SATに対する局所探索法のベクトル化

河合大輔[†] 宮崎修一[††]
岡部寿男[††] 岩間一雄[††]

本研究の目的は，和積形論理式の充足可能性問題（SAT）の解法アルゴリズムである局所探索法の高速化である．局所探索法のアルゴリズムGSATに対し，実働化時の工夫と並列化によって高速化を試みる．SelmanとKautzのGSATを元に，(i) データ構造の改良，(ii) ベクトル並列計算機上でのベクトル化，(iii) PVMを使った40CPUによる並列化，を行った．これにより，合計600倍の高速化を達成した．2nd DIMACS Implementation Challenge, Satisfiability のベンチマーク例題に対して我々のGSATを実行させたところ，既存のプログラムで解けている例題では実行時間がかなり短縮された．また，GSATを改良した局所探索法に本論文の手法を適用させることにより，既存のプログラムでは解けなかった例題を解くまでに至った．

Vectorized Local Search for CNF Satisfiability

Daisuke Kawai,[†] Shuichi Miyazaki,[††] Yasuo Okabe[††]
and Kazuo Iwama[††]

The purpose of this paper is to speed up the local search algorithm for CNF Satisfiability problem by implementation techniques and parallelization. We selected GSAT by Selman and Kautz and attempted speedup by the following techniques: (i) An improvement of the data structure of Selman and Kautz's implementation. (ii) Vectorization on a parallel vector supercomputer. (iii) Parallelization using Parallel Virtual Machine (PVM). By these attempts, we achieved 600-times speedup in total. We executed our GSAT for benchmark instances of the 2nd DIMACS Implementation Challenge, Satisfiability. Our program performed much better than existing programs on most instances. Moreover, we were able to solve some instances that existing programs could not have solved.

1. はじめに

CNF論理式の充足可能性問題（SAT）は基本的な組合せ問題であり，理論および実用の両面から高速アルゴリズムの開発が試みられている．たとえば，バックトラック法[9),17)]，計数法[13)]などのアルゴリズムが知られており，高速化の工夫がなされている．局所探索法[11),22)]は，1990年代の初めに開発されたアルゴリズムである．バックトラック法で解くのには非常に時間のかかる例題を，局所探索法が高速に解くという実験結果が報告され，話題を集めた．また，バックトラック法では数百変数程度の例題を解くのが限界だったのに対し，局所探索法では数千から数万変数の例題を高速に解いてしまうという実験結果が報告されている．局所探索法は，それ以来大きな注目を集めており，さらなる性能向上を目指して様々な改良が行われている[2),3),6),10),15),20),21)]．

本研究の目的は，局所探索法の高速化である．局所探索法の比較的簡単なアルゴリズムであるGSAT[19),22)]に対し，実働化時の工夫と並列化によって徹底的な高速化を試み，性能を検証する．

GSATの高速化のために，まずSelmanらのGSAT[19)]のデータ構造の改良を行った．これにより，約3倍の高速化を達成することができた．次に，京都大学大型計算機センターのベクトル並列計算機VPP800[24)]上で，GSATのベクトル化を行った．2,048個の探索を同時実行することにより，約5倍の高速化を達成した．さらにParallel Virtual Machine[7)]を利用してVPP800の40個のCPUで並列実行させ，ほぼ40倍の高速化を達成した．これらすべての手法を合わせることにより，SelmanらのGSATに対し合計600倍の高速化を達成できた．

† 豊田自動織機製作所
Toyoda Automatic Loom Works Ltd.
†† 京都大学情報学研究科
Graduate School of Informatics, Kyoto University

最後に，2nd DIMACS Implementation Challenge, Satisfiability[5]のベンチマーク例題に対して，我々のGSAT の評価実験を行った．既存のプログラムで解けている例題のほとんどは，解を得るまでの実行時間がかなり短縮された．また，GSAT を改良した Random Walk に対して本研究の高速化手法を適用させたところ，既存のプログラムでは解けていない 2 つの例題を解くことができた．

本論文の構成は以下のとおりである．2 章で SAT および局所探索法を説明する．3，4，5 章で，それぞれ，データ構造の改良，ベクトル化，PVM を用いた並列化による高速化について述べる．6 章では DIMACS ベンチマーク例題に対する実験結果について述べる．最後に 7 章でまとめと今後の課題を述べる．

