RA-008

portfolio型並列SATソルバにおける多様な探索を行うための手法の提案

Diversification of Search Space for Parallel SAT Solvers

薗部 知大* 稲葉 真理 *

1 まえがき

充足可能性問題 (SAT 問題) は古典的な NP 完全問題 [1] として知られており、多項式時間内での解決が難しいと信じられている。しかし近年の研究成果により、この問題を解く SAT ソルバの性能が飛躍的に向上し、回路検証や計画生成等の多くの実アプリケーションへの応用がなされている。

巨大な探索空間を対象とする SAT ソルバにとって、探索の並列化は重要な手法である。従来の並列化手法は探索空間を分割して各ワーカーに割り当てる分割統治法に基づいていたが、近年の並列化手法の主流は portfolio [2] であり、探索を行う各ワーカーが全探索空間を対象に競争的かつ独立に探索を行う。従来の分割統治法では、探索空間の分割に使用する変数の選択方法によって、ワーカーに割り当てられる探索空間の大きさが非常に異なることがしばしばあり、ワーカー間のロードバランシングが困難であるという問題が存在した。これに対して portfolio では、探索空間の分割を行わず、全ワーカーが全空間を対象とした独立な探索を行うため、ロードバランシングの必要性が無い。しかし、全ワーカーが同一の探索を行っては並列効果が得られないので、各ワーカーの探索挙動を変更するための diversification (探索の多様化) [3] が必要となる。これによって、互いに独立な探索を行いつつ、可能な限り異なる探索を行うことで、全体として多様な空間に対する探索を行うことが可能となる。また、一つのワーカーが解を発見した時点でその解が全体の解として採用できるため、全ワーカーは競争的に解の発見につとめるだけで良く、同期を取るような通信処理の必要がほとんどない。

従来の diversification は、ワーカーの探索挙動を異なるものにするために、変数選択手法、restart の頻度等の探索に関わる各種パラメータの変更を行うことで実現してきた。この種類の diversification を、本研究ではワーカーの探索挙動を異ならせるための diversification として、search activity diversification と名付ける。しかし、この種の diversification では、変更する探索パラメータの組み合わせが限られてしまい、スーパーコンピュータのようなよりワーカー数の多い大規模な並列環境では、互いの探索が重複する可能性が高まる恐れがある。そのような場合、portfolio における並列効果が期待できない。

この問題に対して、本研究では各ワーカーの探索挙動を変更させるのではなく、各ワーカーの探索空間を可能な限り変更させることを考える。そのためには、各ワーカーに優先的に探索を行う複数の変数を重複無く割り当て、各ワーカーは割り当てられた変数に対する重点的な探索を行うことで、全体としてより重複の少ない多様な探索を実現することが可能となる。この diversification を、各ワーカーの探索空間を異ならせることを主眼とした search space diversification と名付ける。本論文ではこれを実現するために、実アプリケーションから派生する SAT 問題に備わる特定の変数間に存在する依存関係 (問題の構造とよばれる) を利用して、互いに関連の深い変数を与えられた SAT 問題から簡易的に解析して一つのブロックとして構築し、それらを各ワーカーに割り当てる block branching を提案する。各ワーカーは割り当てられた複数の変数を、定期的に優先的に値を割り当てる探索を行う。計算機実験の結果、多くの問題においてワーカー間で学習される clause の重複が少なくなり、性能の向上に成功した。

本論文の構成は以下の通りである。2 節では SAT 問題と SAT ソルバの概要を説明し、3 節では提案手法の block branching について詳述する。4 節では実験結果を考察し、5 節で関連研究を紹介する。6 節でまとめを述べる。

*東京大学大学院情報理工学系研究科

2 SAT 問題と SAT ソルバ

本節では SAT 問題と SAT ソルバの概要を説明する。

2.1 SAT 問題

充足可能性問題 (Satisfiability problem, SAT 問題) とは、与えられた論理式を真にするための変数の真偽値割り当てが少なくとも一つ存在するか判定する問題である。変数は真 (true) か偽 (false) のどちらかを値として取る Boolean 変数である。問題を表す論理式の形式は乗法標準形 (Conjunctive Normal Form, CNF) が一般的である。

- 乗法標準形 (CNF):
  clause(節) が ∧(and) で連結された論理式
- clause(節、クローズ):
  literal が ∨(or) で連結された論理式
- literal: 変数の正負の出現を表す

