Vol.2018-HPC-167 No.24
2018/12/18

スケーラブルな並列探索による最適化問題の求解

泉 翔太[1,a)]　石井 大輔[1]　美添 一樹[2]

概要：Branch-and-bound 等の探索に基づく最適化手法の並列化は HPC の古典的テーマである．Saraswat らが提案した大域的負荷分散 (GLB) ライブラリは，同型の並列ワーカーからなる機構を提供し，不均一なタスク (例: 並列探索) を 16K コア程度までスケールさせることができる．しかし，GLB ライブラリを最適化問題へ適用すると，暫定解を効率よく並列ワーカー間で共有することが課題となる．そこで，本研究では GLB ライブラリを拡張し，暫定解の共有速度と通信コストのトレードオフを解決することを目指す．また，拡張 GLB ライブラリのベンチマークとして，木構造に基づく最適化問題を用いる．ワーカー間で暫定解を共有する機能として，通信ワーカーを動的に選択する方式と，ワーカー間の超立方体ネットワークを用いる方式を実装した．拡張 GLB ライブラリを用いてベンチマーク問題の並列ソルバーを実装し，複数インスタンスを求解する評価を実験したところ，784 コア使用時に 429–824 倍の速度向上 (並列化効率 0.55–1.05) を達成した．

Solving Optimization Problems with Scalable Parallel Search

Shota Izumi[1,a)]　Daisuke Ishii[1]　Kazuki Yoshizoe[2]

1. はじめに

最適化問題は変数の集合，制約，目的関数が与えられ，制約を満たしながら目的関数の値を最小化するような変数への付値を求める問題である．Branch-and-bound 等の探索アルゴリズムに基づく求解法が提案されている．サイズが大きく複雑な問題インスタンスを扱うために最適化器を PC クラスタ上で並列分散動作させるための研究が行われてきた．しかし，ほとんどの既存手法は並列タスク間に中央集権的な構造を採用しており，大規模並列環境でのスケーラビリティに限界があった (2 節)．

本研究は，最適化問題を並列求解するための均一的な構造をもつ並列分散化スキームを提供することを目的とする．そのために，種々の最適化問題に対する探索に基づく求解処理を並列化するためのライブラリを提案する (4 節)．提案ライブラリは高生産 HPC プログラミング言語 X10 に付属する GLB ライブラリ (3 節) を拡張して実現する．最適化問題の求解では途中で見つかった暫定解を他の並列ワーカーへ伝播する必要があるが，3 つの暫定解共有方法を設計・実装する．また本研究では，提案ライブラリの性能測定のために，木構造に基づくベンチマーク最適化問題を用いる (5 節)．拡張 GLB ライブラリとベンチマーク問題のソルバーを実装し (6 節)，スーパーコンピュータの 784 コアを使用してベンチマークを求解する評価実験を行った．その結果を報告し (7 節)，実験結果について考察を行う (8 節)．

2. 関連研究

並列探索による最適化問題の求解は HPC 分野の古典的なテーマであり，教科書 ([5] 等) でも章が割かれている．

こうした研究では，線形計画問題や制約付き最適化問題 [1], [7]，グラフィカルモデルの最適化問題 [11] など，個別の問題ごとに効率的な求解法を検討する．本研究では木構造に基づく基本的なベンチマークを想定し，さまざまなパラメタ値のもとで効率的な並列求解法を検討する．

並列木探索手法は有力な求解法の 1 つであり，深さ優先

[1] 福井大学
University of Fukui
[2] 理化学研究所
RIKEN
[a)] i170011@u-fukui.ac.jp

探索 [7]，branch and bound アルゴリズム [5]，AND/OR 木探索 [8], [11]，モンテカルロ木探索 [13]，局所探索 [10] など，多数の研究が行われている．本研究で提案するライブラリもおもに並列木探索への応用を想定したものである．

ワークスティーリングに基づき探索木を並列ワーカーへと分割・分散させるのは並列木探索に有用な手段であり，多数の手法が提案されている (例: [7])．多くの研究では共有メモリマシンや疎結合の PC クラスタを想定してきた．近年はより密結合の大規模 PC クラスタが稼働しており，そうした環境を前提としたワークスティーリング技術が提案されている [2], [3], [11]．本研究ではそのようなワークスティーリング技術の 1 つである GLB ライブラリ [12], [14] を拡張する．

