1E2-4

BDD 簡約化アルゴリズムの並列化

Parallelizing BDD Reduction Algorithm

竹内 聖悟 *1　藤本 武彦 *2　安田 宜仁 *1　湊 真一 *1*2
Shogo Takeuchi　Takehiko Fujimoto　Norihito Yasuda　Shin-ichi Minato

*1JST ERATO 湊離散構造処理系プロジェクト
ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency

*2北海道大学 大学院 情報科学研究科
Graduate School of Information Science and Technology, Hokkaido University

A Binary Decision Diagram (BDD) and a Zero-suppressed Binary Decision Diagram (ZDD) are compressed data structures that represent Boolean functions and families of sets, respectively, and support various basic operations. To carry out those operations efficiently, BDDs and ZDDs need to be canonical, in other words, they need to be fully reduced. Knuth proposed a reduction algorithm for them. We need to apply the reduction algorithm after each operation, thus, it is important to accelerate the algorithm, especially when we utilize the large BDDs and ZDDs.

In this paper, we propose a new parallel reduction method that performs incomplete reduction, named "semi-reduction", in parallel for an input ZDD and concurrently performs a reduction algorithm, named "chaser", for the reduced ZDD. We conducted experiments with an OZDD that represents all simple paths in a graph with cliques and showed that the proposed method achieves four-fold speedup with 8 CPU cores.

1. はじめに

Binary Decision Diagram (BDD) [Bryant 86] は論理関数を表すデータ構造で，大規模集積回路の分野で開発された．そのバリアントとして，集合族を表すデータ構造 Zero-suppressed Binary Decision Diagram (ZDD) [Minato 93] がある．BDD と ZDD 上での論理関数や集合族に対する基本的な操作には union や intersection, difference などの多くの有用な演算があり，これらは BDD や ZDD を圧縮したまま計算することが可能である．データをコンパクトに保持でき，圧縮したまま演算が可能であるという利点から幅広い応用がされている．例えば，ZDD を用いるグラフ列挙索引化アルゴリズム [Knuth 09] が提案され，ネットワーク信頼性 [Hardy 07] やリンクパズルの全解列挙 [Yoshinaka 12] などに使われている他，頻出パタンの列挙を行うアルゴリズムについて，ZDD を用いたアルゴリズム ZDD-growth[Minato 07], LCM over ZDD[Minato 08] などが開発されている．

論理関数や集合族に対する演算を圧縮したままできるという利点を得るためには，BDD や ZDD が簡約化されている必要がある，すなわち，BDD や ZDD 中に冗長な節点が存在せず，唯一性が保証されている必要がある．

冗長な節点はその子孫節点の状態から冗長性を確認できるため，トップダウンに構築された BDD や ZDD において生じやすく，簡約化が必要である．また，演算を行うために簡約化が必要なため，繰り返し演算を行うような場合には演算毎に簡約化が必要である．簡約化のアルゴリズムは Knuth により提案されている [Knuth 09]．このアルゴリズムは入力となる BDD や ZDD のサイズに比例する時間がかかるため，大きな BDD, ZDD を扱う際にはこの簡約化アルゴリズムを高速に行うことは重要となる．

高速化の方法の 1 つにはハードウェアによる高速化があり，マルチコアやマルチ CPU の入手容易性から，並列化による高速化が有効である．特に，CPU 単体の速度向上は頭打ちとなっており，ハードウェアによる高速化を得るためには，並列化を行う必要があると言える．

本稿では，並列準簡約化と追駆簡約化による新しい並列化手法を提案する．並列に準簡約化を行い，その処理と並行してサイズが小さくなった BDD, ZDD に対して処理を行うことで高速化を得る手法であり，先行研究の並列化とは直交する手法となっている．本手法をクリークが並行しているグラフに対して実験し，8 スレッドで逐次に対しておよそ 4 倍の高速化を得た．