CNF の評価を真にするためには、すべての clause の評価を真にする必要があり、各 clause に必ず真となる literal が一つ存在しなければならない。そのような変数割り当てが存在する問題を充足可能 (Satisfiable) な問題と呼び、逆にそのような変数割り当てが存在しない問題を充足不可能 (Unsatisfiable) な問題という。

以下に CNF の例を示す。

$$P1 = (a) \wedge (\neg a \vee b) \wedge (\neg b \vee c \vee d)$$
$$P2 = (a \vee \neg b) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee b) \wedge (a \vee b)$$

P1 は三つの clause からなる CNF であり、例えば四つの変数全てに真を割り当てることで全ての clause を真にすることが可能であるため、充足可能な問題である。反対に、P2 は二つの変数にどのような値を割り当てても四つの clause を同時に真にすることは不可能であるため、解の存在しない充足不可能な問題である。clause に含まれる literal が k 個に統一された SAT 問題は k-SAT 問題と呼ばれ、2-SAT 問題に対する多項式時間アルゴリズムは存在するが、k が 3 以上の SAT 問題は NP 完全問題として知られている。

SAT 問題はその生成方法から大きく二種類に分類される。一つは現実世界のアプリケーション [4] から生成される問題で、一般的に変数間の依存関係が強く、構造的 (structured) な問題である。二つ目はアプリケーションの問題とは対照的に、乱数を用いて生成するランダムな問題である。ランダムな問題は一般的にアプリケーションの問題のような構造は備わっておらず、効果的な解の探索アルゴリズムも大きく異なる。本研究の対象は前者のアプリケーションから派生する構造の備わった問題である。

2.2 SAT ソルバ

SAT ソルバは入力として与えられた CNF に対して充足可能性判定を行うプログラムである。また、充足可能な問題に対してその充足解も示す。近年の研究成果によって、SAT ソルバの扱える問題の規模が向上したため、様々な実世界のアプリケーションに応用されている。例として、回路検証 [5]、計画生成 [6] 等が挙げられ、より大規模な問題を高速に解くことが期待されている。

SAT ソルバの探索アルゴリズムは二つに大別される。一つは確率的アルゴリズムであり、ランダム性を利用して探索を行う。まず全ての変数に対して真偽の値をランダムに割り当て、ランダムウォークや適当な変数の値を反転させ、充足される clause 数を増やす確率的局所探索 (Stochastic Local Search, SLS) を用いて解の探索を行う。このアルゴリズムはランダムな問題に対する性能が高く、また解の総数が多い問題に対して高速な探索を行うことが可能である。しかし、確率的アルゴリズムは局所的な探索しか行わないため、充足不可能な問題を解くことができないという特徴がある。

それに対して、体系的 (complete) アルゴリズムは全ての探索空間を対象とした探索を行うため、充足可能、充足不可能どちらの種類の問題に対しても解の存在性を証明できる。本研究ではこの体系的アルゴリズムに基づく SAT ソルバを対象とする。代表的なアルゴリズムとして、Davis-Putnam-Logemann-Loveland (DPLL) アルゴリズム [7] が挙げられる。DPLL アルゴリズムは Boolean 変数をノードとする二分木の backtrack 型の探索を行う。大きく三つの機能から構成され、特定の heuristic に基づいて変数を選択して値を割り当てる decision、decision の対象となった変数に関連する変数に対する値を強制する propagation (unit propagation)、矛盾 (conflict) が発生した際に直前の decision をやり直す backtrack が該当する。近年では、矛盾が生じた際にそのときの変数割り当てを解析して新たな clause を学習する clause 学習や、局所解への陥りを防止するために全ての変数割り当てを取り消して探索を最初からやり直す restart 等が DPLL に加わり、飛躍的な性能向上を遂げた。近年の SAT ソルバは Conflict Driven Clause Learning (CDCL) ソルバとも呼ばれている。

decision に関して、VSIDS(Variable State Independent Decaying Sum) が近年の多くの SAT ソルバで使用されている。VSIDS は、各変数にスコアを付与し、decision では