2. SAT および局所探索法

x_i $(i = 1, 2, \ldots)$ を**変数**といい，真 (1) または偽 (0) の値をとる．変数 x_i およびその否定 $\overline{x_i}$ を**リテラル**といい，リテラルの論理和を**項**と呼ぶ．また，項の論理積を**和積形（CNF）論理式**と呼ぶ．**CNF 論理式の充足可能性問題（SAT）**とは，CNF 論理式 f が与えられたとき，f のすべての項を充足する（1 にする）変数割当が存在するかどうかを問う問題である．

A_1 と A_2 を変数割当てとする．A_1 と A_2 の間で 1 変数（x_i とする）だけ割当てが異なる場合，A_1 と A_2 は互いに**隣接する**といい，A_2 は A_1 から x_i を**フリップ**して得られるという．A を変数割当てとし，A_i を，A から x_i をフリップして得られる変数割当てとする．このとき，A_i での充足項の数から A での充足項の数を減じたものを変数 x_i の（A における）**GAIN** という．局所探索法は最初に変数割当てをランダムに選び，その後，GAIN が最大の変数をフリップすることを繰り返す．すなわち，充足されていない項数が減る割当てへだんだんと進んでいき，最終的には充足されていない項数が 0 の割当て（すなわち充足解）に到達しようというものである．しかし，局所解（すべての変数の GAIN が 0 か負となる変数割当て）に到達すると停止してしまう．この場合に局所解から脱出する手法がいくつか知られている．

GSAT[19),22)] GAIN の最大値が 0 あるいは負でも，GAIN が最大の変数のフリップを続ける．

Weighting[15),20)] 穴（局所解）に落ち込んだら，穴を埋めることによって穴から脱出しようというものである．すなわち，局所解で不充足である項に重みを与える（その項をもう 1 度 CNF 式 f の中に繰り返す）．こうすることにより，現在の変数割当て（局所解）での 0 である項の数（正確には，0 である項の重みの総和）が増加して，現在の変数割当てから別の隣接する変数割当てに移動できるようになる．

Random Walk[21)] 変数フリップの際に，確率 p (≈ 0.5) で，現在の変数割当てで不充足な項からフリップする変数を選択する．また，確率 $1-p$ で通常の GSAT どおりの変数選択を行う．

アルゴリズムの簡単さから，高速化の対象として GSAT を選択した．特に Selman らの GSAT[19)]の高速化について考える．1 秒間あたりのフリップ回数をできるだけ多くすることがここでの目標である．

GSAT には終了条件を定める MAXFLIPS，MAX-TRIES という 2 つのパラメータが必要である．ランダムに初期割当てを選び，フリップを MAXFLIPS 回行うことを **TRY** という．GSAT は TRY を MAX-TRIES 回行う．この過程で充足解が見つかれば，その時点でプログラムは終了する．この過程で解が見つからなければ，失敗して終了となる．

3. データ構造改良による高速化

本章では，**GAIN バケット**（1 フリップの時間短縮のためのデータ構造）を利用した Selman らの GSAT[19)]を紹介し，GAIN バケットの改良による高速化を提案する．

3.1 GAIN バケットの利用

通常の GSAT では，GAIN が最大の変数をフリップする変数とする．そのため，すべての変数の GAIN を調べる必要がある．しかし，GAIN が最大でなくても，GAIN>0 の変数の中からランダムに選ばれた変数をフリップすることにしても性能に影響はないということが実験により分かっている[10)]．Selman らは変数を GAIN>0，GAIN=0，GAIN<0 の 3 つのバケット（それぞれ，**正バケット**，**0 バケット**，**負バケット**という）にグループ分けして，フリップする変数を正バケットの中から選ぶ方法を実装した[19)]．バケットを利用した GSAT では，フリップする変数を正バケットからランダムに選択できる．すなわち，`pb[i]` を正バケットの `i` 番目の変数とするとき，乱数 `random` によって，`pb[random]` を参照するだけでよいため，フリップに要する時間を短縮することができる．