連絡先: 竹内聖悟，JST ERATO 湊離散構造処理系プロジェクト，〒 060-0814 北海道札幌市北区北 14 条西 9 丁目 北海道大学情報科学研究科 工学系 C306, takeuchi@erato.ist.hokudai.ac.jp

2. Preliminary

本稿で扱うデータ構造 ZDD について説明を行う．本稿で提案する手法は ZDD だけでなく BDD も扱えるが，実験で ZDD を扱うことと紙面の都合から BDD については省略する．

ZDD は集合族のグラフ表現である [Minato 93]．入次数が 1 の節点はちょうど一つだけ存在し，根と呼ばれる．内部節点 f は $\mathrm{V}(f), \mathrm{LO}(f), \mathrm{HI}(f)$ の三つのデータからなる．$\mathrm{V}(f)$ は節点 f のラベルとして台集合の要素を保持し，$\mathrm{LO}(f)$ と $\mathrm{HI}(f)$ は f の指す二節点へのポインタを保持する．それぞれ f の LO 節点，HI 節点と呼ばれる．LO 節点への有向枝は LO 枝と呼ばれ，点線矢印で表される．同様に，HI 節点への有向枝は HI 枝と呼ばれ，実線矢印で表される．これらは節点が表している集合の要素を使わないこと，使うことにそれぞれ対応する．二つの終端節点 $\bot$ と $\top$ は論理関数の偽と真に対応し，そこに至るパスに対応する集合が ZDD で表される集合族に含まれていないこと，含まれることに対応する．

図 1(b) に ZDD の簡約化規則を示す．以下の二条件を満たすものを *Reduced Ordered ZDD (ROZDD)* と呼ぶ．第一に，内部節点 u が v を指すとき，$\mathrm{V}(u) < \mathrm{V}(v)$ が満たされなけれ

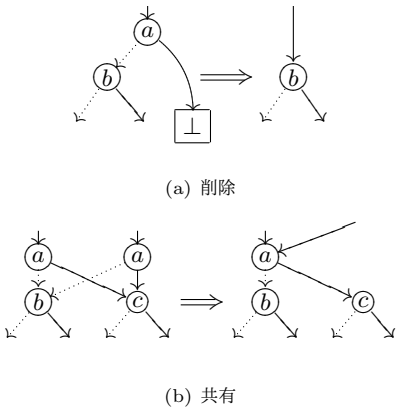

図 1: ZDD の簡約化ルール

ばならない．第二に，以下のいずれの簡約化規則も適用できないという意味で既約でなければならない．

1. 節点 u の HI 枝が $\perp$ を指すならば，u を指す全ての枝を u の LO 節点を指すように変え，u を削除する（図 1(a)）．
2. もし節点 u と v を根とする部分グラフ同士が等価ならば，部分グラフを共有する（図 1(b)）．

第一の条件が満たされ，第二の条件が満たされていない ZDD を OZDD と呼ぶ．以降，単に ZDD と表記しているものは ROZDD を指す．

第一の条件を満たす時は，根から末端までに現れる変数が全て同じ順序となっている．この時に，同一変数を扱う節点集合をグループとして扱い，以降これを**レベル**と呼ぶ．

BDD や ZDD はデータをコンパクトに保持でき，圧縮したまま演算が可能であるという利点から幅広い応用がされている．例えば，ZDD を用いるグラフ列挙索引化アルゴリズム [Knuth 09] が提案され，ネットワーク信頼性 [Hardy 07] やリンクパズルの全解列挙 [Yoshinaka 12] などに使われている他，頻出パタンの列挙を行うアルゴリズムについて，ZDD を用いたアルゴリズム ZDD-growth[Minato 07], LCM over ZDD[Minato 08] などが開発されている．

