分散メモリ環境における階層性と仕事量を考慮したワークスティール戦略

中嶋 隆介　八杉 昌宏　平石 拓　馬谷 誠二

我々はワークスティールに基づく動的負荷分散を提供する並列言語 Tascell を提案している．Tascell では小さいタスクの分割を防ぐために，ワーカが持つ仕事の大きさを実数値で見積もり，その値を優先度あるいは重みとして用いるスティール戦略が採用されている．本研究では分散メモリ環境における既知の問題点と既存の対応を踏まえて，ノード内に分割可能なタスクが残っている場合にも定期的にノード外へタスク要求を行う方式を Tascell に実装した．また，ワーカがサブタスクの結果待ち状態になった際のタスクの取り返しによる小さいタスクの授受を抑制するために，一定回数までは取り返しではないタスク要求を許す方式も実装した．

1　はじめに

ワークスティールは並列化に用いる手法の 1 つで，theif ワーカが victim ワーカの仕事を盗むことで効率的な動的負荷分散を提供する．ワークスティールに基づく動的負荷分散を実現している並列言語として，Cilk [4]，Tascell [5, 9] などがある．Cilk では，多数の論理スレッドを生成して最古優先のワークスティールを採用することで全てのワーカを有効に活用する．

一方，論理スレッドフリーな並列言語 Tascell では，ワーカはタスク要求を受け取るまで基本的には逐次計算を行う．アイドル状態のワーカからの要求時にのみ，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，タスクを生成する．この手法は論理スレッドの生成や管理に伴うコストの削減，作業空間の再利用の促進，参照局所性の改善といった利点に加え，作業空間の複製を必要になるまで遅延可能という特徴を持つ．

Tascell プログラムは起動時に接続先の Tascell サーバを指定することで，サーバに接続した複数の計算ノードを用いた分散メモリ環境での並列計算を行うこともできる．

Tascell では小さいタスクの分割を防ぐために，優先度選択ならびに重み付選択 [16] というスティール戦略が提案されている．これらの戦略ではワーカが持つ仕事の大きさを実数値で見積もり，その値を優先度あるいは重みとして用いることで，より大きな仕事を持つワーカがタスク要求先として選択されることが期待できる．本研究では，Tascell のワークスティール戦略の改善を行った．

従来の Tascell では，分散メモリ環境上でタスク要求を行う際に，まず最初に同一計算ノード内の他ワーカに対してタスクを要求し，ノード内のどのワーカもタスク要求を受け付けないときのみノード内の代表ワーカから外部ノードへタスク要求するスティール

---

* Work-Stealing Strategies for Distributed Memory Environments That Consider Hierarchy and Work Amount.

This is an unrefereed paper. Copyrights belong to the Author(s).

Ryusuke Nakashima, 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, Dept. of Computer Science and Networks, 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, 神奈川大学理学部情報科学科, Dept. of Information Sciences, Faculty of Science, Kanagawa University.

戦略を採用している．このスティール戦略ではノード間のタスク授受の回数を減らすことができる一方で，外部ノードに大きいタスクがある場合でもノード内のワーカへタスク要求を行うため，ノード内でタスクが細断される恐れがあるという問題があった．

また，Tascell では，ワーカがタスクの結果待ち状態になった際のタスク要求先を，結果待ちの原因となっているタスクを実行しているワーカに限定する（タスクの取り返し）という戦略も採用している．このスティール戦略にはワーカの実行スタックの肥大を抑える効果がある反面，当該ワーカ間で取り返し合いが発生し，複数のワーカ間で小さいタスクが授受されるという問題が生じる可能性がある．

これらの問題に対する解決策として，ノード内に分割可能なタスクが存在する場合にも定期的に外部ノードへタスク要求させるスティール戦略を Tascell に実装した．さらに，取り返しにより生じる問題を緩和するために，結果待ち状態の場合に一定回数までは取り返しではないタスク要求を許すという取り返し制約緩和を適用したスティール戦略も実装した．