3.2 GAIN バケットの改良

負バケットの除去

上記のように，バケットを利用することの利点は，フリップする変数の選択を容易に行えるということである．ここで，負バケットの変数をフリップする状況

を考える．これは，正バケットにも 0 バケットにも変数が存在しない，すなわち，すべての変数の GAIN が負であることを意味する．したがって，負バケットから 1 つの変数を選択するということは，すべての変数から 1 つの変数を選択することにほかならない．すなわち，負バケットを用いても用いなくても，選択される変数は同じである．

負バケットを除去しても，上記のようにアルゴリズムの動作は変わらないが，以下で述べるように高速化を図ることができる．探索の途中で変数の GAIN が変化すると，バケット間で変数を移動させなければならない．負バケットが存在しないと，負バケットへの挿入・削除が不要となる．GSAT はある程度進行すると，GAIN>0，GAIN=0 の変数が少なくなり，GAIN<0 の変数がほとんどになる．正バケットと 0 バケットの間を変数が移動することは少なく，負バケットへの挿入・削除が頻繁になるので，負バケットの除去によりかなりの高速化が期待できる．

変数のアドレスの利用

たとえば，変数 x_2 の GAIN が正から 0 に変化したとする．このとき，x_2 を正バケットから 0 バケットに移動しなければならない．正バケット中の x_2 を探索するには，i をインクリメントしながら pb[i]=2 となるまで逐次探索が必要である．最悪の場合，正バケット中の変数数に比例した時間がかかる．そこで，各変数に対して，バケットの先頭から何番目にあるかというアドレスを用意する．これにより逐次探索が不要となり，高速化が可能となる．

3.3 性能評価

バケットなしの GSAT （オリジナルの GSAT），3 バケットの GSAT（Selman らの GSAT），アドレスなしの 2 バケット GSAT，アドレスありの 2 バケット GSAT の比較を行う．フリップする変数の選択は，バケットなしの GSAT では GAIN 最大の変数を選択し，バケットありの GSAT では正バケットからランダムに選択している．Selman らの GSAT はプログラムを Web[19]からダウンロードしたものである．**表 1** に各プログラムの SUN ワークステーション UltraSPARC-IIi（300 MHz）上での 1 秒（CPU 時間）あたりのフリップ回数を示す．充足解を持つランダム例題[25]を 10 個ずつ使用し，平均をとった．変数数は 500，1,000，2,000 とし，項数は変数数の 4.3 倍に設定した．いずれの場合も MAXFLIPS は変数数の 10 倍，MAXTRIES は 10 に設定した．アドレスあり 2 バケット GSAT によって，Selman らの GSAT の約 3 倍の高速化を達成できた．表 1 に，2 章で紹介した Random Walk に本章の高速化手法を適用した結果も示す．Random Walk では，充足されていない項の記憶など GSAT より必要な処理が多いため，速度は劣る．

表 1 各プログラムの 1 秒あたりのフリップ回数
Table 1 The number of flips per second executed by each program.

変数数	500	1,000	2,000
項数	2,150	4,300	8,600
バケットなし	74,405	44,964	23,481
3 バケット	103,999	101,437	95,696
2 バケット（アドレスなし）	265,957	226,244	169,492
2 バケット（アドレスあり）	280,899	276,243	222,469
Random Walk	161,391	151,606	126,358

4. ベクトル化による高速化

GSAT では，割当てから割当てに移動する各ステップでフリップする変数を選択し，変数の GAIN と GAIN バケットの更新を行う．よって，各ステップの実行は，前のステップでフリップされた変数に完全に依存する．ゆえに，1 TRY 中の複数のステップをベクトル化して一括実行するのは困難である．これまでは，コスト削減のため，フリップする変数は 1 つのバケットからランダムに選択した．しかし，バケット間で変数の移動があるので，バケットの更新が必要である．ベクトル化によって一度に全変数の GAIN を調べることができれば，バケットを利用しなくても高速化が期待できる．しかし，実験を行ったところ，このやり方では GSAT の 2 倍の高速化にとどまり，3 章で述べたデータ構造改良の方が効果が高いという結果になった．また，GAIN の更新部分はループ回転数が少ないため，ベクトル化がかえって効率を落としてしまう．よって，ベクトル化が意味をなすのはフリップする変数決定部分だけである．これでは，実行時間全体に対するベクトルユニットの稼働時間が半分以下となる．高い効果を得るためには，データ構造改良を残したままベクトルユニットがつねに稼働するベクトル化が必要である．