まだ値の入っていない変数の中で一番スコアの高い変数が選ばれる。clause が学習されると、その clause に含まれる変数のスコアが上昇し、定期的に全スコアが定率で減少する。この仕組みにより、現在探索されている空間に関わる変数のスコアが上昇していき、着目している空間への探索を集中的に行うことができる。

巨大な空間を対象とする SAT ソルバにとって、探索の並列化も重要な手法である。従来の並列 SAT ソルバは探索空間の分割統治に基づいていたが、近年の主流は portfolio [2] である。分割統治法では、探索空間の分割に使用する変数によって各ワーカーに割り当てられる空間の大きさが非常に異なることが多々あり、ワーカー間のロードバランシングが困難という問題が存在した。特に、解の存在しない充足不可能な問題に対する探索では、全ワーカーがその不可能性を示す必要があり、解の判定には全ワーカーの探索の終了が必要であった。それに対して portfolio では探索空間の分割を行わず各ワーカーが与えられた問題を独立に探索するため、ロードバランシングの必要がない。しかし、全ワーカーが同一の探索をしては並列効果が得られないので、各ワーカーの探索 (空間) を変更するための diversification (探索の多様化) [3] が必要となる。これによって、互いに独立な探索を行いつつ、可能な限り異なる探索を行うことで、全体として多様な空間に対する探索を行うことが可能となる。また、一つのワーカーが解を発見した時点でその解が全体の解として採用できるため、全ワーカーは競争的に解の発見につとめるだけで良く、同期を取るような通信処理の必要がほとんどない。

本研究では diversification の方式を二つに大別する。一つ目は各ワーカーの探索挙動を変更する search activity diversification であり、各ワーカーの主要な探索パラメータ (decision heuristic、restart の頻度等) をそれぞれ異なるものにすることで、各ワーカーの探索を可能な限り異なるものにすることが主眼の手法である。もう一つは、各ワーカーの探索空間を変更する search space diversification であり、各ワーカーに複数の変数を重複無く割り当て、各ワーカーがそれらに対する優先的な探索を行うことで、可能な限り互いの探索空間を異なるものにすることが主眼の手法である。本研究では、後者の search space diversification に属する手法の提案を行う。

3 block branching の提案

これまでの portfolio に基づく並列 SAT ソルバ (例えば [8]) では、主に共有メモリ環境を想定した、比較的小規模な並列環境 (8 から 16 ワーカー規模) において search activity diversification を主に行ってきた。十数程度のワーカー数であれば、decision heuristic や restart の頻度を変更するだけでも、ワーカー間で異なる探索を行わせることが可能である。しかし、これからよりプロセッサの集積度が増して容易に大規模な並列環境が手に入ると予測できる時代を迎えるにあたり、例えば数千以上のワーカー数から成るより大規模な並列環境では、portfolio における探索の多様性を search activity diversification だけでは確保できないと考えられる。その理由として、各ワーカーの探索パラメータの組み合わせは限られており、それらの変更だけでは多くのワーカー間で探索空間の重複が生じる可能性が高い。これを防止するためには、各ワーカーに複数の変数を重複無く割り当て、各ワーカーがそれらに対する優先的な探索を行う search space diversification を施すことで、互いに独立な探索を行いつつ全体として多様な探索空間への探索を行うことが可能となる。

search space diversification を行うにあたり、各ワーカーに割り当てる変数が重要な要素となる。[9] では、各ワーカーにランダムに三個の変数を割り当て、各ワーカーは常にそれらを探索木の根の部分で値を割り当て探索を行う。すなわち、各ワーカーを特定の探索空間に束縛することで diversification を図っている。しかし、割り当てる変数をランダムに選択するということは、互いに関連の低い変数どうしが選択される恐れがあり、それらを探索木の根で先に値を割り当てても特定の探索空間へ必ずしも導くことができないと考えられる。また、それらを常に探索木の根に固定してしまうと、本来それらの変数より先に値を割り振るべき変数が存在する場合、効率的な探索を阻害してしまう可能性もある。