本論文の構成は以下の通りである． 2 章では並列言語 Tascell の概要を述べる． 3 章では従来の Tascell で採用しているワークスティール戦略について説明する． 4 章では従来の問題点と提案するスティール戦略について説明する． 5 章では性能評価の結果を示す． 6 章で関連研究を紹介し，最後に 7 章で本論文のまとめと今後の課題を述べる．

2 並列言語 Tascell

本章では並列言語 Tasacell とその実装に関して [5,18] に基づき説明する．

2.1 概要

並列言語 Tascell はワークスティールに基づく動的負荷分散のための拡張 C 言語である．

Tascell では一時的なバックトラックに基づくタスクの遅延分割を行う．計算中の Tascell ワーカはタスクの分割が可能な点に到達しても，最初は「タスクを生成しないこと」を選択し逐次的に計算を行う．他のワーカからタスク要求を受信すると，

```
int a[12];    // manage unused pieces
int b[70];    // the board, with (6+sentinel) × 10 cells
// Try from the j0-th piece to the 12th piece in a[].
// The i-th piece for i<j0 is already used.
// b[k] is the first empty cell in the board.
int search (int k, int j0)
{
 int s=0; // the number of solutions
 // iterate through unused pieces
 for (int p=j0; p<12; p++) {
   int ap=a[p];
   for (each possible direction d of the piece) {
     ... local variable definitions here ...
     if (Can the ap-th piece in the d-th direction be placed on
           the board b?);
     else continue;
     Set the ap-th piece onto the board b and update a.
     kk = the next empty cell;
     if (no empty cell?) s++;          // a solution found
     else s += search (kk, j0+1);  // try the next piece
     Backtrack, i.e., remove the ap-th piece from b and restore a.
   }
 }
 return s;
}
```

図 1　バックトラック探索により Pentomino パズルの全解探索をするプログラム (C 言語による疑似コード)

(1) 一時的なバックトラックを行って過去の計算状態を復元し，
(2) その時点でタスク要求があったかのようにタスクを生成し，
(3) バックトラックからもとの計算状態に戻り，
(4) 自身が行っていた計算を再開する．

このように最古のタスク生成可能状態までバックトラックすることでより大きなタスクを生成しやすくなる．また，各論理スレッドが固有の作業空間を必要とする Cilk とは異なり，Tascell では逐次計算を行っている間は 1 つの作業空間を再利用できる．Tascell はタスク要求されたときのみにタスクを生成するので，作業空間のコピーも実際に必要になったときだけ行われる．

2.2 Tascell 言語

図 1 は Pentomino パズルの全ての解を見つけるバックトラック探索を行う C プログラムであり，図 2 はそれを並列化した Tascell プログラムである．

この Tascell プログラムでは，`pentomino` というタスクオブジェクトの構造を定義している．フィールド `k`，`i0`，`i1`，`i2`，`a`，`b` はタスクの入力を，フィールド `s`

```
task pentomino {
  out: int s;    // output
  in: int k, i0, i1, i2;
  in: int a[12]; // manage unused pieces
  in: int b[70]; // the board, with (6+sentinel) × 10 cells
};
task_exec pentomino {
  this.s = search (this.k, this.i0, this.i1, this.i2, &this);
}
worker int search (int k, int j0, int j1, int j2, task pentomino *tsk)
{
 int s=0; // the number of solutions
 // parallel for construct in Tascell
 for (int p : j1, j2)
  {
   int ap=tsk->a[p];
   for (each possible direction d of the piece) {
    ... local variable definitions here ...
    if(Can the ap-th piece in the d-th direction be placed on the board tsk->b?);
    else continue;
    dynamic_wind // construct for specifying undo/redo operations
     { // do/redo operation for dynamic_wind
       Set the ap-th piece onto the board tsk->b and update tsk->a.
     }
     { // body for dynamic_wind
       kk = the next empty cell;
       if (no empty cell?) s++; // a solution found
       else                     // try the next piece
        s += search (kk, j0+1, j0+1, 12, tsk);
     }
     { // undo operation for dynamic_wind
       Backtrack, i.e., remove the ap-th piece from tsk->b and restore tsk->a.
     } // end of dynamic_wind
   }
  }
 handles pentomino (int i1, int i2)
     // Declaration of this and setting a range (i1-i2) is done implicitly
  {
   // put part (performed before sending a task)
   { // put task inputs for upper half iterations
     copy_piece_info (this.a, tsk->a);
     copy_board (this.b, tsk->b);
     this.k=k; this.i0=j0; this.i1=i1; this.i2=i2;
   }
   // get part (performed after receiving the result)
   { s += this.s; }
  }  // end of parallel for
 return s;
}
```

