# 仮想環境を考慮した要求駆動型負荷分散

良本 海　八杉 昌宏　平石 拓　馬谷 誠二

並列言語 Tascell の要求駆動型負荷分散では，各ワーカは論理スレッド等を生成せずに逐次的に計算を行い，他のワーカから要求が来たときにのみ，一時的バックトラックにより最古のタスク生成可能状態を復元してタスクを生成することで，効率の良い並列処理を実現している．この方式では，ワークスティール処理の完了のために，タスクを要求するワーカだけでなく要求されるワーカも CPU 資源の利用をともなう処理を行う必要がある．そのため，仮想環境では，仮想マシンの CPU 利用率を制限すると，制限分以上に性能が低下する恐れがある．本研究では，要求されるワーカが関与せず盗めるタスクを一定数以内で準備しておくようにすることで，仮想環境での並列性能の改善を図る．

## 1　はじめに

マルチコアプロセッサを搭載したコンピュータが広く普及し，それらを用いた並列計算が一般化している．それに伴い，効率と拡張性に優れた並列計算の需要が高まっている．また，計算資源の効率的利用のために，仮想化といった技術も普及しており，仮想環境上での並列処理が行われるようになっている．しかしながら，仮想環境上での並列処理では，物理コア数以上の仮想 CPU が時分割多重で実現されると，並列処理の性能が低下する恐れがある．

並列言語 Cilk とその実装 [1] では，ワーカはそれぞれ多数の論理スレッドを生成し，最古優先のワークスティールにより良好な負荷分散を伴う並列処理を実現している．一方，並列言語 Tascell の実装 [2] で用いられている要求駆動型負荷分散では，通常時，各ワーカは論理スレッドを生成せず逐次的に計算を行う．他のワーカから要求がくると，一時的なバックトラックにより最古のタスク生成可能状態を復元し，タスクを生成し，送信する．これにより，効率の良い並列処理を実現している．

仮想環境上の並列処理では，CPU の物理コア（仮想環境では単に物理 CPU と呼ばれる）数を超えて仮想 CPU を運用する場合がある．そのような場合，物理 CPU を仮想 CPU に対して時分割で割り当てることによって，見かけ上は仮想 CPU の数で動作する．しかし，実際には時分割で割り当てているので，いずれかの仮想 CPU が休止状態となっている時間が存在する．そのような状況における Tascell を用いた並列計算では，休止中の CPU に割り当てられたワーカがタスク要求にすぐに反応できないことにより並列処理の性能が低下する恐れがある．

この問題に対処するため，本研究では Tascell 処理系の改良を行った．具体的には，各ワーカはすぐにスティールできるタスクを一定数用意しておくようにし，スティールされる側の関与なしでスティールできる機会を増やした．また，スティールされる側は，タスクが一定数用意されているかを定期的に確認し，

---

* This is an unrefereed paper. Copyrights belong to the Authors.

Kai Yoshimoto, 九州工業大学大学院情報工学府, Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology.

Masahiro Yasugi, 九州工業大学大学院情報工学研究院, Department of Artificial Intelligence, Kyushu Institute of Technology.

Tasuku Hiraishi, 京都大学学術情報メディアセンター, Academic Center for Computing and Media Studies, Kyoto University.

Seiji Umatani, 京都大学大学院情報学研究科, Graduate School of Informatics, Kyoto University.

```
1  int fib(int n) {
2    if (n <= 2) return 1;
3    else {
4      int s1;
5      int s2;
6      do_two {
7        s1 = fib(n - 1);
8        s2 = fib(n - 2);
9      } handles fib {
10       this.n = n - 2; // タスクの入力を渡す
11       s2 = this.r;    // タスクの結果を受け取る
12     }
13     return s1 + s2;
14 }
```