X10 を用いて実装された並列木探索に基づくソルバーが提案されている [1], [2], [6], [10]．それぞれの既存ソルバーは特定の問題や解法に依存しているが，本研究では一般の最適化問題向けに負荷分散機能を汎用化することを目指す．

3. X10 言語と GLB ライブラリ

X10[*1] は APGAS モデルに基づいた HPC のための高生産プログラミング言語である．分散した計算ノードを表す**プレース** やプレース内で並列計算を行う**並行タスク**等の概念を用いて，PC クラスタ等の HPC 環境を抽象化し，高水準なプログラミングを可能にしている．

GLB (global load balancing)[12], [14] は X10 の標準ライブラリに含まれるソフトウェア部品で，協調動作する並列ワーカーのための負荷分散および実行終了のための機能を提供する．GLB は，各ワーカーの処理が同一で，かつワークロードの予測が難しい場合に向いている．

GLB を使用するには，ユーザーがまず問題固有の処理を TaskQueue クラスとして実装する必要がある．TaskQueue には求解の単位逐次処理を記述した process メソッド，ワークロードの分割処理，求解結果を扱うためのメソッド群を実装する．GLB による並列計算では，各ワーカーが各 CPU コアに配備され，図 1 のような処理を行う．おもな処理は **repeat** 部 (2–11 行目) であり，ワークロードを処理する **while** 部 (3–8 行目) と負荷分散処理 (9,10 行目) からなる．**while** 部は n 単位ずつワークロードを処理するとともに，他ワーカーへのワークロードの分散処理を行う．

負荷分散処理は 2 ステップのスティール処理からなる．まず w ワーカーを乱択して行い，その後はライフラインと呼ばれるワーカー間ネットワークの隣接ノードに対して行う．スティールの通信に対しては，ワークロードをもつワーカーの DistributeOpt 処理 (7 行目) が返信し，ワークロードの部分をスティール元へ分け与える．ライフラインのトポロジは直径 l，分岐数 z の超立方体である (ユーザが l を指定し，z は $\lceil \log_l(\text{ワーカー数}) \rceil$ とする)．StealViaLifeline 後にワークロードが得られなければ，大域的にワークロードがなくなったことが保証される．

Input: 環境 E, TaskQueue インスタンス Q
Output: タスクの結果
Parameter: $n, w, l, z \in \mathbb{N}$

```
1:  LL := InitLifeline_E(l, z)
2:  repeat
3:    while Q.process(n) do
4:      if Q.isOptUpdated() then
5:        DistributeOpt_{E,LL}(Q)
6:      end if
7:      DistributeToThieves_E(Q)
8:    end while
9:    StealRandomly(E, Q, w)
10:   StealViaLifeline(E, Q, LL)
11: until Q.isEmpty()
12: return Q.getOptimum()
```

図 1 拡張 GLB のワーカープロセス

GLB は簡潔な並行求解処理の記述と，同型の並列ワーカー群による計算により，スケーラブルな並列処理を可能にする．文献 [14] では，グラフ中の媒介性解析とアンバランス木探索について，16K コアまでのスケーラビリティを得た結果が報告されている．文献 [6] では数値制約充足問題の探索空間を分割，各並列ワーカーへ分散し，600 コアを使用して最大 516 倍の速度向上を達成している．

4. 最適化問題のための拡張 GLB ライブラリ

最適化問題の求解について同様に GLB による並列化を考える場合，求解中に見つかった暫定的な最適解を並列分散環境上で共有し，ワークロードの枝刈りを行ったり，大域的最適解を同定したりする必要がある．そこで本研究では GLB ライブラリを拡張し，暫定解 (各並列ワーカーが受け持つ部分探索空間内での最適解) を他の並列ワーカーへ効率よく伝播する処理を検討する．

TaskQueue クラスはワーカーに対し，ワークロードと処理中に得られた暫定解を扱う機能を提供する．本拡張に関連するメソッドを説明する．

- process(n): ワークロードを n 単位分処理する．戻り値として処理後にワークロードが残っているかどうかを返す．
- isOptUpdated(n): 直前の process の最中に暫定解が更新されたかどうかを返す．