図 2　Pentomino パズルのバックトラック探索を行う Tascell プログラム

は結果を格納するために用意されている．pentomino タスクを受け取った Tascell ワーカは pentomino の task_exec 部を実行する．

task_exec を実行中のワーカは，this キーワードによって受け取ったタスクオブジェクトを参照することができる．例えば，図 2 の task_exec 内ではタスクオブジェクトの入力フィールドの値を引数としてワーカ関数 search を呼び出している．

Tascell の並列構文を利用できるのは，キーワード worker によって属性付けられた関数内に限られる．search 関数では Tascell の並列 for 構文を用いて反復計算を分割する．その構文は次の通りである．

for (int *identifier* : $expr_{from}$, $expr_{to}$) $stat_{body}$
handles *task-name* (int $identifier_{from}$,
int $identifier_{to}$)
{ $stat_{put}$ $stat_{get}$ }

タスク要求ハンドラ（$stat_{body}$ の実行中に有効）が呼び出されると，未処理の反復の半分にあたる計算が新たな *task-name* タスクとして生成され，そのタスクオブジェクトは $stat_{put}$ で初期化される．$stat_{put}$

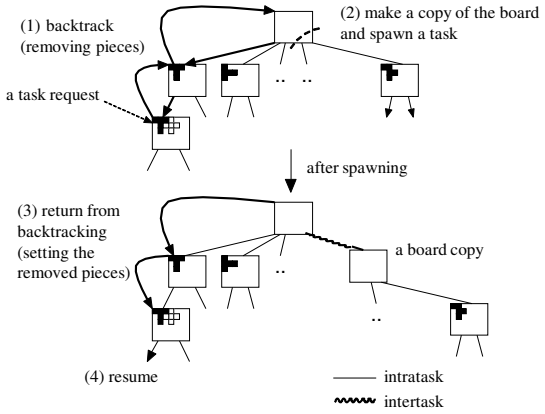

図 3　一時的バックトラックによるタスク生成の流れ．(1) undo しつつバックトラック (すなわち，ピース除去)．(2) 反復の半分をタスクとして生成 (一時的に過去の状態に戻った盤面をコピー)．(3) redo しつつバックトラックから復帰 (すなわち，ピース再設置)．

では，実際に割り当てられた範囲を $identifier_{from}$ と $identifier_{to}$ によって参照することができる．ワーカは $stat_{get}$ の実行によって生成されたタスクの結果を統合（マージ）する．ワーカは，タスク要求されない限り並列 `for` 文を逐次的に実行するため，ローカル配列などの 1 つの作業空間を再利用し続けることができる．

並列 `for` 文はそれぞれの $stat_{body}$ で動的に入れ子にすることもできる．これにより，複数のタスク要求ハンドラを同時に有効にすることができる．ワーカは各並列 `for` 文のエントリポイントでポーリングによるタスク要求の検出を試みる．タスク要求を検出すると，一時的なバックトラックを行い，できる限り古いハンドラを呼び出すことでより大きなタスクを生成する．