4.1 TRY 同時実行によるベクトル化

異なる TRY のステップは異なる初期割当てから進行しており，互いに独立なので，異なる複数の TRY のステップはベクトル化が可能である．複数の TRY のステップをベクトル化して一括に実行すれば，複数の TRY が同時に実行されているかのようになる．我々は，この種のベクトル化，すなわち，TRY 同時実行によるベクトル化を採用した．これにより，ベクトルレジスタをつねに稼働させることができる．TRY を実行するループをベクトル化するためには，ループを交換して，そのループを最も内側にする必要がある．

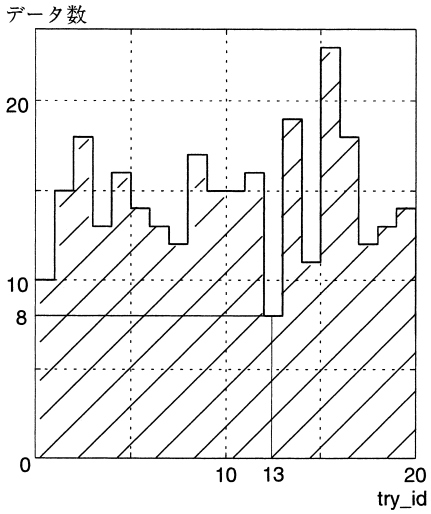

図 1 各 TRY のデータ数の例
Fig. 1 Example of data owned by each try.

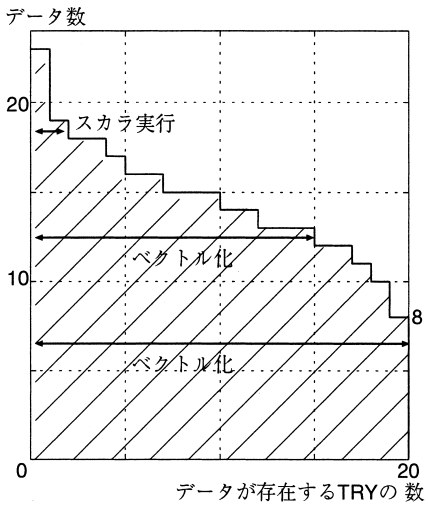

図 2 try_id の収集
Fig. 2 Gathering try_id.

ステップを実行するループの中には，GAIN の更新部分およびバケットの更新部分にループが存在するので，ループ交換後は TRY を実行するループが分割される．分割された各ループがプログラム中の変数を参照するので，変数には TRY の番号を表すインデクス try_id が新たに必要となり，変数は配列になる．

次に，TRY 同時実行によるベクトル化の方法を**図 1** を使って詳しく説明する．GSAT プログラムが配列のデータ，たとえば，フリップによって GAIN が変化した変数を参照する場合を考える．図 1 は各 TRY でのこのようなデータの数の例を表す．簡単のため，20 個の TRY が同時実行されるとする．TRY 逐次実行の GSAT（アドレスあり 2 バケットの GSAT で，ベクトル化されていないもの）では，データ参照の順序は図 1 の縦方向である．よって，参照するデータ数の合計は斜線部分の面積に対応する．一方，TRY 同時実行の GSAT は，ベクトル化を用いて図 1 の 1 つの行を一度に処理する．しかし，データが実際に存在するのは斜線部分だけである．たとえば 13 番目の TRY にはデータが 8 個しかないので，9 行目を実行するとき，ベクトルデータがここで切れてしまう．そこで，データが存在するかどうかを判定する if 文を挿入して，行全体をマスク方式でベクトル化する．

これでベクトル化は可能であるが，次に効率を考えてみる．再び，図 1 を用いて説明する．GSAT プログラムが最初から 8 行目までを実行するときは，20 個のデータをベクトル化によって一括に参照するのでベクトル化の効果が大きい．しかし，20 行目を実行するときは 1 個のデータしか参照しないので，ベクトル化の効果はなく，かえって効率が落ちてしまう．そこで，ベクトル化の効果をより高める方法を考えた．図 1 の各行を実行する際に，次の行でもデータが存在する TRY の try_id を収集演算のベクトル化によって前につめておく．これを**収集**と呼ぶ．**図 2** は図 1 のすべての行に対して収集を行ったものである．収集を行えば，データが存在する TRY だけをベクトル化できるので効率が良い．さらに，データが存在する TRY の数が少ないときは（たとえば，18 行目以降），スカラ実行に切り替えることにより，さらなる効率化を図ることができる．