図 1 の 4–6 行目が追加した伝播処理である．ワークロードの処理中に保持している暫定解が更新された場合，その値を他ワーカーへと送信する．暫定解の伝播処理 (5 行目の DistributeOpt) としては以下の 3 方式を検討する．

[*1] http://x10-lang.org

- **ブロードキャスト方式** (以下「B 方式」と呼ぶ)．全ワーカーへ送信する．
- **乱択方式** (パラメタ: x．R 方式)．一様無作為に選んだ x ワーカーへ送信する．
- **ライフライン方式** (パラメタ: l, z．L 方式)．パラメタ値 (l, z) のライフライン上で隣接するワーカーへ送信する．ライフラインは負荷分散処理と共用する．

5. ベンチマーク最適化問題

本節では，拡張 GLB ライブラリの性能評価においてベンチマークとするための最適化問題について述べる．

完全木に基づく最適化問題 (perfect-tree-based optimization problem, 以下「PTO 問題」と呼ぶ)．深さ d の完全 b 分木を考える．各ノードには 0–255 の重みをシード値 r に基づき一様無作為に付与する．目的関数はあるパス上の重みの合計とし，最小の重み合計値が最適解となる．

PTO 問題は文献 [3], [12], [14] に述べられている UTS (unbalanced tree search) 問題を参考にして設計した．UTS 問題は与えられた木の全ノードを走査することを目的とした問題である．本研究の PTO 問題は乱択した重み付きの木構造に関する最適化問題となっている．また，2 人ゲームの解析に用いられる P-game と呼ばれる方法を参考にしている [4], [9].

ベンチマークの要件として以下のような項目を検討する．

(1) 探索空間の幅と深さを設定でき，探索空間の大きさを容易に見積ることができる．
(2) 可能な枝刈りの数を調節でき，必ず探索が必要な空間を容易に見積ることができる．

本研究では PTO 問題の解法として，与えられた完全木の各パスの重み合計値を求め，その中から最小値を算出する方法をとる．そのため問題インスタンス中の木が探索空間を表すものと考えられる．すると，要件 (1) については，パラメタ b と d により探索空間の幅と深さを設定することができ，その大きさ $\sum_{i=0}^{d}(b^i)$ も得ることができる．要件 (2) については，b, d から定まる木とシード値 r を変更することにより可能な枝刈りの数が調節できる．その際，完全木の内容は，最適値を判定するために必ず探索が必要な部分木と，それ以外のノードへと静的に分類することができる．

PTO 問題の逐次求解アルゴリズムを図 2 と 3 に示す．また図 3 の処理を拡張 GLB の TaskQueue.process とすれば，並列求解が可能になる．図 3 のアルゴリズムは動的にノードのデータ構造 N を生成しながらスタック S に格納し，完全木の深さ優先探索を行う．スタックには根ノードから現在ノード N までの累積和 W も格納する．暫定的な最適解 O と W を比較することにより部分木の探索を打ち切ること (枝刈り) ができる (8 行目)．また，スタックから

Input: PTO 問題 (b, d, r)
Output: 最小の重み合計
1: $N := \mathsf{RootNode}_r()$;　$S := \mathsf{InitStack}(N)$
2: $O := \infty$;　$\mathsf{process}(\infty)$;　**return** O

図 2 PTO 問題の逐次求解処理

Input: 処理ノード数 n
Output: ワークロードが残っているかどうか
Parameter: PTO 問題 (b, d, r)，スタック S，暫定解 O
1: **for** $i \in \{1 \ldots n\}$ **do**
2:　**if** $|S| = 0$ **then break end if**
3:　$(N, W, S) := \mathsf{Pop}(S)$
4:　**if** $\mathsf{Depth}(N) < d$ **then**
5:　　$NS := \mathsf{Expand}_b(N)$
6:　　**for** $N' \in NS$ **do**
7:　　　$N' := \mathsf{Hash}_r(N, N')$
8:　　　$W := W + \mathsf{Weight}(N')$
9:　　　**if** $W < O$ **then** $S := \mathsf{Push}(N', W, S)$ **end if**
10:　　**end for**
11:　**else**
12:　　**if** $W < O$ **then** $O := W$; **break end if**
13:　**end if**
14: **end for**
15: **return**　$|S| > 0$