3. 関連研究

Knuth の逐次簡約化アルゴリズムと Iwashita による並列化のアルゴリズムを簡易に説明する．

3.1 逐次簡約化アルゴリズム

Knuth が提案した簡約化アルゴリズムは 2 段階の手法で，末端から順に各変数に対する ZDD 節点を処理していく手法である [Knuth 09]．OZDD の各レベルに対して，処理するレベルより下は簡約化が済んでいる前提で，内部節点それぞれについて削除と共有ルールの適用を行う．1 段階目では，各節点に対して，削除ルールが適用できる節点は削除し，そうでない節点については共有される節点候補のリストを作成する．同じ子節点を持つものは同じ節点へと簡約化されるため，片方の子節点をハッシュ値として利用することで，節点候補リストが作成される．効率良くメモリを使用するため，上記の操作中に子節点への枝を同じ子節点を持つ節点へのリンクとして使う．2 段階目では，そのようにして作られた共有される節点候補リストを辿り，同一の節点があれば共有ルールを適用する．

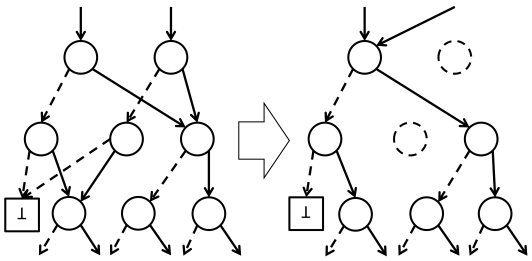

図 2: 末端から順に簡約化を行う必要のある ZDD: (左図は簡約化前, 右図は簡約化後)

これらの処理を OZDD の末端から根へと順に行うことで，入力 OZDD と同じアイテム集合を表現し，かつ唯一性のある，ROZDD が出力される．

唯一性の保証のためには，末端から簡約化規則を適用する必要がある．図 2 左の OZDD に対して末端から簡約化規則を適用すると，図右のような ZDD となる．一方，図左の上から 1 段目に対して簡約化規則を適用した場合，2 つの ZDD 節点は LO 枝が異なる ZDD 節点を指しており，共有されることはない．この 2 つの ZDD 節点が同一だと知るには子孫節点が既約となっている必要があり，簡約化により唯一性を保証するためには末端から処理を行わなければならないことが分かる．

3.2 Iwashita による並列化

簡約化アルゴリズムの並列化の先行研究として Iwashita の手法がある [Iwashita 14]．逐次アルゴリズム中の節点処理を並列に行う手法で，末端から上へと順に処理が行なわれる．

節点処理を並列に行う際，本来共有される節点を異なるスレッドで処理することがあり，片方の処理が終わるのを他方が待つ必要がある．このコストを避けるために，各節点をハッシュ関数により分類し，タスクのためのテーブルを使い，同じ節点は必ず同じスレッドが扱うような処理を行っている．これにより各スレッドの処理が独立に行うことができ，効率良く並列化が行なわれる．Iwashita は ZDD 構築の並列化を行っており，簡約化部分を単体で扱った実験結果は文献に掲載されていない．この手法では各レベルで並列化しているため，各レベルのノード数が少なく，レベルの多いような OZDD では効果が低くなると考えられる．

4. 並列準簡約化と追駆簡約化による並列化

本稿では，並列に準簡約化を行いつつ追駆して従来の簡約化を行う新しい並列化の手法を提案する．**準簡約化**は必ずしも完全ではない簡約化だが，個々の処理を独立に行うことが容易なため並列処理により効率良く処理できる特徴がある．**追駆簡約化**は，不完全な簡約化の結果を正しく簡約化するために行なうもので，並列準簡約化により追駆簡約化に対する入力 OZDD サイズが小さくなるため，高速な処理が可能となる．

まず，準簡約化について説明する．準簡約化では，入力 OZDD をボトムアップに順番に処理するのではなく，並列に処理を行う．子孫節点での簡約化の結果を利用しないため，冗長な ZDD 節点は残り得る．この処理は非同期で行うため，十分なレベルがあれば並列化効率がスレッド数倍になることが期待される．また，複数のレベルを一つの単位として処理することができる，これをチャンクと呼ぶ．複数のレベルをボトムアップに処理することで，その範囲では簡約化の結果を利用できるた