4.2 性能評価

ベクトル化と収集の効果を検証し，どういう収集方法が効率良いかを調べるため，以下の 6 個のプログラムで実験を行った．以下のプログラムはすべて 3 章で述べた 2 バケット（アドレスあり）に基づいている．

(1) ベクトル化を用いない．
(2) 収集せずベクトル化だけを用いる．
(3) すべての行を毎回収集する．
(4) 一定回（ここでは 2 回）おきに収集する．
(5) 最初の数回（ここでは 3 回）はデータ数が多いので収集せず，その後は毎回収集する．
(6) 各 TRY に存在するデータ数の最小値 min（図 2 では 8）をあらかじめ求める．最初から min 行目まではすべての TRY にデータが存在するので収集を行わない．その後は毎回収集する．

プログラム 2 から 6 はベクトル化を用いており，2,048 TRY を同時に実行する．よって，1 ステップで 2,048 回のフリップをすることになる．そのうち，プ

表 2 各プログラムでの 1 秒あたりのフリップ回数
Table 2 The number of flips per second executed by each program.

		変数数	100	200
		項数	430	860
		1	287,799	312,898
ベクトル化		2	840,699	858,256
	収集	3	1,558,935	1,691,241
		4	1,404,014	1,530,819
		5	1,494,083	1,612,150
		6	1,553,924	1,707,838

ログラム 3 から 6 は収集も行い，データ数が少ないときはスカラ実行に切り替えている．**表 2** は京都大学大型計算機センターのベクトル並列計算機 Fujitsu VPP800〔1 GOPS（Giga operations per second），8 GFLOPS（Giga floating-point operations per second）per CPU〕[24]での各プログラムの 1 秒（CPU 時間）あたりのフリップ回数を表す．例題は表 1 と同様の 100 変数と 200 変数の例題 100 個ずつを使用し，平均をとった．項数は変数数の 4.3 倍に設定した．MAXFLIPS を変数数の 10 倍とし，MAXTRIES は TRY 数の合計が等しくなるようにプログラム 1 では 2,048，プログラム 2 から 6 では 1 とした．成功率（総例題数に対する解が得られた例題の数）はいずれの方法も同じで，100 変数でほぼ 100%，200 変数でほぼ 50%であった．

表 2 から分かるように，収集を行わなくてもベクトル化のみでほぼ 2.5 倍の高速化が得られた．さらに収集を行えば，収集の方法にかかわらず，合計 5 倍の高速化が達成できた．ただし，2,048 個も TRY を同時実行しているのに 5 倍の高速化はベクトル化にしては遅い．高速化が十分でない原因は，ループ交換による変数の配列化である．これにより間接参照が増加し，ベクトル化の際にはベクトル間接参照の増加につながる．TRY 同時実行の GSAT では，TRY ごとにデータ数が異なるため収集が必要であるだけでなく，ベクトル間接参照が頻繁に現れ，避けるのが困難である．

5. PVM による高速化

前章では，ベクトル化を用いて 5 倍の高速化を達成した．本章では，さらなる高速化のため複数の CPU を用いて並列化を行う．4.1 節でも述べたように，GSAT において，異なる TRY のステップは互いに独立である．よって，複数の TRY はベクトル化のみならず，並列化も可能である．並列化には Parallel Virtual Machine（PVM）[7]を用いた．並列化の方法は以下のとおりである．1 つの CPU がマスター，その他の CPU がスレーブとなり，スレーブは TRY（ベクトル化された 2,048 TRY）を実行する．完全な並列化を行うため，各スレーブごとに異なる乱数の種を与える．VPP800 では 40 個の CPU が利用できるので，$2{,}048 \times (40-1)$ の TRY を同時実行できる．各スレーブは TRY が終了するごとにマスターに通信する．マスターは終了した TRY 数を数えて，MAXTRIES に達したらプログラムを終了する．あるいは，1 スレーブでも解が得られたらプログラムを終了する．このように，必要な通信が非常に少ないため，スレーブの台数分の高速化が期待できる．

表 3 40 CPU と 1 CPU の場合の 1 秒あたりのフリップ回数
Table 3 The number of flips per second made by 40 processors and 1 processor.