本研究では、与えられた CNF を簡易的に解析して、問題の構造を成す互いに関連の深い変数を一つのブロックにまとめそれらを各ワーカーに割り当て、各ワーカーは割り当てられた変数に対する優先的な探索を行う block branching を提案する。アプリケーションの問題には特定の変数間に依存関係を示す構造が備わっており、例えば代表的な decision heuristic の VSIDS は矛盾から判明する変数間の関係性に着目した decision を行うことで効率的な探索を実現しているため、ランダムに変数をまとめる手法より効果的に特定の探索空間へ導くことが期待できる。しかし、それらの関係は実際に探索を行うことで解析できるものであるため、探索を開始する前に判定を行うことは困難である。そこで、与えられた CNF の中に含まれる、二つの literal から構成される binary clause に着目を行う。同一 binary clause 内に含まれる二つの変数は、片方の値が決定するともう片方の値も (unit propgation によって) 強制されるため、互いに依存関係の強い変数どうしといえ

る。アプリケーションの問題には一般的に多くの binary clause が CNF に含まれているため、それらを探索開始前に高速かつ簡易的に解析し、関連の深い変数どうしを一つのブロックとしてまとめることができる。それらを各ワーカーに割り当て、各ワーカーは割り当てられた変数を探索木の根に固定はせずに、定期的にそれらに対して優先的に decision するようにさせることで、特定の探索空間を優先的に探索しつつ柔軟な探索を行うことを目指す。それを実現するためには、割り当てられた変数の VSIDS スコアを定期的に上昇させることで、clause 学習と調和を図る VSIDS の枠組みにそのまま組み込むことが可能となる。ただし、学習によって得られるスコアに干渉しすぎないような実装の工夫が必要であり、次節でその詳細を述べる。

3.1 block branching の詳細と疑似コード

変数の分割方法は、CNF 中の binary clause に着目し、それらの含意関係を解析する。$(a \vee b)$ という clause なら、$(\neg a \Rightarrow b)$ かつ $(\neg b \Rightarrow a)$ と含意を使って表せる。前者は、literal の b は a に False が代入された瞬間に、b には True が (propagation によって) 代入されるため、literal の b は literal の $\neg a$ に支配されていると言える。このような支配関係を全ての binary clause を解析することによって構築し、変数 (literal) の集団をブロック単位にまとめることができる。解析には union-find アルゴリズムを用いる。

併合の方法に関して、まず初期状態は自分を支配する literal(親と呼ぶ) を自分自身とする。任意の binary clause の $(a \vee b)$ に対して、
$\mathrm{parent}(b) := \mathrm{parent}(\neg a)$
$\mathrm{parent}(a) := \mathrm{parent}(\neg b)$
という関係を構築する。ここで parent(x) は literal の x の親を示し、x の属するブロックの代表となる変数である。すなわち、対象ブロックの親が別のブロックの親に併合される。この代入式を全ての binary clause に対して行うことで、共通の親を持つ literal の群が構築される。これを一つのブロックとする。

binary clause を解析して、関連の深い literal からなるブロックの構築の例を図 1 を用いて説明する。まず State #1 の状態を考える。変数は a, b, c, d の四種類があり、それぞれに対応する literal が八個ある。この図は初期状態を表しており、自身の親は自分自身であることを矢印で示している。State #2 において最初の binary clause の $(a \vee b)$ の解析を行う。前述の通り、literal の b は、literal の $\neg a$ によって支配されているため、literal の b の親は $\neg a$ となる。literal の a に関しても同様の処理が行われる。その関係が有向エッジで示されている。次に、State #3 において $(c \vee d)$ が評価され、literal の d の親は $\neg c$ となる (literal の c にも同様の処理が行われる)。この時点でブロックは四個である。最後に State #4 に $(a \vee d)$ が評価された後の状態を示す。まず、literal の d が literal の $\neg a$ に支配されている関係に関して、すでに d の親は $\neg c$ であるため、$\neg c$ の親が $\neg a$ になり、ブロックの併合が行われる。$\neg a$ と $\neg c$ の間には論理的関係性はないものの、本手法では、可能な限り簡易的にブロックの生成を行うことを目的としているため、このような関連の無い literal 同士の併合も許容する。また、解析を行う binary clause の順番によって構築されるブロックの内容や状態も変化するが、これも許容する。同様に a は $\neg d$ に支配される関係に関しても、a の親である $\neg b$ が $\neg d$ を親として、二つのブロックが一つのブロックになる。以上のように、三つの binary clause を解析することで、八個の literal は二つのブロックにまとめられる。