**図 1　フィボナッチ数を求める Tascell プログラム**

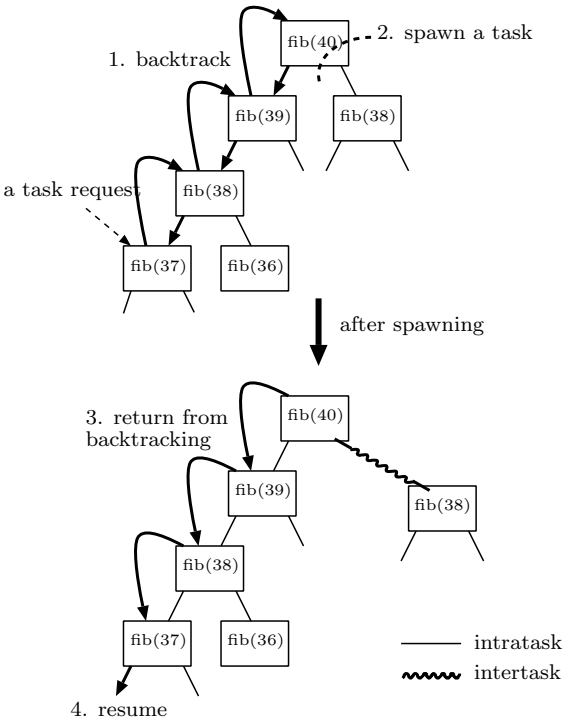


**図 2　Tascell における一時的バックトラックおよびタスク生成**

満たない場合は補給するようにした．

以下，本論文は，第 2 節で，並列言語 Tascell とその従来実装について述べ，第 3 節で，仮想環境上での従来実装の問題点について述べる．第 4 節で，本研究の参考となる既存実装として並列言語 Cilk の実装 [1] とその改良版である Indolent Closure Creation [3] について述べる．第 5 節で，仮想環境を考慮した提案手法を述べ，第 6 節で提案手法に基づく Tascell の新しい実装の詳細を述べる．第 7 節で今後，評価を進めるにあたっての予定について述べる．第 8 節で関連研究について述べ，第 9 節でまとめと今後の課題を述べる．

## 2　並列言語 Tascell

### 2.1　概要

Tascell 言語 [2] は，Cilk 言語 [1] のように，ワークスティールによる並列計算を実現する拡張 C 言語である．図 1 に例として，フィボナッチ数を doubly-recursive アルゴリズムにより並列に計算する Tascell プログラムを示す．プログラム中にタスクを生成可能な箇所（たとえば図 1 の `do_two`）や，タスク（およびその結果）を他ワーカに与える（から受け取る）ための処理（図 1 の `handles fib` ブロック）を記述しておくと，必要なタスク生成やそのワーカへの割り当てが実行時に行われる．

Tascell コンパイラは，Tascell 言語から C 言語への変換器として実装されている．この際バックトラックを最小限のオーバーヘッドで実現するため，文献 [5] [6] で提案されている L-closure や Closure に基づく低コストの入れ子関数を利用している [2]．

### 2.2　Tascell におけるワークスティール

Tascell のワーカは Cilk [1]（4.1 節参照）と異なり，他のワーカからワークスティールの要求（タスク要求）を受け取るまでは，自身に割り当てられたタスクを逐次実行し，タスクの生成は行わない．すなわち，タスク生成が可能な箇所（`do_two` など）に到達してもその場ではタスク生成を行わず，**完全な逐次実行を選択したかのように**計算を続ける．

ワーカ（victim）が計算中に他ワーカ（thief）からタスク要求を受け取ると，タスク生成が可能であった箇所のうち最古のもの，すなわち最大のタスクが生成できると期待できる箇所の実行状態を一時的バック

トラックにより復元し，**過去の選択を変更したかのように**タスクを生成する．victim ワーカはタスクの生成後，一時的バックトラック前の実行状態を再度復元し，自身の計算を再開する．このタスク生成の一連の流れを図 2 に示す．

### 2.3 ワークスティールの従来実装

本節では従来の Tascell のワークスティールの実装について説明する．

各ワーカは，自身が実行すべきタスクの情報を格納する**タスクスタック**と，他ワーカから受け取ったタスク要求情報を格納する**リクエストキュー**，および他ワーカに送信したタスクを管理する**サブタスクスタック**を持つ．

ワーカは，自身のタスクスタックが空ではなく，かつその top にあるタスクが実行可能であればそのタスクを実行する．一方，スタックが空になるか，top にあるタスクが他のタスクの結果の待ち合わせなどにより実行できなくなると，そのワーカは thief として（適切な戦略により）他のワーカから 1 つを victim として選択し [†1]，タスク要求メッセージ（`treq`）を出す．`treq` を受け取ったワーカは，その情報をリクエストキューに追加する．

各ワーカはタスクの実行中，ポーリングによりリクエストキューにタスク要求が届いていないかを確認する．要求が届いていれば，その各アイテムに対して以下の処理を行った後，キューから取り除く．

