうことができる。基本的には、以下のような再帰的関数を用いる。以

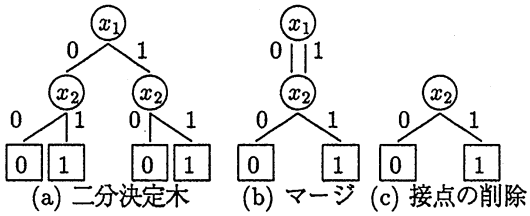

図 1: 二分決定グラフの例 ($f(x_1, x_2) = x_2$)

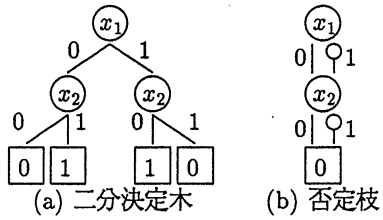

図 2: 否定枝の例 ($f(x_1, x_2) = (((\neg x_1) \cdot x_2) + (x_1 \cdot (\neg x_2)))$)

下、bdd_1, bdd_2 は二つの論理関数を表す二分決定グラフの接点へのポインタであるとする。なお、b を二分決定グラフの接点へのポインタとして、$b.label$ でその接点の変数ラベルを、$b.0$ でその接点の 0 枝の指す接点のポインタを、$b.1$ でその接点の 1 枝の指す接点のポインタを表す。すると、論理演算を行なう関数はつぎのように表せる。

```
function apply(bdd1, bdd2, op_code)
begin
  bdd1, bdd2 の論理演算を行い、結果が出れば終了。
  /* 演算結果の変数ラベル, 0 枝, 1 枝を求める */
  if (bdd1.label = bdd2.label) then begin
    label = bdd1.label;
    val0 = apply(bdd1.0, bdd2.0, op_code);
    val1 = apply(bdd1.1, bdd2.1, op_code);
  end
  else if (bdd1.label < bdd2.label) then begin
    label = bdd1.label;
    val0 = apply(bdd1.0, bdd2, op_code);
    val1 = apply(bdd1.1, bdd2, op_code);
  end
  else if (bdd1.label > bdd2.label) then begin
    label = bdd2.label;
    val0 = apply(bdd1, bdd2.0, op_code);
    val1 = apply(bdd1, bdd2.1, op_code);
  end
  現在までに生成された接点集合を探索し、
  (label, val0, val1) なる接点があるかどうかを判定。
  なければ新たに付加する。
end;
```

論理演算の部分は、例えば、AND 演算であれば、

```
function and(bdd1, bdd2)
begin
  if (bdd1 = 1 葉) then return(bdd2);
  if (bdd2 = 1 葉 ∨ bdd1 = bdd2) then return(bdd1);
  if (bdd1 = 0 葉 ∨ bdd2 = 0 葉 ∨ bdd1 = -bdd2)
    then return(ZERO);
  return(UNKNOWN);
end;
```

のようなベキ等律を用いた演算を行なう。

最後の (label, val0, val1) があるかどうかの判定の部分には、通常チェイニング付きの(同じキーを持つものをリンクで結合する) ハッシュ法が用いられる。以下にハッシュ表を探索する関数を示す。

```
function hash_search(label, 0 枝, 1 枝)
begin
  key = hash_func(label, 0 枝, 1 枝);
  ptr = hash_table[key];
  while (ptr ≠ NULL) do begin
    ptr の指す接点と (label, 0 枝, 1 枝) が
    等しければ ptr を返して終了。
    ptr = ptr → next_node;
  end;
  新しい接点の割り当てと、ハッシュ表への登録。
end;
```

3 並列構成アルゴリズムとその実現

以下では、共有記憶を有する並列計算機アーキテクチャに適した二分決定グラフ構成の並列アルゴリズムについて考察する。並列計算機中の各 CPU は、共有記憶にアクセスしながら全く独立に動作するものとする。概念図を図 3に示す。

実現においては、Sequent S-81 (CPU 80386 (16 MHz) × 28, 主記憶 86.75 MB) で、C 言語を用いた。並列プロセスの起動や共有記憶へのアクセスのロックはシステム標準のライブラリを用いた ([2])。

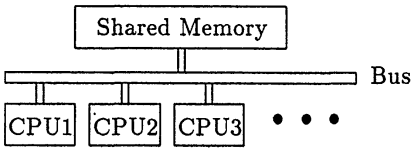

図 3: 共有記憶並列計算機アーキテクチャ

3.1 演算列の並列化

ここでは、代入を含む論理式表現で与えられた論理関数に対応する二分決定グラフを構成する問題を考える。論理関数は、入力論理変数と論理演算により、再帰的に定義されている。入力論理変数に対応する二分決定グラフは定数時間で求めることができるので、後は論理式中の論理演算に対応する演算を二分決定グラフ上で行なうことにより、論理式に対応する二分決定グラフを求めることができる。