図 3 PTO 問題の n 単位分の求解処理 (TaskQueue.process())

のポップ時 (5 行目) に (スタックトップ近傍から) 重みが小さいノードを優先的に取り出すようにすることで求解処理を効率化できる．

6. 実装

X10 処理系バージョン 2.5.4 の GLB ライブラリに対して，4 節で述べた最適化問題向けの拡張を行った．暫定解の伝播方式は B, R, L の 3 方式を設定により選択できる．拡張 GLB ライブラリを用いて PTO 問題の並列ソルバーを実装した．また，評価実験用の逐次ソルバーを X10 で図 2 の処理を実装することで作成した．並列・逐次ソルバーともに Pop 処理は，スタックトップから同一の深さのノードを辿り，その中から最小の累積重みをもつものを選択する処理とした．$\mathsf{Hash}(N, N')$ 処理は，Zobrist ハッシュ法に基づきノード N' のハッシュ値を計算する．これにより並列処理中に動的にノード固有のハッシュ値を求めることができる．Zobrist ハッシュ法ではまずシード r に基づき乱数テーブルを作成しておく．ノード N' のハッシュ値は N のハッシュ値と N' の位置に基づく乱数テーブルのエントリとの排他的論理和とする．$\mathsf{Weight}(N)$ 処理は (N のハッシュ値) mod 256 とした．

実験では X10 処理系のネイティブ (C++) コンパイラと

MPI バックエンド (Open MPI 3.0.1) を用いた.

7. 実験結果

PTO 問題をスーパーコンピュータ上で逐次ソルバーおよび並列ソルバーで求解し，並列化や暫定解伝播の 3 方式の効率を評価する実験を行った．実験環境には理化学研究所の RAIDEN (Broadwell EP) を利用した．RAIDEN の各ノードは 2 つの Intel Xeon E5-2690 v4 2.6GHz プロセッサ (計 28 コア) と 256GB のメモリを備えており，本実験では 28 ノード (計 784 コア) を用いた.

X10 の 1 プレースを RAIDEN の 1 コアに割り当て，GLB の 1 ワーカーが 1 プレースを専有する．GLB のパラメタは，$n = 20000, w = 4, l = 2$ と設定した．また乱択方式のパラメタを $x = 3$ と設定した.

実験結果を表 1 に示す．各列は実験で求解した各インスタンス (a), . . . , (f) に対応し，各行の内容は順に以下のようになっている.

- 問題パラメタの値.
- 逐次ソルバーが要した実行時間.
- 並列ソルバーを 784 コア上で実行した際の速度向上 (暫定解伝播の 3 方式ごと).
- 完全木のノード数.
- 完全木中で必ず探索が必要な部分木のノード数 (各パスの重み合計が (大域) 最適解以下となる木).
- 逐次ソルバーが実際に探索したノード数[*2].
- 並列ソルバーが探索したノード数 (784 コア使用時，暫定解伝播の 3 方式ごと)．括弧内には (784 コア使用時の) 各並列ワーカーが探索したノード数の最大値を記載.
- 暫定解の通信回数 (784 コア使用時，暫定解伝播の 3 方式ごと)．暫定解を発見した回数と，括弧内に暫定解をワーカー間で送信した総回数を記載.

並列ソルバーが各暫定解伝播方式を用いて達成した速度向上を，インスタンスごとに図 4 に示す.

8. 考察

実験では L 方式を用いた並列処理が最も効率がよく，784 コア使用時に 429–824 倍の速度向上 (並列化効率 0.55–1.05) を得た．本稿で報告していない他インスタンスについても同様の結果を得ており，同規模のインスタンスであればこの範囲の並列化効率が得られるものと考えている.

PTO 問題の求解処理を並列化すると，各並列ワーカーが個別に暫定解を発見して枝刈りを実施するため，逐次求解時よりも枝刈りが効き，探索ノード数が減ることがある．実験結果中では，インスタンス (a) と (e) の探索ノード数 (L 方式による求解時) が減っている．インスタンス (e) ではこれに起因して線形を越える速度向上を達成したものと考えられる.