- 自身がタスク送信可能であるかを確認し，もし可能であれば前述の一時的バックトラック処理によりタスクを生成し，`task` メッセージとして thief に返信する．また，そのタスクに対応するアイテムをサブタスクスタックに push する．
- タスク送信が不可能であれば，その旨を伝えるメッセージ（`none`）を thief に返信する．

`none` を受け取った thief ワーカは，別のワーカを選択し（場合によってはしばらく待機して同じワーカに），`treq` を再送信する．

タスクの実行を完了したワーカは，そのタスクの結果を `rslt` メッセージとして victim ワーカに返信する．`rslt` を受け取ったワーカはサブタスクスタックの対応するアイテムに結果を書き込んでおく．書き込まれた結果の取り込み（図 1 の `s2=this.r` の処理等）やサブタスクスタックの pop は，victim の計算実行がそのタスクを生成した位置（`do_two` など）まで戻ってきたときに処理される．

なお，実装上は，各メッセージの送受信が共有メモリ環境内で行われる場合には，直接的なメッセージのやりとりがワーカ間で行われているわけではなく，送信元ワーカ用スレッドが，受信先ワーカの代理となり，受信先ワーカのデータを直接書き換えることでメッセージ送信処理を（受信先ワーカ用スレッドと相互排他しつつ）実現している．たとえば，`treq` の送信処理では，受信先（victim）ワーカのリクエストキューへのアイテムの追加を，`rslt` の送信処理では受信先（victim）ワーカのサブタスクキューのアイテムへのタスクの結果の書き込みを送信元スレッドが直接行うことで，メッセージ処理を実現している．一方で，これらのメッセージを受けてのタスク生成・送信処理あるいはタスクの結果の取り込み処理については，受信先ワーカ用スレッドが実行する．

上記のように，送信元ワーカ用スレッドが受信先ワーカの代理となって振る舞うときは，送信元ワーカがそのまま動作しているとみなすことも可能であり，受信先ワーカの（相互排他以外の）関与なしに，キューへの追加や結果の書き込みを行っているとみなすこともできる．以後，受信先ワーカの関与という点について考える場合は，送信元ワーカがそのまま動作しているとみなすこととする．

## 3 仮想環境上での並列処理

第 1 節で述べたように，仮想環境上の並列処理では，物理 CPU 数を超えて仮想 CPU を運用する場合がある．例えば，別のホストにマイグレーションする場合や，1 つのホストで複数の仮想マシンを運用するために物理 CPU 数以上の仮想 CPU が必要となる場合などが考えられる．

†1 結果の待ち合わせの際には，その結果を計算するタスクをスティールしたワーカを victim として選択することでスタックサイズを逐次計算の定数倍以下とすることが可能である [4].

物理 CPU を仮想 CPU に対して時分割で割り当てることによって，見かけ上は仮想 CPU の数で動作するが，いずれかの仮想 CPU が休止状態となっている時間が存在する．そのような状況における Tascell を用いた並列計算では，休止中の CPU に割り当てられたワーカがタスク要求にすぐに反応できないことにより並列処理の性能が低下する恐れがある．

第 2 節で述べた Tascell の実装では，`treq` の送信元ワーカがリクエストキューへのアイテムの追加までは，受信先ワーカの関与なしで行うことができるが，タスク生成・送信処理は受信先ワーカが行う必要がある．受信先ワーカは仕事をしながら定期的にリスクエストを確認しているが，休止中はリクエストの確認もできない．

これにより並列処理性能がどの程度低下するかは仮想環境に依存する．特に，送信元となる仮想 CPU で OS レベルのスリープが行われたときに，仮想マシンモニタレベルでも受信先となる仮想 CPU へと物理 CPU が割り当て直されるような仮想環境であれば性能低下は少なくて済む可能性がある．また，その切り替えコストも性能に影響すると考えられる．

また，実際に行われるスティール試行の回数も，どの程度の並列性能低下となるかに関係すると考えられる．ワークスティールは特に並列計算全体の終盤で多く行われるが，この期間の負荷分散がうまくいかないことで全体が終了するまでの時間が延びることも，逆に，仕事をもつワーカがワークスティールされずに終了できることで全体が終了するまでの時間が短縮されることも考えられる．