例えば、$(((\neg x_1)\cdot x_2)+(x_1\cdot(\neg x_2)))$ に対応する二分決定グラフを求めるには、以下の処理を行なう必要がある。なお、演算の結果として求められるのは共有二分決定グラフの接点 (接点へのポインタ) である。

1. x_1, x_2 に対応する二分決定グラフの接点を求める。
2. $(\neg x_1)$ を行なう。
3. $(\neg x_2)$ を行なう。
4. $(\neg x_1)$ と x_2 の AND 演算を行なう。
5. x_1 と $(\neg x_2)$ の AND 演算を行なう。
6. 上記二つの AND の演算結果の OR 演算を行なう。

論理演算と二分決定グラフ上の演算を同一視すると、$(((\neg x_1)\cdot x_2)+(x_1\cdot(\neg x_2)))$ に対応する二分決定グラフを求めるときには、2 回の否定演算と、2 回の AND 演算、1 回の OR 演算を行なうことになる。また、これらの演算の間には依存関係があり、否定演算の後でなければ AND 演算は実行できず、また 2 つの AND 演算が両方とも終了した後でなければ OR 演算は実行できない。なお、2 つの否定演算はどちらを先に (あるいは同時に) 行なっても良いし、同様に 2 つの AND 演算はどちらを先に (あるいは同時に) 行なっても良い。

よって並列化の手法として、まず、実行するべき演算を依存関係で順番づけて実行できる順に並べて配列に入れておき、apply を実行できる CPU が順にそれを計算して行くという手法が考えられる。このとき問題となるのが共有メモリへの同時書き込みで、以下の二つの場合がある。

1. 実行する演算 (あるいは演算配列のインデックス) を求める場合。この場合には、インデックスの更新を伴う。
2. 新しい接点をハッシュ表へ登録する場合。ハッシュ表あるいはチェインで結合されている部分が更新される。

これらに関しては、ロック機構を用いて唯一つの CPU しか実行できないようにする必要がある。なお、ロックに関しては、

- ロックをかける領域を最小限にとどめる、
- 異なる変数へのアクセスには、異なるロック変数を用いる

の二点が重要である。

上記のアルゴリズムを実現するには、実行するべき論理演算を配列に設定し、以下のプロセスを並列実行させれば良い。

```
procedure do_apply()
begin
  while (TRUE) do begin
    lock(global_lock);
      op_id = op_count++;
    unlock(global_lock);
    op_id が実行すべき演算の数を越えたら終了。
    op_id 番目の演算子が計算されるのを待つ。
    op_id 番目の演算 を apply で実行する。
  end;
end;
```

上記の lock は共有変数へのアクセスのロックを行なう関数で、引数はロック用の変数で

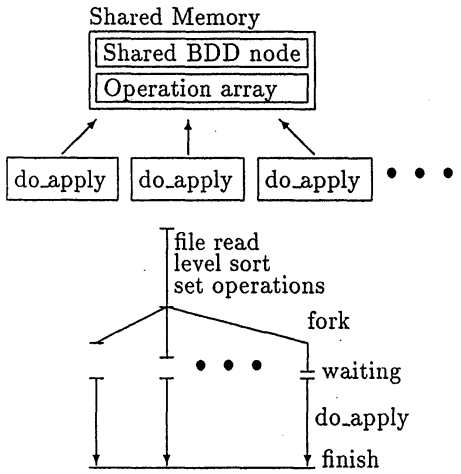

図 4: 並列実行方式

ある。do_apply の並列実行の概念図を図 4に示す。上の図は do_apply と共有記憶との関係を表し、下の図は並列実行の様子を表す。

演算子の計算終了を待つ部分は、計算しているCPU が書き込み、待つ方は読むだけであるので、ロックは不要である。apply の中で用いられる hash_search は以下のように変更しなければならない。

```
function parallel_hash_search(label, 0 枝, 1 枝)
begin
  key = hash_func(label, 0 枝, 1 枝);
  ptr = hash_table[key];
  old_top = hash_table[key];
  while (ptr ≠ NULL) do begin
    ptr の指す接点と (label, 0 枝, 1 枝) が
    等しければ ptr を返して終了。
    ptr = ptr → next_node;
  end;
  ダミーの新しい接点を割り当てる (new_node)。
  new_node へ label、0 枝、1 枝を代入。
  lock(hash_lock[key]);
    ptr = hash_table[key];
    while (ptr ≠ old_top) do begin
      ptr の指す接点と (label, 0 枝, 1 枝) が
      等しければ ptr を返して終了。
      ptr = ptr → next_node;
    end;
    new_node → next_node = hass_table[key];
    hass_table[key] = new_node;
  unlock(hash_lock[key]);
  接点の更新。
end;
```