図 2 の Tascell プログラムにおいて，ワーカがタスク要求を受理したときの一時的バックトラック，タスク生成，および一時的バックトラックからの復帰の流れを図 3 に示す．ここで，“バックトラック” と，“バックトラックからの復帰” が，ワーカが終えた計算全体を undo/redo 操作することを意味するのではなく，最小限の undo/redo 操作によって最古のタスク生成可能状態の復元と，その状態からの復帰を意味するということに注意されたい．Pentomino におけるピースの除去/設置のような，アプリケーション依存の undo/redo 操作は，`dynamic_wind` 構文によって指定する．

2.3　Tascell フレームワーク

Tascell フレームワークは Tascell サーバおよび Tascell 言語のコンパイラにより構成されるフレームワークである．Tascell コンパイラは Tascell 言語から C 言語，あるいは拡張 C 言語である XC-cube 言語 [12, 17] への変換器として実装されている．

コンパイルされた Tascell プログラムは 1 台以上の計算ノードで実行される．各計算ノードでは 1 つ以上のワーカによる共有メモリ環境での並列計算が行われる．

Tascell サーバは計算ノード間のメッセージの中継，ユーザインターフェースとの入出力処理，各計算ノードの負荷情報の管理等を行う．Tascell サーバに複数の計算ノードを接続することにより，分散メモリ環境での並列計算も実行できる．また，Tascell サーバには計算ノードだけでなく別の Tascell サーバも接続することができる．

2.4　ワーカが保持する内部データ

Tascell ワーカは以下のスタックおよびキューを保持する．

task スタック　そのワーカ自身が保持するタスクのリスト．各エントリは allocated（タスクオブジェクト受信待ち），initialized（タスクオブジェクト受信済み），running（実行中），suspended（サブタスクの結果待ち），done（計算終了）などの状態をとる．

reqest キュー　他のワーカからのタスク要求を受理し，タスク生成待ちになっているエントリのキュー．タスクが生成されると，対応するエントリが次に述べる subtask スタックに移動する．

subtask スタック　他のワーカに送信したサブタスクのリスト．各エントリは allocated（request キューでタスク生成待ち），initialized（タスク

生成・送信済み），done（タスクの結果を受信済み）の状態をとる．

task スタック，subtask スタックの各エントリに対して，1 つのタスクオブジェクトへの参照が対応付けされている．サブタスクに関しては，生成側の subtask スタックと計算側の task スタック内のエントリが同一のタスクオブジェクトを参照する必要があり，分散メモリ環境ではそれぞれのノードで別の実体とするが，必要なデータはノード間通信で一致させる．

2.5 ワーカ間のメッセージ

ワーカ間でのタスク要求やタスク送信などはメッセージを用いて行われる．Tascell ではノード間・ノード内に関わらず共通のメッセージが用いられる．以下に主なメッセージを示す．

treq メッセージ (task request) 送信先のワーカにタスクを要求するメッセージ．メッセージの書式は以下の通りである．

`treq` ⟨送信元アドレス⟩ ⟨送信先アドレス⟩

タスクの要求先がどのワーカでもよい場合，送信先アドレスを明示的に指定するかわりに `any` とすることもできる（any 要求）．

task メッセージ 他のワーカにタスクを送信するメッセージ．メッセージの書式は以下の通りである．

`task` ⟨分割回数⟩ ⟨送信元アドレス⟩:⟨タスク ID⟩ ⟨送信先アドレス⟩ ⟨タスク種別⟩ ⟨データ⟩

⟨分割回数⟩ は，そのタスクが最初のタスクから何回分割されたサブタスクであるかを示す自然数で，タスクの大きさの目安になる．この数はメッセージを中継する Tascell サーバがどの計算ノードに大きなタスクがあるかを見積もるために利用することを目的としており，必ずしも必要なものではない．