## 4 既存手法

本節では，本研究の提案手法の参考となる既存実装として並列言語 Cilk の実装 [1] とその改良版である Indolent Closure Creation [3] について述べる．

### 4.1 マルチスレッド言語 Cilk の実装

Cilk 言語 [1] は，C 言語を拡張して作られた並列言語である．Cilk の特徴は，`spawn` と `sync` キーワードを使用して並列化可能な箇所を指定し，実際の並列化は処理系が自動で行う点にある．Cilk による並列プログラムの簡単な例として，フィボナッチ数を求めるプログラムを並列化したものを図 3 に示す．

```
1  cilk int fib(int n) {
2    if(n <= 2) return 1;
3    else {
4      int s1;
5      int s2;
6      s1 = spawn fib(n - 1);
7      s2 = spawn fib(n - 2);
8      sync;
9      return s1 + s2;
10   }
11 }
```

**図 3　フィボナッチ数を求める Cilk プログラム**

Cilk の標準的な実装 [1] では，各ワーカは，固有のランタイムスタック，および，実行可能な手続き（論理スレッド）の両端キュー（ready deque）を持つ．各ワーカは，両端キューの末尾側で要素を push または pop することができ，他のワーカも先頭側（末尾の反対側のもう片方の端）から要素を取り出すことができる．ワークスティールは，アイドル状態となったワーカが thief (盗む側) となり，ランダムに victim (盗まれる側) として選ばれたワーカ（の ready deque の先頭側）から，論理スレッドを盗んで実行する形で行われる．

各 `cilk` 手続きは，fast clone と slow clone の 2 バージョンのコード（手続き）へとコンパイルされる．各ワーカは `spawn` キーワードのついた手続き呼び出しを実行すると，新しい論理スレッドでその手続きを実行する．

新しい論理スレッドで手続きを実行するには，単に fast clone を通常の手続きと同様に呼び出す．fast clone 開始時には，ワークスティールが可能なように，論理スレッドの局所変数の値などを保存するフレームをヒープ割り当てし，意味的には slow clone のコードと合わせてクロージャとして生成する．このクロージャが論理スレッド（の継続）に相当する．

論理スレッド $t_p$（parent）が，新しい論理スレッド $t_c$（child）を spawn するときには，（$t_c$ ではなく）$t_p$ の継続（クロージャ）が，両端キュー（ready deque）

の末尾に push される．これは，ワーカとしては新しい論理スレッド $t_c$ の実行を開始するため，1 つ古い論理スレッド $t_p$ は新たにスティール対象となったと考えればよい．

末尾に push された論理スレッド $t_p$ は，他のワーカから盗まれる対象となる．論理スレッド $t_c$ を実行している間に，$t_p$ が他のワーカによって盗まれることがなかった場合には，$t_c$ のための呼び出しの完了後，$t_p$ の継続を push したワーカによって両端キュー（ready deque）の末尾から $t_p$ の継続は pop される．その後，$t_p$ の実行再開（継続の実行）は pop されたクロージャではなく，fast clone の実行として行う．

子の論理スレッド $t_c$ が終了するまでの間に，親である $t_p$（の継続）が他のワーカによって盗まれていた場合は，$t_c$ のための呼び出しの完了後に，$t_p$ は fast clone として実行は本格的に再開せず，直ぐに中止する．

アイドル状態となったワーカが thief (盗む側) となり，ランダムに victim (盗まれる側) として選ばれたワーカ（の ready deque の先頭側）から，論理スレッド（の継続）をクロージャとして盗んだ場合，その再開はクロージャに含まれる show clone を実行することで行われる．

両端キュー（ready deque）の末尾側からの victim による pop と，両端キュー（ready deque）の先頭側からの thief によるスティールでは，要素を重複して取り出さないようする必要がある．この実現のために，Cilk の標準的な実装 [1] において **THE プロトコル**が提案されている．ここでは，各ワーカの持つ変数 `T`，`H` がそれぞれ deque の末尾（tail）と先頭（head）を表すようにして用いる．また，`E` は例外のために用いるがここでは説明を省略する．victim は `T` を変化させて push や pop を行い，thief は `H` を変化させてスティールを行う．push は `T` を 1 増やして次の場所を使い，pop は `T` を 1 減らす．スティールは `H` を 1 増やし，反対側から要素を取り出す．THE プロトコルでは，この際，取り出した直後に互いの `H` と `T` を読み，`H` > `T` でなければ，取り出しが成功したことがわかるというものである．最初の取り出しの試行が成功しなかったときには，ロックを獲得してからリトライする．