変数数	100	200	500
項数	430	860	2150
40 CPU	63,269,478	63,825,510	67,817,472
1 CPU	1,677,181	1,657,016	1,726,874
比率	37.72 倍	38.52 倍	39.28 倍

並列化の効果を調べるために，40 CPU を使用した場合と 1 CPU を使用した場合の 1 秒（経過時間）あたりのフリップ回数を VPP800 上で比較した（**表 3**）．実行したプログラムは 2 バケット（アドレスあり）を毎回収集してベクトル化したものである．100 変数，200 変数，500 変数の例題を 5 個ずつ用いて平均をとった．項数はこれまでと同様に変数数の 4.3 倍である．今回は，正しい結果を得るため十分長く実行する必要があるので，充足解を持たない例題[1]を使用した．いずれの場合も期待どおり約 40 倍の高速化が得られた．

6. DIMACS ベンチマークに対する実験

これまでに述べたいくつかの高速化手法によって，1 秒あたりのフリップ回数において Selman らの GSAT の約 600 倍の高速化が達成できた．実際の性能を検証するため，本論文の手法で高速化した我々の GSAT を 2nd DIMACS Implementation Challenge，Satisfiability のベンチマーク例題[5]に対して実行させた（**表 4**）．比較のため Selman らの GSAT を UltraSPARC-IIi（300 MHz）上で実行させた結果と，DIMACS の予稿集[5]に記載されている 7 つのプログラムの実行結果も表 4 に載せる．例題は充足解のあるものだけを用い，解を見つけるまでの時間を計測した．表 4 の各行に例題名と各プログラムがその例題を解くのに必要であった実行時間〔経過時間（秒）〕を示す．小数点以下は切り捨てた．すなわち，表中の 0 は 1 秒未満で解けたことを意味する．我々の GSAT

表4 DIMACSベンチマークに対する結果〔各行は，例題名，変数数，項数，それぞれのプログラムにより解かれた時間（秒）を表す．少数点以下は切り捨てている．並列化GSATおよびSelmanらのGSATは我々の実験環境での結果であり，プログラム番号1〜7の結果は，文献5)の結果をそのまま転記したものである．プログラム番号1〜7は表5の文献に対応している．空欄は，我々が行った実験では6時間以内に解を得られなかったことを意味し，プログラム1〜7については，文献に記載されていなかった（解けなかった，または実行しなかった）ことを表す．並列化GSATの列の結果の中で * 印の付いているものは，並列化Random Walkにより解かれたものである〕
Table 4 Experimental results for DIMACS benchmark instances.

例題名	変数数	項数	実行時間（秒）								
			並列化	Selman	他のSATプログラム[5]						
			GSAT	GSAT	1	2	3	4	5	6	7
aim-100-2_0-yes1-1	100	200	1	227	0	17	135	2	398		1
aim-100-2_0-yes1-2	100	200	2	96	0	29	2	5	239	13,929	1
aim-100-2_0-yes1-3	100	200	1	49	0	13	17	1	63	22,500	1
aim-100-2_0-yes1-4	100	200	2	226	0	15	0	0	1,456		0
f400.cnf	400	1,700	3	48	2,844	34		5,727	210,741	60	10,870
f800.cnf	800	3,400	182			1,326				27,000	
f1600.cnf	1,600	6,800	*509								
f3200.cnf	3,200	13,600	*19,840								
g125.17.cnf	2,125	66,272	261			103,310				453,780	
g125.18.cnf	2,250	70,163	17	4		126				480	
g250.15.cnf	3,750	233,965	17	1		72				120	
g250.29.cnf	7,250	454,622	2,751							398,820	
ii32b3.cnf	348	5,734	15	55	2	4	4	1	1	5,400	17
ii32c3.cnf	279	3,272	8	3	1	3	0	5	1	12,180	
ii32d3.cnf	824	19,478	38	25	973	19	10	3	20	1,200	
ii32e3.cnf	330	5,020	11	0	1	3	4	1	3	3,900	3
par16-2-c.cnf	349	1,392	183		23		329	48	1,464		145
par16-4-c.cnf	324	1,292	554		6		210	116	1,950		145
ssa7552-159.cnf	1,363	3,032	*30		1	82	6	1	1		1
ssa7552-160.cnf	1,391	3,126	*40		1	86	6	1	23	175,500	1

表5 表4中の番号とSATプログラム[5]の対応と計算機の性能
Table 5 Authors of each SAT program in Table 4 and machine performance.