暫定解伝播の 3 方式に関しては，784 コア使用時のほとんどの場合 (インスタンス (f) 以外) で L 方式が R 方式よりも良好な速度向上を達成した．また，すべての場合で L 方式と R 方式は B 方式よりも良好な速度向上を達成した．L 方式はすべての場合で他の方式よりも少ない探索ノード数を達成した．L 方式の探索ノード数は，インスタンス (a) と (e) で逐次の結果よりも減ったが，他インスタンスでも高々 8% 増えたのみであった．R 方式は L 方式よりも数割多い程度だったが，B 方式は数倍まで増えたインスタンスがあった．暫定解の通信回数について比較すると，R 方式がインスタンス (d) を除いて最も少ない総通信回数となった．しかし速度向上と探索回数削減は L 方式に及ばなかったため，最適解伝播の通信量が十分でなかったと考えられる．インスタンス (d) でも L 方式に及ばなかったので，伝播の効率も悪い恐れがある．B 方式では他の方式と比べて総通信回数が数倍程度多いが，暫定解の発見回数は数 % 程度である．理由としては，頻繁な暫定解伝播によりよい暫定値をつねに持っていることも考えられるが，並列度がある程度以上に増えると，暫定解の通信が大量に発生してほとんど探索ができない状況に陥っているからだと考えられる.

実験結果では逐次実行時間が探索ノード数と分岐数 b の積と比例することがわかった (係数 10^8 程度)．逐次ソルバーの探索ノード数を，並列ソルバーの 1 ワーカーが探索した最大ノード数で割ると，通信オーバヘッドを差し引いた速度向上の上限が見積もれると考えられる．しかし実験結果では半数 (インスタンス (b),(e),(f)) が実際よりも低い速度向上となった．理由としては，探索ノード数として b 個の子ノード展開を行う場合，枝刈りによりすぐにノードを捨てる場合，優先度付きスタックの捨てエントリーの場合などが計上されていて精度が悪い点や，並列化によりキャッシュが効いている点などが考えられるが，詳しい調査は今後の課題とする.

インスタンス (f) の実験データ (速度向上，探索ノード数) は各実行結果のばらつきが大きかった．本稿で報告しないものを含め，複数インスタンスで同様のばらつきがあった．実行ごとの枝刈りの発生数に起因すると考えるが，詳しい調査は今後の課題とする.