THE プロトコルは一見すると，単に変数を読み書きしているだけだが，読み出しアクセス，書き込みアクセスはそれぞれ不可分である必要がある．加えて，書き込みアクセス後に，`H` > `T` でないことの確認のための読み出しアクセスを行うが，現代的な計算機ではその間に（プログラム通りに書き込み後に読み出しという順序となるように）メモリバリア命令を必要とする．現代的な計算機では，メモリバリア命令は比較的重いため，ある程度のオーバヘッドとなる．Tascell が Cilk よりも高い性能を示すのは，スティールが起きない限り，このオーバヘッドを不要としている点が大きい [2].

### 4.2 Indolent Closure Creation

Indolent Closure Creation [3] は，Cilk 言語の別の実装として位置付けられる．その名の通り，その基本的なアイディアは 4.1 節で述べた Cilk 言語の標準的な実装におけるクロージャ（論理的スレッドの継続）の生成（と両端キューへの push）を本当に必要になるまで遅延させるというものである．

Indolent Closure Creation では，Tascell と同様に victim 側でポーリングを用いる．スティール試行が空の両端キュー（ready deque）に対して発生してスティールに成功しなかったということをポーリングで調べる．成功しなかったスティール試行を検出すると，ランタイムスタックを用いて手続き呼出しとして実行中のすべての論理スレッド（のうちクロージャ未生成のもの）に関してクロージャ生成（と両端キューへの push）を行う．

## 5 提案手法

前述（第 3 節）の問題を回避するため，Tascell におけるワークスティールの新たな手法を提案する．従来の Tascell ではタスクの受渡しの際に，要求元ワーカと要求先ワーカの双方の CPU 資源の利用が必要になる．よって，要求先ワーカが仮想環境の CPU スケジューリングにより停止している場合には，要求元ワーカも動作がストップしてしまうという問題があった．これに対し本研究では，一方の CPU 資源の利用

だけで受渡しのできるタスクを用意することで対処する．具体的には，4.1 節で述べた Cilk 言語の標準的な実装と同様に，各ワーカが両端キューを持ち，そこに要求元ワーカの動作のみで獲得できるタスクを蓄えられるようにする．要求元ワーカが `treq` を代理処理するという見方をすると，両端キューからサブタスクを取り出して自身に `task` メッセージとしてタスク送信するところまで（あるいは `none` メッセージ送信まで）代理で行ってしまうともいえる．

もちろん，タスクを常に生成したのでは，タスク生成にかかるオーバーヘッドを含め，従来の，必要に応じてタスクを生成することによる効率の良さが損なわれてしまう．そこで，各ワーカはすぐにスティールできるタスクを一定数（例えば 3 個）用意しておくようにし，スティールされる側の関与なしでスティールできる機会を増やす．また，スティールされる側は，タスクが一定数用意されているかを定期的に確認し，満たない場合はその分のタスクを生成して補給することとする．

あるワーカが生成済みのタスクの結果の待ち合わせを行う際，従来の Tascell であれば，結果が有れば受け取り，無ければそのワーカは thief となって victim からタスクをスティールして実行しようとする．一方，提案手法では，あるタスクの結果を待ち合わせたい場合，そのタスクが両端キューに残ったまま盗まれていないことがあるので，その場合はそのワーカが自分で pop してこれを実行する．

提案方式は，タスクの生成を必要最小限にしようという考えに近いという点で，4.2 節の Indolent Closure Creation と似ている点もあるが，生成可能なタスクすべてではなく，一定数を目標にタスクを生成して補給する点，空の両端キューとなる前から補給する点（スティール試行の時点で補給が完了していて，スティールが victim ワーカの関与なしに成功すると期待できる点），などが Indolent Closure Creation と異なる．

## 6 仮想環境を考慮した Tascell の実装

本節では，第 5 節で述べた提案手法に基づき，第 2 節で述べた Tascell の従来実装を改良した新しい実装の詳細を述べる．

### 6.1 データ構造

各ワーカに対して，サブタスクを要素する**両端キュー**を配列として追加した．従来実装のサブタスクスタックも残しており，各サブタスクは配列またはサブタスクスタックからアクセスできる．