ハッシュ表の接点の探索の最初の部分は読むだけであるので、ロックなしで行なう。つぎにロックをかけた後でもう一度あるかどうかの検査を行なう。これは、ロックを待っている間に新しい接点が登録される可能性があるためである。label, 0 枝, 1 枝 の新しく割り当てた接点への代入はロックの外で行ない、ロックの中では、新しい接点をハッシュ表に結合するという操作のみを行なう。これにより、ロックをかける領域を小さくできる。また、ロック変数として hash_lock という配列の key をインデックスとした変数を用いているので、異なる key へのアクセスが待たされることがなくなる。

これを実現したところ、8 bit の乗算器 (16 入力、16 出力、総接点数 80,458) で、以下のような結果を得た。

台数	1	5	10	15	20
時間 (秒)	18.35	5.633	4.799	4.799	4.899
効率 (倍)	1	3.26	3.82	3.82	3.75

なお、実行時間は、すべての CPU が起動されて do_apply を実行できる状態になった時点から、do_apply の実行が終了するまでを、主 CPU のクロックで計測したものである。

3.2 演算の分割

$(((\neg x_1) \cdot x_2) + (x_1 \cdot (\neg x_2)))$ に対応する二分決定グラフの構成において、論理演算のレベルを外部入力変数からの最大距離で定義すれば、

- $(\neg x_1)$ と $(\neg x_2)$ がレベル 1 、
- $(\neg x_1)$ と x_2 の AND 演算と x_1 と $(\neg x_2)$ の AND 演算がレベル 2 、
- 二つの AND 演算結果の OR 演算がレベル 3 となる。

全節の手法は、あるレベルに複数の演算がある場合には効果があるが、上記のレベル 3 のように一つしか演算がない場合には効果がなく、演算の数が 2 や 3 でも有効とはいえない。

この場合、一つの演算を複数の CPU で実行する手法が必要となる。ここではそれを、二分決定グラフ上のシャノン展開を用いて行なう手

法を提案する。基本的なアイデアは文献[6]に述べられているのと同じで、$f_1(x_1, x_2, x_3, ..., x_n)$ と $f_2(x_1, x_2, x_3, ..., x_n)$ の論理演算を、

1. $f_1(0, x_2, x_3, ..., x_n)$ と $f_2(0, x_2, x_3, ..., x_n)$ の論理演算、
2. $f_1(1, x_2, x_3, ..., x_n)$ と $f_2(1, x_2, x_3, ..., x_n)$ の論理演算、
3. 上記の演算結果から hash_serach により、$f_1(x_1, ..., x_n)$ と $f_2(x_1, ..., x_n)$ の論理演算結果を求める、

の3つの部分に分割することにより、1. と 2. の部分を並列に実行させようとするものである。これで2つの部分を並列に実行できる。同様に

1. $f_1(0, 0, x_3, ..., x_n)$ と $f_2(0, 0, x_3, ..., x_n)$ の論理演算、
2. $f_1(0, 1, x_3, ..., x_n)$ と $f_2(0, 1, x_3, ..., x_n)$ の論理演算、
3. $f_1(1, 0, x_3, ..., x_n)$ と $f_2(1, 0, x_3, ..., x_n)$ の論理演算、
4. $f_1(1, 1, x_3, ..., x_n)$ と $f_2(1, 1, x_3, ..., x_n)$ の論理演算、
5. 上記の演算結果から 3 回の hash_serach により、$f_1(x_1, ..., x_n)$ と $f_2(x_1, ..., x_n)$ の論理演算結果を求める、

とすると、4つの部分に分割されたことになる。展開された演算は、深さと論理値の割り当てパターンで識別できる。

上では、静的な展開を示したが、実際の二分決定グラフの演算では、f_1 と f_2 に対応するグラフの接点へのポインタが与えられ、それらが同じ変数でラベル付けされているとは限らないので、apply と同様の接点のラベルに応じた処理が必要となる。以下に展開の方法を示す。

```
function expand(bdd1, bdd2, op_code, depth, pattern)
begin
  b1 = bdd1;
  b2 = bdd2;
  for i = 1 to depth do begin
    b1, b2 の論理演算を行なう。
    結果が出ればそれを返す。
      /* 以下で b1, b2 を更新する */
    if (b1.label = b2.label) then begin
      if (pattern[i] = 0) then begin
        b1 = b1.0;      b2 = b2.0;
      end
      else begin
        b1 = b1.1;      b2 = b2.1;
      end
    end
    else if (b1.label < b2.label) then begin
      if (pattern[i] = 0) then b1 = b1.0;
      else b1 = b1.1;
    end
    else if (b1.label > b2.label) then begin
      if (pattern[i] = 0) then b2 = b2.0;
      else b2 = b2.1;
    end
  end;
  b1 と b2 の値を返す。
end;
```