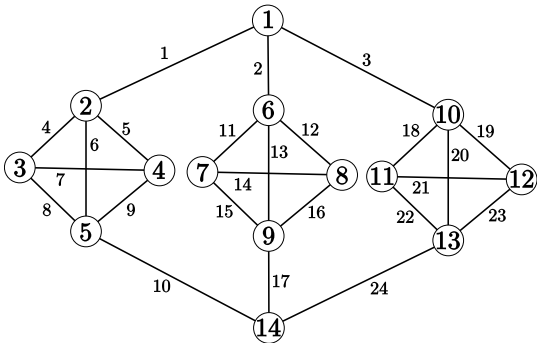

図 3: クリーク並行グラフの例，4 ノードのクリークが 3 並列になっている

め，共有される ZDD ノード数を増やすことが期待される．

追駆簡約化は，準簡約化処理と並行し，準簡約化の出力 OZDD を 1 スレッドで簡約化する．追駆簡約化で処理するレベルが，準簡約化で処理途中の場合は待つ必要があるが，準簡約化が並列化により十分高速に行なわれれば，待ち時間はほとんど生じない．

提案手法による高速化の上限について述べる．並列化により準簡約化がどれだけ速くなったとしても，追駆簡約化ではその出力 OZDD を簡約化するため，出力 OZDD サイズに応じた時間が必要となる．このことから二通りの上限が考えられる．一つは，準簡約化の出力 OZDD に対する追駆簡約化を，準簡約化と並行して行うのではなく，準簡約化の処理後に行った場合の時間から高速化の上限が得られる．もう一つは，実行時間からではなく OZDD サイズからの見積で，準簡約化の入力 OZDD と出力 OZDD のサイズ比から得られる上限である．例えば準簡約化が全く OZDD サイズを減らす事ができない場合は，提案手法による並列化では高速化を得られないことがわかる．

提案手法と Iwashita の手法とでは並列に処理する部分が独立であり，組合せによりさらなる並列化の効果が期待される．

5. 実験

提案手法の有効性を示すために，逐次アルゴリズムとの比較実験を行う．実験には，ZDD をトップダウンに構築するグラフ列挙索引化アルゴリズムによって生成される OZDD を用いる．トップダウンに構築を行うと，下のレベルの既約が保証されておらず，冗長な ZDD 節点が含まれる．そのために簡約化が必要であり，従来は Knuth による逐次の簡約化アルゴリズムを適用し，簡約化を行っている．

実験では，あるグラフ上の 2 点間の全単純経路を表す OZDD を生成し，それに対する簡約化において手法の比較を行う．扱うグラフは 2 つの端点を持ち，その 2 点の間にクリークを並行して持つようなグラフで，以降このグラフを クリーク並行グラフと呼ぶ．このグラフでは，各クリークの処理は独立である特徴があり，OZDD についても各クリークに相当するレベルをまとめて扱うことが有効であると考えられる．4 ノードのクリークを 3 つ並列に並べた，クリーク並行グラフの例を図 3 に示す．この例では，14 ノードと 24 辺からなる．

実験には，10 ノードからなるクリークが 100 個並ぶクリーク並行グラフにおいて，頂点 1 から頂点 1002 までの単純経路を列挙した際に得られる OZDD を利用した．この OZDD のレベル数は 4,700 となる．並行するクリークの一つを考えると，クリーク内部の 45 辺と 2 端点への辺をあわせた 47 辺があるが，始点は最初に処理されるので 46 辺が一つの単位となる．このように規則的なパターンがあるため，チャンクサイズを適切に設定することで，準簡約化の効率が上がることが期待できる．