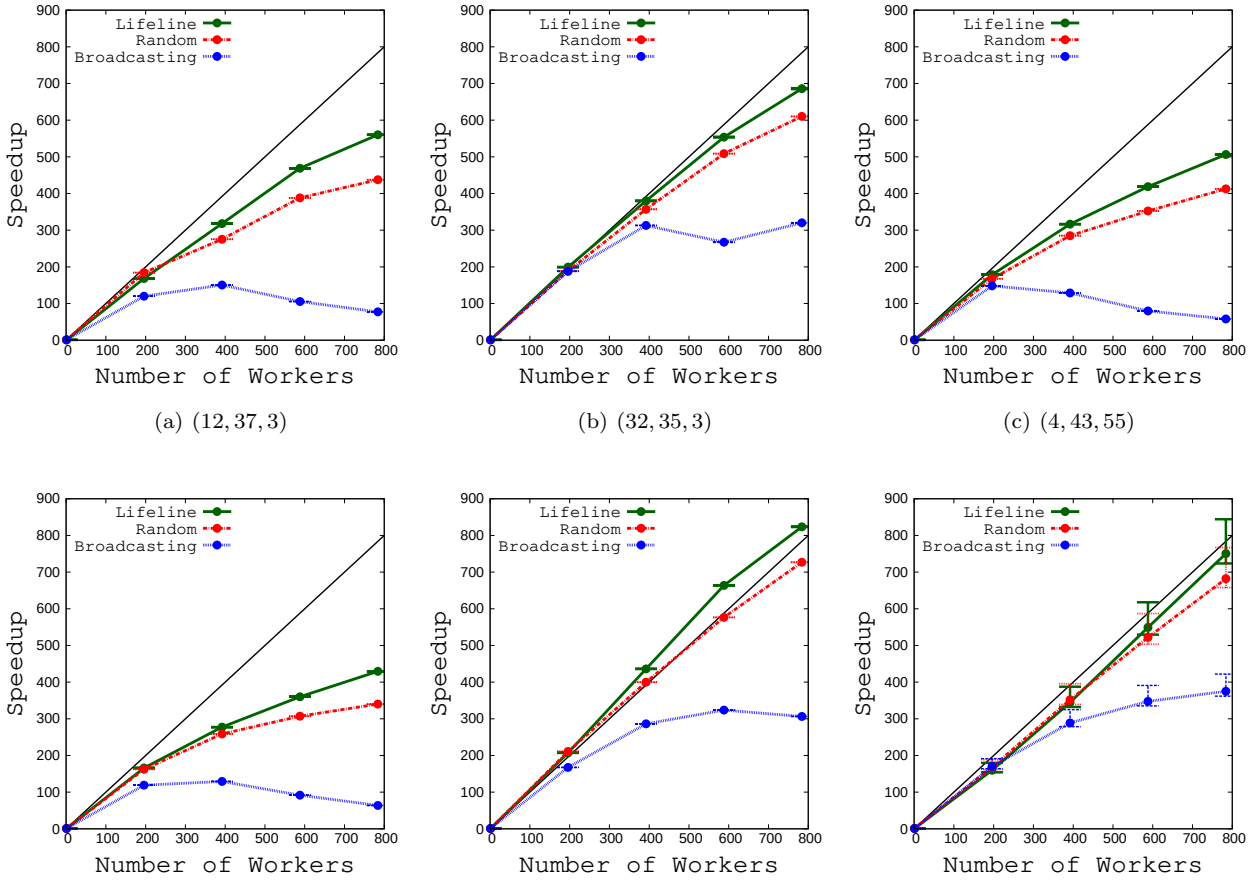
9. まとめ

本論文では，探索に基づく最適化問題の並列求解のために X10 GLB を拡張する手法について述べた．また，提案手法のためのベンチマーク問題について説明した．実験では 784 プレース・コア上で，ワーカー間ネットワーク (ライフライン) を活用した暫定解共有を図り，良好な速度向

[*2] 同じ累積重みが複数出現する場合，2 つ目以降は探索前に刈り取られるため，本ノード数は「必ず探索が必要なノード数」より目減りする.

表 1 実験結果

	(a)	(b)	(c)	(d)	(e)	(f)
(b, d, r)	$(12, 37, 3)$	$(32, 35, 3)$	$(4, 43, 55)$	$(12, 37, 35)$	$(4, 43, 5)$	$(4, 43, 19)$
実実行時間 (逐次)	2810	6900	2170	1720	18950	35700
速度向上 (L)	560	686	506	429	824	751
速度向上 (R)	428	610	412	340	727	682
速度向上 (B)	77	320	58	64	306	375
全ノード数	$9.28 \cdot 10^{39}$	$4.94 \cdot 10^{52}$	$1.03 \cdot 10^{26}$	$9.28 \cdot 10^{39}$	$1.03 \cdot 10^{26}$	$1.03 \cdot 10^{26}$
探索ノード数 (下限)	$2.86 \cdot 10^{10}$	$4.04 \cdot 10^{10}$	$5.81 \cdot 10^{10}$	$2.42 \cdot 10^{10}$	$4.22 \cdot 10^{11}$	$8.35 \cdot 10^{11}$
探索ノード数 (逐次)	$3.95 \cdot 10^{10}$	$4.06 \cdot 10^{10}$	$5.82 \cdot 10^{10}$	$2.42 \cdot 10^{10}$	$5.49 \cdot 10^{11}$	$9.23 \cdot 10^{11}$
探索ノード数 (L)	$3.66 \cdot 10^{10}$	$4.21 \cdot 10^{10}$	$5.88 \cdot 10^{10}$	$2.48 \cdot 10^{10}$	$4.96 \cdot 10^{11}$	$9.98 \cdot 10^{11}$
	$(6.41 \cdot 10^{7})$	$(6.24 \cdot 10^{7})$	$(1.01 \cdot 10^{8})$	$(4.86 \cdot 10^{7})$	$(6.72 \cdot 10^{8})$	$(1.26 \cdot 10^{9})$
探索ノード数 (R)	$3.81 \cdot 10^{10}$	$4.42 \cdot 10^{10}$	$6.05 \cdot 10^{10}$	$2.63 \cdot 10^{10}$	$5.29 \cdot 10^{11}$	$1.07 \cdot 10^{12}$
	$(8.24 \cdot 10^{7})$	$(6.92 \cdot 10^{7})$	$(1.25 \cdot 10^{8})$	$(6.21 \cdot 10^{7})$	$(7.86 \cdot 10^{8})$	$(1.51 \cdot 10^{9})$
探索ノード数 (B)	$9.94 \cdot 10^{10}$	$4.72 \cdot 10^{10}$	$7.73 \cdot 10^{10}$	$3.92 \cdot 10^{10}$	$9.78 \cdot 10^{11}$	$1.72 \cdot 10^{12}$
	$(3.22 \cdot 10^{8})$	$(1.34 \cdot 10^{8})$	$(4.48 \cdot 10^{8})$	$(2.64 \cdot 10^{8})$	$(1.73 \cdot 10^{9})$	$(2.15 \cdot 10^{9})$
暫定解通信回数 (L)	11745 (46980)	4866 (19464)	5247 (20988)	4696 (18784)	15056 (60224)	12858 (51432)
暫定解通信回数 (R)	11524 (34572)	6177 (18531)	5690 (17070)	6606 (19818)	15811 (47433)	12593 (37779)
暫定解通信回数 (B)	166 (129978)	52 (40716)	47 (36801)	122 (95526)	228 (178524)	117 (91611)