block branching を構成する各関数の疑似コードを以下に示す。

```
// 各 literal の状態の初期化を行う変数
init() {
  for each lit
    lit.parent = lit; // 親は自分自身に
    lit.size = 1; // block のサイズ
}

// 対象の literal の親を求める関数
find(x) {
  if x.parent != x
    x.parent = find(x.parent)
  return x.parent
}

// 二つのブロックを併合する関数
union(x, y) {
  xp = find(x);
  yp = find(y);
  // もし併合後のブロックのサイズが閾値を超えている場合は併合を取り消す
  if(xp.size + yp.size > THRESHOLD)
    return
  yp.parent = xp;
  xp.size += yp.size;
}

// ブロックを構築する関数本体
```

```
makeBlocks() {
  init();
  for each binClause in clauses
    union(!binClause[0], binClause[1]);
    union(!binClause[1], binClause[0]);
  [sort blocks in descending order of each size]
  [select bigger blocks]
}
```

makeBlocks 関数が処理の本体であり、init 関数でまず全ての literal の親を自分自身で初期化し、ブロックの大きさを 1 とする。その後 CNF 内の全ての binary clause に対して、二項関係を union-find アルゴリズムによって併合する。union 関数内の “THRESHOLD” はブロックの併合に関する制限値であり、この制限値を超える場合は併合が取り消される。この処理によって最終的に構築されるブロックの個数を制御する。実際には、ワーカー数を超える最小の個数のブロックが構築されるように、二分探索を用いて動的にこの制限値が決定される。最終的に、構築されたブロックはそのサイズによって降順にソートされ、上位のワーカーの個数分のブロックが使用される。literal の個数が少ないブロックは、大きいものと比べより局所的な関係しか得られないため、小さいブロックより大きいブロックを使用する。そのため、ワーカー数より多い数のブロックが構築された場合は、残りの小さなブロックの使用を考慮しない。

```
1. run_count = 0;
2. boostBlock() {
3.   if (run_count++ % INTERVAL > 0)
4.     return;
5.   for USE literals in the given block
6.     increaseVSIDSScore(literal, BUMP_RATIO);
7. }
```

上記の疑似コードは、マスターから割り当てられたブロックに属する変数に VSIDS のスコアを付加する関数 boostBlock 関数の疑似コードである。この関数はワーカー内において restart が行われた直後に呼び出される。なお、対象とするソルバは近年の多くのソルバで使用されている VSIDS を decision heuristic として使用しているものを想定する。この関数が呼び出されることで、割り当てられたブロック内の変数の decision の優先度 (VSIDS のスコア) が上昇され、restart 後の探索において優先的に値が割り振られるようになる。6 行目の increaseVSIDSScore 関数が対象の literal(変数) に対する VSIDS スコアを上昇させる関数であり、“BUMP_RATIO” がスコアの上昇率を表すパラメータである。通常この関数は clause が学習されたときに呼び出され、上昇率は 1 である。restart 後の探索において対象の変数を優先的に decision の対象とするためには、“BUMP_RATIO” は比較的大きな値である必要がある。今回の実験では実験的に妥当だと思われる 10000 を採用している。また、この関数が restart の度に毎回呼び出されると、本来 clause 学習によって体系づけられる VSIDS スコアを過度に破壊してしまう可能性があるため、1 行目のこの関数の呼び出し回数を記録する変数を用いて、3 行目でこの関数の主要部分の処理を “INTERVAL” 回の restart につき一回に制限している。そして、ブロックに含まれている変数は問題によって異なり、多い場合は千個を超える場合もある。それら全ての変数に対して VSIDS のスコアを上昇させてしまうと、優先的に decision すべき変数が多くなりすぎてしまい、探索に悪影響を与える可能性がある。そこで、与えられたブロックから優先すべき変数を “USE” 個に限定する処理を加える。対象とすべき “USE” 個の変数は、ブロック内に含まれる変数はすべて等価という扱いをしているため、特に優先順位をつけずに適当に (ランダムに) 選択する。なお今回の実装では、各ワーカーに割り当てられるブロックは常に固定し、探索中に変更を行うことは無い。