表 1: クリーク並行グラフ上のパスを表す OZDD 節点の簡約化される節点数

チャンクサイズ	準簡約化 冗長節点数	準簡約化 等価節点数	追駆簡約化 入力節点数
1	58,851,550	2,200,550	52,678,650
2	59,435,250	2,387,800	51,907,700
4	59,711,850	2,421,900	51,597,000
8	60,143,625	2,464,100	51,123,025
16	63,097,145	2,525,686	48,107,919
32	79,568,395	1,792,418	32,369,937
64	96,629,851	1,006,036	16,094,863
128	104,683,472	644,061	8,403,217
256	109,064,403	467,930	4,198,417
512	111,499,255	352,572	1,878,923
1024	111,946,441	341,297	1,443,012
46	112,913,750	253,400	563,600

実験には，Intel Xeon CPU E7-2830 2.13GHz 8core を 8CPU，計 64 コアのマシンを利用した．Non-Uniform Memory Access (NUMA) 環境であるため，性能を引き出すためにはメモリアクセスやコアの配置に注意する必要がある．予備実験から，メモリはアクセスするコアの近くに配置する設定の性能が良く，以下ではその設定で実験を行った．

5.1 実験結果

OZDD 簡約化の実験結果を示す．まず，逐次手法による簡約化の計算時間は 1.050 秒であった．この時，簡約化前の ZDD 節点数は 113,730,752 で，得られた既約な ZDD の節点数は 563,600 となり，簡約化によりおよそ 200 分の 1 の節点数となった．以降，並列化の実験結果ではこの結果との比較によりどれだけ高速化が行なわれたかを評価する．

予備実験として，準簡約化におけるチャンクサイズの影響評価を行うため，チャンクサイズを変えながら準簡約化による節点削減の評価を行った．準簡約化の出力 OZDD 節点数や，準簡約化の内，冗長節点と等価節点数の内訳を表 1 に示す．チャンクサイズを大きくし，先読みが深くなるほどに節点数を大きく削減できていることがわかる．また，問題に対する知識からチャンクサイズを 46 に設定したところ，チャンクサイズ 32 や 64 を大きく上回る性能を得た．このチャンクサイズで準簡約化を行った結果は節点数から既約な ZDD と一致している．

次に性能評価を行うため，チャンクサイズを 46 に固定しスレッド数を変えながら実験を行った．準簡約化と提案手法全体の計算時間，さらに全体の速度が逐次に対して何倍となったかを，代表的なスレッド数について表 2 に示す．また，図 4 には準簡約化及び提案手法の高速化の結果を全てプロットした．X 軸はスレッド数を，Y 軸は何倍の速度となったかを表す．図中の $*$ と点線で表される線は準簡約化を，$+$ と実線で表される線は全体での速度比を表し，準簡約化では 1 スレッドの結果に対する高速化，全体では逐次の手法に対する高速化を示す．

表 2: クリーク並行グラフ上のパスを表す OZDD での，準簡約化と提案手法の計算時間

スレッド数	準簡約化（秒）	提案手法（秒）	高速化
1	1.400	1.667	0.630
2	0.722	1.450	0.724
4	0.384	0.542	1.937
8	0.212	0.269	3.903
16	0.138	0.281	3.737