両端キューの末尾 (tail) と先頭 (head) を表す 0 以上の整数はそれぞれ 16 ビット幅とし，32 ビット符号なし整数の上位 16 ビット（tail），下位 16 ビット（head）として，これを保持するワーカ毎の 32 ビットの変数 `ltq_th`（初期値 0）を用いることとした．こうすることで，compare and swap (CAS) を利用して，両者を一度に，また両者が特定の値のときのみ不可分に更新することができ，両端キューの実装をシンプルにすることができる．Cilk の実装のように，各ワーカに変数 `T` と変数 `H` を持たせて，読み書きとメモリバリアを用いることも考えられるが，仮想環境を考慮した Tascell の実装でも，victim ワーカは変数 `ltq_th` を定期的に読んでタスク補給の判断をする際には CAS は不要であり，また，現代的な計算機ではメモリバリアも CAS とどちらも重いという点で大差はない．

push は head が変わらないまま tail をインクリメント，pop は head が変わらないまま tail をデクリメント，head からのサブタスクのスティールも tail が変わらないまま head のインクリメント，によりで実現できる．pop の際にはサブタスクをフリーリストに返却するようにする．

両端キューを実現するための配列は，同時にスティールされたサブタスクも保持することとした．これは head の値をそのまま用いて，0 以上 head 未満のサブタスクが該当する．この部分は結果待ちに使うことになる．なお，両端キューとしては head 以上 tail 未満である．サブタスクスタックには「両端キューに含まれるスティールされていないサブタスク列」と「スティールされたサブタスク列」が隣接して積まれた形となる．

各サブタスクには，従来のステータスに加えて，拡張ステータスを持たせることにした．ここには，(1)

結果が存在するか，(2) 相互排他を本格的に行うべきか，(3) 結果待ちの間にタスクをスティールしてきて実行しようとしているか，(4) スティールされたのち初期化済みか（タスク返答先など），を表す各 1 ビットのフラグを設けた．特に (1) と (3) を不可分に確認・更新するため，ここでも CAS を用いる．

### 6.2 thief ワーカの動作

タスクの要求元は `treq` メッセージを代理処理することとして，リクエストキューは使わず，victim の両端キューの先頭 (head) を，末尾 (tail) を変えずに 1 増やせるかを CAS により試みる．成功した場合，その位置のサブタスクを初期化し，その後自身に `task` メッセージを返す．タスク本体を管理するタスクを生成してタスクスタックに push して，タスク本体の処理を開始する．(タスク本体の処理中は相対的に victim となることがある．) タスク本体の処理を完了したら，そのタスクの結果を `rslt` メッセージとして victim ワーカに返信する．victim の代理として，該当するサブタスクの拡張ステータスを「結果が存在する」に変更する．結果待ちの間にタスクをスティールしてきて実行しようとしていなかった場合は，`rslt` メッセージに対する ACK メッセージを自身に返す．

### 6.3 victim ワーカの動作

victim ワーカは，自身の両端キューに一定数のタスクがない場合は，従来の実装でリクエストキューの `treq` を検知するときと同じように，これをポーリングで検知し，一時的バックトラックを行ってサブタスクを生成し，両端キューに補給（push）する．また，サブタスクスタックにも push する．

victim ワーカは，thief ワーカのスティールに協力（関与）する必要はない．ただし，上記の補給（push）時には CAS を用いて不可分に head を変えずに tail を増やすようにする．

victim ワーカが過去に生成したサブタスクの結果が必要になるとき，(1) そのサブタスクはスティールされていない．(両端キューからわかる) (2) そのサブタスクはスティールされていて処理されていて結果もある．(3) そのサブタスクはスティールされていて処理されているようであるが結果がない．の 3 通りの場合がある．仮想環境向け実装では，(1) が新たに加わっている．(1) の場合は，両端キューからサブタスクを自身で pop し，自身で実行する．一方，(2) か (3) かの場合，その判断の前にそのサブタスクが初期化されるまで待つ．これは，thief はスティールに成功してからタスク返答先などを書き込んでいるためである．仮想環境向けの実装でありながら，この部分はビジーループで「スティールされたのち初期化済み（タスク返答先など）」フラグが立つのを待っているため，簡単ではないが，今後，ビジーループを使わない形に修正することが望ましい．(2) の場合は，結果を取り出すとともに，スティールされたサブタスクをサブタスクスタックから pop する．また，head と tail を共に 1 減らし，両端キューは空のまま，配列の 0 以上 head 未満のサブタスクを一つ減らす．(3) の場合は，victim ワーカが，結果を待つ間は相対的には thief となり，そのサブタスクをスティールしたワーカが victim となるようなスティールを行って，そのスティールしたタスクを実行する．スティール試行中に，結果（`rslt` メッセージ）を受け取った場合は，スティール失敗後に `rslt` メッセージに対する ACK メッセージを自身に返す．