4 実験

本提案を代表的 SAT ソルバ MiniSAT 2.2 [10] に実装し、性能比較実験を行った。MiniSAT 2.2 は逐次のソルバであるため、分散並列環境での使用を想定し、Message Passing Interface (MPI) を用いて並列化を行った。本ソルバを ParaDMiniSAT (Parallel Distributed MiniSAT) と呼ぶこととする。ParaDMiniSAT は search activity diversification として restart の頻度、初期 VSIDS スコアのランダム化、Counter Implication Restart (CIR) [11] を用い、search space diversification として block branching を使用する。全プロセス数 n に対して、一プロセスは探索全体の管理および共有学習 clause データベースの管理を行うマスタープロセスと、n-1 個のワーカープロセスに役割が分かれる。あるワーカーで学習された学習 clause はマスターに送信され、マスターを介して他のワーカーに共有される。マスターはワーカーに対して順番に探索状況および学習 clause に関する問い合わせを行う。その際、二つ以上の同一学習 clause が共有されないように重複防止処理を行っている。また、共有のために送受信される学習 clause は長さ 4 以下の比較的制約の強い clause に限定する。実験に使用するベンチマークは SAT Competition 2011 の ap-

plication 部門の 300 題で、実験に用いるマシンは、Linux OS、二つの Intel Xeon CPU X5680 3.33 GHz の 6 コア CPU(ハイパースレッディングにより実質計 24 コア)、144 GB RAM の構成である。一問に対する実行時間上限は実時間で 5000 秒とする。block branching のパラメータとして、boostBlock 関数内の BUMP_RATIO は 10000、INTERVAL には 5(または 10) を採用し (INTERVAL に 5 なら INT5 と表記)、USE には 30 とブロック内の全ての変数 (USE30 と USEALL と表記) を採用する。なお、使用するプロセス数は 16 とし、各問題に対してそれぞれ二回実行したうちの良い結果を採用する。block branching はよりプロセス数の多い大規模な並列環境を想定しているものの、プロセス数が少ない環境においても diversification を図ることは重要であり、その効果を見ることが可能であると考えられる。

表 1 はそれぞれのソルバの解けた問題数と合計処理時間を表している。表中の “SAT” は時間内に解けた充足可能な問題の問題数、“UNSAT” は充足不可能な問題の問題数、“total” はそれらの和、“time S” は充足可能な問題全 110 問に対する合計処理時間 (解けなかった問題は 5000 秒として換算、単位は秒)、“time U” は充足不可能な問題全 132 題に対する合計処理時間、“total S+U” は全 242 題に対する合計処理時間を表す。三つのソルバにおいてどのソルバでも解けなかった残りの 58 題は除いている。この結果から、block branching を用いない ParaDMiniSAT 単体 (no_bb) より、block branching を用いた bb_INT5_USEALL と bb_INT10_USE30 の結果が、総解答数および合計処理時間の観点において優れていることが分かる。

図 2 は、no_bb と bb_INT10_USE30 の結果を比較したものである。これらのグラフには y 軸に二つの評価軸があり、左側は処理時間を表しており、折れ線グラフが対応する。処理時間に関して、no_bb の結果を各問題の処理時間で昇順にソートしてプロットしたものであり、bb_INT10_USE30 の結果をそれらにあわせてプロットしたものである。すなわち、x 軸の n 番目の値は no_bb の n 番目に早く解けた問題に対応する二つのソルバの処理時間に相当する。そして右側の y 軸の値は、二つのソルバにおける学習 clause の重複率の比率を表しており、棒グラフが対応する。これは、各ソルバにおいて、マスターに送信される学習 clause の重複率をそれぞれ計算し、bb_INT10_USE30 の重複率を no_bb の重複率で割った値を各問題において算出している。学習 clause の重複が多いということは、ワーカー間で探索空間の重複がより生じていることを表しているため、並列効果の観点から重複が少ないことが望ましい。図 2 の左のグラフは充足可能な問題 69 題に対する結果を、右のグラフは充足不可能な問題 106 題に対する結果を示している (学習 clause の共有が全く行われなかった問題等の、重複率が計算できなかった問題は除外している)。まず処理時間の観点では、block branching を用いることで比較的処理時間の大きい問題においてその有効性が見受けられる。そして学習 clause の重複率について、充足可能な問題に関しては 69 題中 57 題で、充足不可能な問題に関しては 106 題中 85 題で block branching を用いることで重複率が少なく (比率が 1 未満) なり、block branching によって様々な空間への探索が促され、ワーカー間で学習 clause の重複がより少なくなっていると考えられる。すなわち、より効果的な diversification を行うことができている証左である。