番号	著者	プログラム名	* Ratio
	Selman, B., Kautz, H.	GSAT（3バケット利用）[19]	0.95
1	Dubois, O., Andre, P., Boufkhad, Y. Carlier, J.	C-SAT (backtracking)[4]	3.34
2	Hampson, S., Kibler, D.	Cyclic, Opportunistic Hill-Climing[12]	7.43
3	Jaumard, B., Stan, M. Desrosiers, J.	Davis-Putnum-Loveland procedure[14]	10.82
4	Pretolani, D.	H2R, BRR (pruning)[16]	18.03
5	Resende, M.G.C., Feo, T.A.	Greedy Randomized Adaptive Search Procedure (GRASP)[18]	1.84
6	Spears, W.M.	SASAT (Simulated Annealing)[23]	4.63
7	Gelder, A.V., Tsuji, Y.K.	2cl (Combination of branching and limited resolution)[8]	3.14

*Ratio: VPP800に対するプログラム実行計算機の計算時間比

とSelmanらのGSATは6時間実行した．空欄は6時間で解けなかったことを表す．他の7つのプログラムでは，空欄は結果が掲載されていない（解けなかったか，または実行しなかった）ことを表す．**表5**に各プログラムの著者と名前を示す．表5のプログラム番号は表4と対応している．また，各プログラムを実行するマシンの性能の違いを比較するため，DIMACSがテストプログラムを提供しており，各著者は自分のプログラムの実行環境でのそのテストプログラムの実行時間を記載している．我々もここで使用したマシン（VPP800およびUltraSPARC-IIi）でのテストプログラム実行時間を計測した．表5のRatioの欄に，VPP800（スカラ実行の場合）に対する，テストプログラムの計算時間比を示す．たとえば，1番のプログラムを実行したマシンは，VPP800のスカラ実行に比べて3.34倍遅いことになる．

Selman らの GSAT とプログラム番号 2 と 6 は，局所探索法をベースにしたプログラムである．また，プログラム番号 1，3，4，7 は，バックトラック法をベースにしたプログラムである．g 例題は，バックトラック法には解かれていないが，局所探索法では解けていることが分かる．逆に par 例題は，バックトラック法では解けているが，局所探索法では解けていない．我々の並列局所探索法は，局所探索法にとって得意な例題を高速に解いているばかりか，局所探索法では解けなかった例題も，バックトラック法に匹敵する速度で解いていることが分かる．さらに，これまでどのプログラムも解くことができなかった例題 f1600.cnf を 8 分，f3200.cnf を 5 時間 30 分で解くことができた．

7. おわりに

本論文では，Selman らの GSAT に対し，データ構造の改良と VPP800 上でのベクトル化・並列化により合計 600 倍の高速化を達成した．そして，DIMACS のベンチマーク例題に対する結果から，本研究の高速化手法の有効性が評価できた．今後は各種の局所探索法に本研究の高速化手法を適用し，様々なタイプの例題集合に対して実験を行う必要がある．