(a) $(12, 37, 3)$ (b) $(32, 35, 3)$ (c) $(4, 43, 55)$

(d) $(12, 37, 35)$ (e) $(4, 43, 5)$ (f) $(4, 43, 19)$

図 4 速度向上．節点位置の区間は実行 10 回分のデータを表す．

上 (最大で 824 倍) が得られることを確認した.

今後の課題として，より大規模環境での実験，提案手法の性質の解析，他の最適化問題への応用などに取り組むことが挙げられる.

謝辞 本研究の一部は科研費 18K11240 の補助を得て行った.

参考文献

[1] D. Bergman, A. A. Cire, A. Sabharwal, H. Samulowitz, V. Saraswat, and W.-j. V. Hoeve. Parallel Combinatorial Optimization with Decision Diagrams. In *CPAIOR*, *LNCS* 8451, pages 351–367. Springer, 2014.

[2] B. Bloom, D. Grove, B. Herta, A. Sabharwal, H. Samulowitz, and V. Saraswat. SatX10: A Scalable Plug & Play Parallel SAT Framework. In *SAT*, *LNCS* 7317, pages 463–468, 2012.

[3] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, 2009.

[4] S. Fuller, J. Gaschnik, and J. Gillogly. An analysis of the alpha-beta pruning algorithm. Technical report, Carnegie Mellon University, 1973.

[5] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[6] D. Ishii, K. Yoshizoe, and T. Suzumura. Scalable Parallel Numerical Constraint Solver Using Global Load Balancing. In *ACM SIGPLAN Workshop on X10*, pages 33–38. ACM, 2015.

[7] J. Jaffar, A. Santosa, R. Yap, and K. Zhu. Scalable distributed depth-first search with greedy work stealing. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 98–103. IEEE, 2004.

[8] T. Kaneko. Parallel Depth First Proof Number Search. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 95–100, 2010.

[9] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In *17th European Conference on Machine Learning*, pages 282–293, 2006.

[10] D. Munera, D. Diaz, S. Abreu, and P. Codognet. A Parametric Framework for Cooperative Parallel Local Search. In *14th European Conference on Evolutionary Computation in Combinatorial Optimisation (EvoCOP)*, *LNCS* 8600, pages 13–24, 2014.

[11] L. Otten. AND / OR Branch-and-Bound on a Computational Grid. *Journal of Artificial Intelligence Research*, 59:351–435, 2017.

[12] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *PPoPP*, pages 201–212. ACM Press, 2011.

[13] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa. Scalable distributed monte-carlo tree search. *Fourth Annual Symposium on Combinatorial Search (SOCS)*, pages 180–187, 2011.

[14] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, and V. Saraswat. GLB : Lifeline-based Global Load Balancing Library in X10. In *PPAA*, pages 31–40, 2014.