5 関連研究

portfolio の先駆けとなった並列ソルバは ManySAT [8] であり、それに引き続く形で CryptoMiniSat [12] や Plingeling [13] が登場した。これらのソルバ diversification として search activity diversification を行っており、また短い学習 clause の共有をワーカー間で行っている。これらのソルバは共有メモリ環境に対する実装になっており、ワーカー数が比較的少ない並列環境で動作させることを想定している。

search space diversification を行っている研究として [9] が挙げられる。この研究では、ランダムに三個の変数をそれぞれのワーカーに割り当て、ワーカーはそれらを探索木の根の部分で値を割り当て固定する。本研究ではランダムに選択はせず、CNF から得られる問題の構造に近いものを binary clause から抽出して、それらに含まれる互いに関連の深い変数同士をブロックとしてまとめワーカーに割り当てている。ワーカーは割り当てられたブロック内に含まれる変数を優先的に decision しつつも常に探索木の根に固定せず、柔軟な探索を目指している。

portfolio 以外の手法として、partition tree アプローチ [14] では分割統治法とは異なり、分割後の探索木の葉ノードに該当する部分のみを探索するのではなく、内部ノードも使用することで、分割統治法の欠点を補う探索を実現している。JaCk-SAT [15] は CNF 自体を分割して、二つ以上の独立した問題としてそれぞれの充足解を列挙して、全体を充足する解を発見するソルバである。これらのソルバは portfolio に基づくものではないものの、ワーカーの探索空間を異なるものにさせるという観点から portfolio の diversification として利用する価値があると考えられる。

本研究の提案手法 block branching は、特定のソルバに

依存する処理が無いため、portfolio に基づく並列 SAT ソルバであれば容易に実装が可能であり、より強力な diversification を実現することが可能であると考えられる。

6 まとめ

本研究では、portfolio に基づく並列 SAT ソルバにおける各ワーカーの diversification(探索の多様化) をより強力に実現するために、各ワーカーの探索空間を可能な限り異なるものにすることを目指した search space diversification として block branching の提案を行った。block branching は互いに関連の深いと考えられる変数 (literal) を CNF 中に含まれる binary clause を用いて解析し、それらをブロックとしてまとめ、各ワーカーに割り当てる。各ワーカーは割り当てられたブロック内の変数に対して優先的に decision を行うことで、個々のワーカーが関連の深い変数に対する探索を独立に行いつつ、全体として多様な空間に対する探索を実現することが可能となる。実験の結果、block branching によってより多くの問題をより短時間で解くことが可能となった。また、ワーカー間の共有学習 clause の重複率も減少し、異なる空間への探索を行うことができていることも確認した。今後の課題として、今回の手法では問題の構造として CNF 中に含まれる binary clause を用いて変数のブロック分けを行ったが、より長い clause や学習 clause を考慮したりした、他の分割方法との比較が必要であると考えている。

参考文献

[1] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd annual ACM symposium on Theory of computing*, STOC'71, pp. 151–158, 1971.

[2] Carla P. Gomes and Bart Selman. Algorithm portfolio design: theory vs. practice. In *Proceedings of the 13th conference on Uncertainty in artificial intelligence*, UAI'97, pp. 190–197, 1997.

[3] Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pp. 252–265, 2010.

[4] Joao Marques-Silva. Practical applications of boolean satisfiability. In *Proceedings of Workshop on Discrete Event Systems*, WODES'08, pp. 74–80, 2008.

[5] Alexander Smith, Andreas Veneris, Moayad Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE TRANS. ON CAD*, Vol. 24, No. 10, pp. 1606–1621, 2005.

[6] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European conference on Artificial intelligence*, ECAI'92, pp. 359–363, 1992.

[7] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, Vol. 5, No. 7, pp. 394–397, 1962.

[8] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *JSAT*, Vol. 6, No. 4, pp. 245–262, 2009.

[9] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pp. 443–448, 2009.

[10] Niklas Sörensson. MINISAT 2.2 and MINISAT++ 1.1. *A short description in SAT Race 2010*, 2010.

[11] Tomohiro Sonobe and Mary Inaba. Counter implication restart for parallel SAT solvers. In *Proceedings of the 6th international conference on Learning and Intelligent Optimization*, LION'12, pp. 485–490, 2012.