参考文献

1) Asahiro, Y., Iwama, K. and Miyano, E.: Random generation of test instances with controlled attributes, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.377–393, American Mathematical Society (1996).
2) Cha, B. and Iwama, K.: Performance test of local search algorithms using new types of random CNF formulas, *Proc. IJCAI-95*, pp.304–310 (1995).
3) Cha, B. and Iwama, K.: Adding new clauses for faster local search, *Proc. AAAI-96*, pp.332–337 (1996).
4) Dubois, O., Andre, P., Boufkhad, Y. and Carlier, J.: SAT versus UNSAT, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.415–436, American Mathematical Society (1996).
5) Johnson, D.S. and Trick, M.A. (Eds.): Cliques, Coloring, and Satisfiability, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, American Mathematical Society (1996).
6) Fukunaga, A.S.: Variable-selection heuristics in local search for SAT, *Proc. AAAI97*, pp.275–280 (1997).
7) Geist, A., Berguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: *Parallel Virtual Machine*, The MIT Press (1994).
8) Gelder, A.V. and Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.559–586, American Mathematical Society (1996).
9) Goldberg, A., Purdom, P. and Brown, C.: Average time analysis of simplified Davis-Putnam procedures, *Information Processing Letters*, Vol.15, pp.72–75 (1982).
10) Gent, I., and Walsh, T.: Unsatisfied variables in local search, *Hybrid Problems, Hybrid Solutions* (*AISB-95*), Amsterdam (1995).
11) Gu, J.: Efficient local search for very large-scale satisfiability problems, *Sigart Bulletin*, Vol.3, No.1, pp.8–12 (1992).
12) Hampson, S. and Kibler, D.: Large plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.437–455, American Mathematical Society (1996).
13) Iwama, K.: CNF satisfiability test by counting and polynomial average time, *SIAM J. Comput.*, Vol.12, pp.385–391 (1989).
14) Jaumard, B., Stan, M. and Desrosiers, J.: Tabu search and a quadratic relaxation for satisfiability problem, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.457–478, American Mathematical Society (1996).
15) Morris, P.: The breakout method for escaping from local minima, *Proc. AAAI-93*, pp.40–45 (1993).
16) Pretolani, D.: Efficiency and sability of hypergraph SAT algorithms, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.479–498, American Mathematical Society (1996).
17) Purdom, P. and Brown, C.: An analysis of backtracking with search rearrangement, *SIAM J. Comput.*, Vol.12, pp.717–733 (1983).
18) Resende, M.G.C. and Feo, T.A.: A GRASP for satisfiability, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.499–520, American Mathematical Society (1996).
19) Selman, B. and Kautz, H.: GSAT USER'S GUIDE Version 35. ftp://ftp.research.att.com/dist/ai/GSAT_USERS_GUIDE.Z (1993).
20) Selman, B. and Kautz, H.: An empirical study

of greedy local search for satisfiability testing, *Proc. AAAI-93*, pp.46–51 (1993).
21) Selman, B. and Kautz, H.: Local search strategies for satisfiability testing, *2nd DIMACS Challenge Workshop* (1993).
22) Selman, B., Levesque, H.J. and Mitchell, D.G. A new method for solving hard satisfiability problems, *Proc. AAAI-92*, pp.440–446 (1992).
23) Spears, W.M.: Simulated annealing for hard satisfiability problems, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.26, pp.533–557, American Mathematical Society (1996).
24) 京都大学大型計算機センター，ベクトル並列計算機 VPP800/63．http://www.kudpc.kyoto-u.ac.jp/Supercomputer/
25) Watanabe Lab, Dept. of Comp. Science, Tokyo Institute of Technology, Random Generation of Unique Solution 3SAT Instances. http://www.is.titech.ac.jp/˜watanabe/gensat/a1/index.html

(平成 12 年 8 月 24 日受付)
(平成 12 年 12 月 1 日採録)

河合 大輔

昭和 50 年生．平成 12 年京都大学大学院情報学研究科通信情報システム専攻修士課程修了．同年豊田自動織機製作所入社．

宮崎 修一（正会員）

昭和 45 年生．平成 5 年九州大学工学部情報工学科卒業．平成 7 年同情報工学専攻修了．平成 10 年同博士課程修了．博士（工学）．平成 10 年より京都大学大学院情報学研究科助手．計算の複雑さ理論の研究に従事．電子情報通信学会会員．

岡部 寿男（正会員）

昭和 39 年生．昭和 63 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年同大学工学部助手．同大学大型計算機センター助教授を経て，現在同大学大学院情報学研究科助教授．博士（工学）．並列アルゴリズム等の研究に従事．電子情報通信学会，システム制御情報学会，日本ソフトウェア科学会，IEEE，ACM，EATCS 各会員．

岩間 一雄（正会員）

昭和 26 年生．昭和 48 年京都大学工学部電気工学科卒業．昭和 55 年同大学院博士課程修了．工学博士．昭和 53 年京都産業大学理学部計算機科学科講師．昭和 57 年同助教授．昭和 58 年より 59 年までカリフォルニア大学バークレー客員準教授．平成 2 年九州大学工学部情報工学科助教授，平成 4 年同教授を経て，平 9 年京都大学大学院工学研究科情報工学専攻教授．現在同大学院情報学研究科教授．アルゴリズムと計算の複雑さの理論の研究に従事．電子情報通信学会，ACM，SIAM 各会員．