分割された演算の結果から最終的な演算結果を求めるには、expand と同様の処理を行ない、分割時の変数ラベルを求めた上で hash_search を行なえば良い。

演算の分割数は、現在、各レベル中の演算の数で決めている。例えば、あるレベルの演算の数が2以下のときに各演算を二つに分割することにしたとする。このとき、あるレベルで op1, op2 の二つしか演算がなかったとすると、演算を入れる配列には、

```
op1  1      0
op1  1      1
op2  1      0
op2  1      1
op1  1  MERGE
op2  1  MERGE
```

のように設定され、do_apply で expand をしながら演算の実行を行なう。最初の要素は演算に関する情報を表し、二番目の要素は深さを、三番目の要素は展開のパターンを表す。なお、MERGE は分割された演算から最終結果を求めるための演算を表す。

レベル中の演算の数と、各演算の分割数の間にはトレードオフがある。現在は、以下の用無分割を用いている。

演算数	1	2, 3	4 ~ 10	11 ~ 25
分割数	128	32	16	8
演算数	26 ~ 35	36 <		
分割数	4	1		

このとき、8 bit の乗算器 (16 入力、16 出力、総接点数 80,458) で、以下のような結果を得た。

台数	1	5	10	15
時間 (秒)	18.35	4.05	2.15	1.47
効率 (倍)	1	4.53	8.58	12.52
台数	20	25		
時間 (秒)	1.23	1.2		
効率 (倍)	14.89	15.29		

CPU 1 台のときは、分割を行なわずに実行させている。また、9 bit の乗算器 (18 入力、18 出力、総接点数 194,986) で、以下のような結果を得た。

台数	1	5	10	15
時間 (秒)	82.42	17.72	9.32	6.67
効率 (倍)	1	4.65	8.85	12.36
台数	20	25		
時間 (秒)	5.32	4.47		
効率 (倍)	15.50	18.45		

9 bit の乗算器の場合の実行時間は、演算の分割数を増やして、以下のようにとすると、25 CPU で 4.083 秒 (20.19 倍) になる。

演算数	1	2, 3	4 ~ 10	11 ~ 25
分割数	128	64	32	16
演算数	26 ~ 35	36 <		
分割数	4	1		

4 おわりに

本稿では、二分決定グラフの構成法の共有記憶型並列計算機に適した並列化アルゴリズムを示し、さらに実験によってその有効性を示した。25CPU で、20 倍強の高速化が達成されている。

今後は ISCAS のベンチマークでテストを行なうとともに、大規模な二分決定グラフの構築に応用したいと考えている。

付録

実際の実行時間に関しては、入力ファイルの読み込み、論理演算のレベル付けと展開と演算の配列への代入に 8 ビットの乗算器で約 1.4 秒、9 ビットの乗算器で約 2.. 秒かかる。また、CPU の起動には、以下のように起動台数に応じた時間がかかっている。

CPU 台数	1	5	10	15	20	25
時間 (秒)	0	0.35	0.77	1.20	1.60	2.03

謝辞

カーネギーメロン大学において本研究の機会を与えていただくとともに、助言および議論していただいた Edmund M. Clarke 教授に深謝いたします。日頃から御討論いただく神戸大学太田有三助教授はじめ羽根田研究室の皆様に感謝いたします。御迷惑をおかけしている Sequent のルートの川村尚生様に感謝します。なお、本研究は一部文部省科学研究費奨励研究 (A)03750289 および實吉奨学金による。

参考文献

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. on Comput.*, Vol. C-27, No. 6, pp. 509–516, June 1978.

[2] Ed. Anita Osterhaug. *Guide to Parallel Programming, On Sequent Computer Systems, 2nd Ed.* Prentice Hall, 1989.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Proc. of 27-th DA Conf.*, pp. 40–45, June 1990.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Comput.*, Vol. C-35, No. 8, pp. 677–691, Aug. 1986.

[5] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. on Comput.*, Vol. C-40, No. 2, pp. 205–213, Feb. 1991.

[6] Shinji Kimura and Edmund M. Clarke. A parallel algorithm for constructing binary decision diagram. In *Proc. of ICCD'90*, pp. 220–223, Sept. 1990.

[7] J. C. Madre and J. P. Billion. Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proc. of 25-th DA Conf.*, pp. 205–210, June 1988.

[8] 湊真一, 石浦菜岐佐, 矢島脩三. 論理関数の共有二分決定グラフによる表現とその効率的処理手法. 情報処理学会論文誌, Vol. 32, No. 1, pp. 77–85, Jan. 1991.