## 7 性能評価に向けて

仮想環境における並列処理の性能評価については高山，光来らの研究 [8] がある．仮想環境の設定として，(1)VM 間での物理 CPU の共有，(2)VM が利用できる CPU 使用率を制限，(3)VM への物理 CPU 割り当てを削減，を試し，特に (2) で Tascell でも並列性能の顕著な低下が報告されている．仮に，本研究で課題とした，時分割により休止状態の仮想 CPU があることにその原因があれば，提案実装の効果が現れる可能性があると期待される．

また，両端キューを最初から用いている Cilk との性能比較も進めるのが望ましい．

## 8 関連研究

マルチスレッド言語 OPA とその実装 [7] も，Indolent Closure Creation と同様にポーリングを用い

ている．OPA の実装では，victim がスティールの要求をポーリングで検出して，両端キューにスレッドがあれば，thief へ送り出す．また，両端キューにスレッドなければ，高速に動作するためにランタイムスタックに確保していたフレームを，ヒープに確保しなおしたフレームに変換し，スティール可能とする．また，ヒープに確保したフレームにを用いて実行を再開するときは 1 フレームずつ行うため，スタック全体を再構築する Indolent Closure Creation よりも素早く再開できる．OPA の実装では，全体的にポーリングを用いているため，Tascell と同様に要求元ワーカが，要求先ワーカに待たされる恐れがある．この問題に対処するには，thief が直接，victim のスレッドをスティールできればよく，これは既存の両端キューを用いて比較的容易に対応可能と考えられる．

## 9 まとめ

本研究では，並列言語 Tascell の要求駆動型負荷分散では，ワークスティール処理の完了のために，タスクを要求するワーカだけでなく要求されるワーカも CPU 資源の利用をともなう処理を行う必要がある点に着目した．仮想環境では，仮想マシンの CPU 利用率を制限すると，制限分以上に性能が低下する恐れがある．

本研究では，仮想環境向けの Tascell の実装として，要求されるワーカが関与せず盗めるタスクを一定数以内で準備しておくようにする方法を提案した．

今後は，評価を行い，提案手法の有効性を確かめていきたい．

**謝辞** 本研究の一部は，JSPS 科研費 JP17K00099, JP26280023 の助成を受けて行ったものです．

## 参 考 文 献

[1] Frigo, M., Leiserson, C. E., and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices (PLDI '98)*, Vol. 33, No. 5(1998), pp. 212–223.

[2] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T.: Backtracking-based Load Balancing, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*, February 2009, pp. 55–64.

[3] Strumpen, V.: Indolent Closure Creation, Technical Report MIT-LCS-TM-580, MIT, June 1998.

[4] Wagner, D. B. and Calder, B. G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proceedings of Principles and Practice of Parallel Programming (PPoPP'93)*, 1993, pp. 208–217.

[5] Yasugi, M., Hiraishi, T., and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proceedings of the 15th International Conference on Compiler Construction (CC2006)*, Lecture Notes in Computer Science, No. 3923, Springer-Verlag, March 2006, pp. 170–184.

[6] 八杉昌宏, 平石拓, 篠原丈成, 湯淺太一: L-Closure：高性能・高信頼プログラミング言語の実装向け言語機構, 情報処理学会論文誌：プログラミング, Vol. 49, No. SIG 1 (PRO 35)(2008), pp. 63–83.

[7] 馬谷誠二, 八杉昌宏, 小宮常康, 湯淺太一: オブジェクト指向並列言語 OPA のための遅延正規化手法, 情報処理学会論文誌：プログラミング, Vol. 45, No. SIG 5 (PRO 21)(2004), pp. 12–25.

[8] 高山都旬子, 光来健一: VM が利用可能な CPU 数の変化に対応した並列アプリケーション実行の最適化, 日本ソフトウェア科学会第 *33* 回大会講演論文集, September 2016.