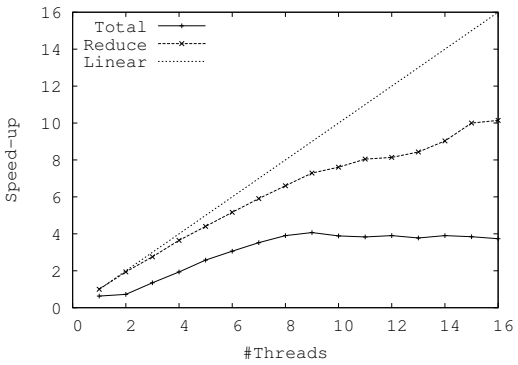

図 4: クリーク並行グラフ上のパスを表す OZDD での，提案手法の高速化と並列準簡約化の高速化

提案手法は最大でおよそ 4 倍の高速化が得られている．

5.2 考察

図 4 から，並列準簡約化の高速化はスレッド数に対して逓減していないが，全体の性能は逐次に対しておよそ 4 倍で頭打ちとなっている．これは提案手法の節で述べた並列化による高速化の上限から説明ができる．上限の 1 つは追駆簡約化に対する入力 OZDD サイズから得られるもので，この場合は表 1 の ZDD 節点数から，上限はおよそ 200 倍となる．実際の処理時間から見る上限では，表 2 の 1 コアでの実験結果から，準簡約化後の OZDD に対する追駆簡約化はおよそ 0.267 秒かかっており，逐次に対しておよそ 4 倍の高速化が提案手法の上限となる．この上限は 8 スレッドでおおよそ達成されており，高速化の上限に達したことからこれ以降の性能が頭打ちになっていると考えられる．

この 4 倍という上限は，OZDD サイズから得られた 200 倍という上限よりもかなり小さい．今回の実験では冗長な節点の削除が多いが，節点 1 つを削除する処理にかかる時間は非常に短い．処理時間が微小な場合，並列化の効果と並列処理のためのオーバーヘッドとが拮抗し，並列化の効果が得られないことがある．このことが，今回並列化性能の上限が小さくなった原因の一つであると考えられる．

6. 終わりに

本稿では，BDD, ZDD 簡約化アルゴリズムについて，並列準簡約化と追駆簡約化による新しい並列アルゴリズムを提案した．クリーク並行グラフ上のパス列挙から得られた OZDD での実験結果から，提案手法は 8 スレッドで 4 倍程度の高速化を得ており，提案手法の有効性を示すことができた．

今後の課題としては，岩下が提案した並列化手法との比較や組合せが考えられる．提案手法は，岩下の手法と直交する手法であると考えられ，これを実験で確かめることは重要である．さらに，提案手法や Iwashita の手法，両者を組み合わせた手法について，どのような問題において有効であるかを明らかにする必要がある．例えば，提案手法は今回のクリーク並行グラフ上のパス列挙のように，繰り返し構造を持つような問題に対て有効であると考えられる．

提案手法は，データの処理順序に依存関係があって並列化が難しい手法に対する並列化の枠組みと見ることが出来る．すなわち，不完全でも並列化が容易な処理を導入し，その並列処理により高速に入力データを小さくし，小さくなったデータに対して本来の処理を並行して行うことで，効率的な処理を行う枠組みである．この枠組を他の課題へと適用することは，今後の課題の一つである．

参考文献

[Bryant 86] Bryant, R.: Graph-Based algorithms for boolean function manipulation, *IEEE Transactions on Computers*, Vol. 35, pp. 677–691 (1986)

[Hardy 07] Hardy, G., Lucet, C., and Limnios, N.: K-Terminal Network Reliability Measures With Binary Decision Diagrams, *IEEE Transactions on Reliability*, Vol. 56, No. 3, pp. 506–515 (2007)

[Iwashita 14] Iwashita, H.: *Practical Techniques and Applications of Binary Decision Diagrams in Property Verification Problems*, Ph.d thesis, Hokkaido Univeristy (2014)

[Knuth 09] Knuth, D. E.: *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley Professional, 12th edition (2009)

[Minato 93] Minato, S.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems, in *30th ACM/IEEE Design Autiomation Conference (DAC-93)*, pp. 272–277 (1993)

[Minato 07] Minato, S. and Arimura, H.: Frequent Pattern Mining and Knowledge Indexing Based on Zero-Suppressed BDDs, in *Knowledge Discovery in Inductive Databases*, pp. 152–169 (2007)

[Minato 08] Minato, S., Uno, T., and Arimura, H.: LCM over ZBDDs: Fast Generation of Very Large-Scale Frequent Itemsets Using a Compact Graph-Based Representation., in *PAKDD*, pp. 234–246 (2008)

[Yoshinaka 12] Yoshinaka, R., Saitoh, T., Kawahara, J., Tsuruma, K., Iwashita, H., and Minato, S.: Finding All Solutions and Instances of Numberlink and Slitherlink by ZDDs, *Algorithms*, Vol. 5, No. 2, pp. 176–213 (2012)