[12] Mate Soos. CryptoMiniSat 2.5.0. *A short description in SAT Race 2010*, 2010.

[13] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. *A short description in SAT Race 2010*, 2010.

[14] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pp. 372–386, 2010.

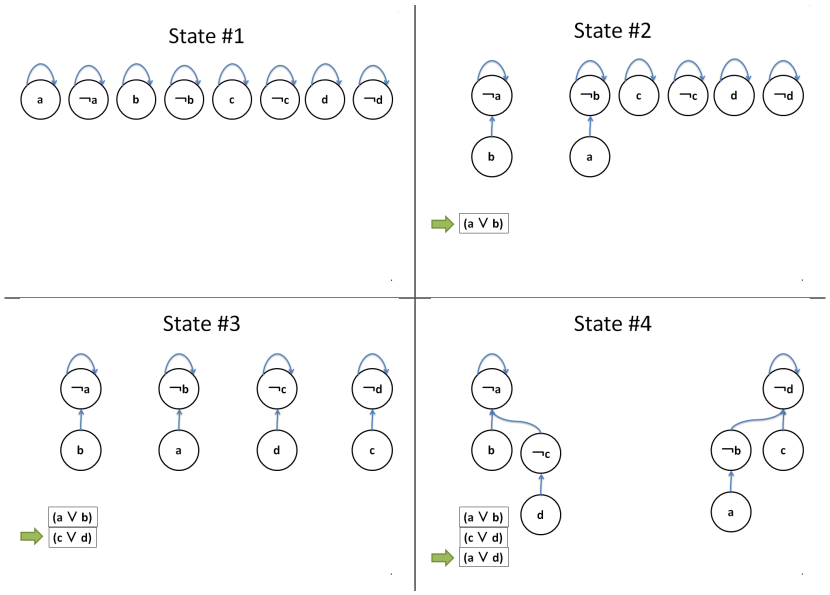

図 1: ブロック構築の例

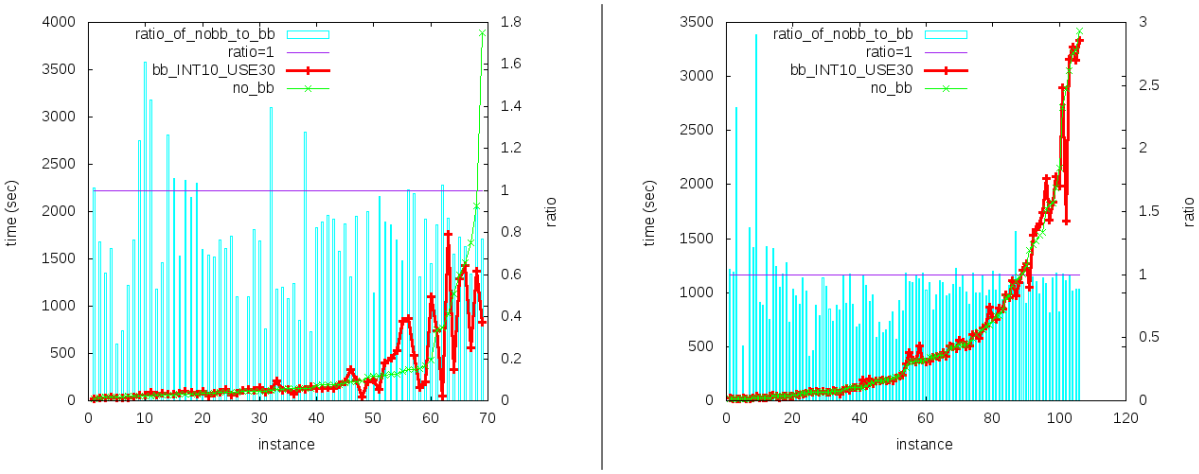

図 2: no_bb と bb_INT10_USE30 の処理時間と学習 clause の重複率の結果 (左が充足可能な問題、右が充足不可能な問題が対象)

[15] Daniel Singer and Anthony Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pp. 249–258, 2008.

表 1: 実験結果の内訳

	SAT	UNSAT	total	time S	time U	time S+U
bb_INT5_USEALL	110	132	242	46150	82600	128750
bb_INT10_USE30	110	132	242	34280	73050	107330
no_bb	108	130	238	48390	78390	126780