⟨タスク ID⟩ は，ワーカが生成したサブタスクに対して割り当てるユニークな整数である．⟨タスク種別⟩ はどのタスク型に対応するタスクかを示す番号である．⟨データ⟩ はそのタスクの入力である．

none メッセージ `treq` メッセージを受け取ったワーカがタスクを生成できる状態にない場合，そのタスク要求を拒否するために送信するメッセージである．メッセージの書式は以下の通りである．

`none` ⟨送信先アドレス⟩

rslt メッセージ (result) タスクの計算結果を分割元のワーカへ送信するためのメッセージである．メッセージの書式は以下の通りである．

`rslt` ⟨送信先アドレス⟩:⟨タスク ID⟩ ⟨データ⟩

⟨タスク ID⟩ には `task` メッセージを受け取った時の整数をそのまま指定する．⟨データ⟩ はそのタスクの出力（結果）である．

rack メッセージ (result ack) `rslt` メッセージを受け取ったワーカがその送信元に送り返す確認メッセージである．メッセージの書式は以下の通りである．

`rack` ⟨送信先アドレス⟩

ワーカは，送信した `rslt` メッセージのうち，対応する `rack` メッセージを受け取っていないものが 1 つでもある場合，タスクの分割を行わない（全ての `treq` メッセージに対して `none` メッセージを返す）．この仕組みは取り返しの対象でない仕事に由来するタスクの生成を防ぐためのものである．

3 従来のワークスティール戦略

本章では従来の Tasacell で採用されているワークスティール戦略に関して [9, 16] に基づき説明する．

3.1 タスク要求の処理

従来の Tascell では，タスクスタックが空になったワーカ（thief）は以下の戦略でタスク要求先（victim）を決定する．

(1) 同一計算ノード内の別のワーカを victim として選択し，タスク要求メッセージを送信する．

(2) ノード内にタスク生成が可能なワーカがいない場合，ノード内の代表ワーカが，接続している Tascell サーバにタスク要求（`treq`）メッセージを送信する．

(3) Tascell サーバは接続されているノードの中か

ら 1 つのノードをランダムに選択し，treq メッセージを転送する．このとき，

- treq メッセージの送信元のノード
- task メッセージを送った回数と rslt メッセージを受け取った回数が同数のノード（タスクが存在しないノード）

は転送先ノードとして選択しない．

(4) 転送された treq メッセージを受け取った計算ノードでは適当なワーカ 1 つを victim として選択し，そのメッセージを送信する．タスク生成可能なワーカがいない場合は，none メッセージが Tascell サーバを再び介して thief ワーカに返される．

3.2 一様ランダム選択

初期の Tascell [5] では，3.1 節の手順 (1) および (4) においてノード内で victim を選択する際ワーカの持つタスクの大きさなどの情報によらず無作為に victim を選択する「一様ランダム選択」方式を採用していた．一様ランダム選択方式では，thief は以下の手順でノード内から victim を決定する．

(1) thief はランダムに選んだ victim に対して treq メッセージを送信する．

(2) タスク要求を受け取った victim は，タスク生成が可能な場合，タスクを生成し thief へ送信する．タスクを持っていないなどの理由によりタスク生成ができない場合は拒否（none）メッセージを thief へ送信する．

(3) thief は none メッセージを受け取った場合には，手順 (1) に戻り残りのワーカへのタスク要求を試みる．（本研究で実際に用いている実装では，タスク要求が victim の request キューに受理されない場合，thief は，ノード内の別のワーカらを一周 victim として選んでいく形でタスク要求の転送を試みる．結局受理されない場合に none メッセージを生成する．ノード外から treq メッセージを受け取った場合もメッセージ処理スレッドが thief の代理として同様に処理する．）

一様ランダム選択では無作為に victim ワーカを選択するため，小さいタスクを持つワーカから頻繁にスティールしてしまう恐れがある．

3.3 優先度選択

大きなタスクを持つワーカを要求先として選択する戦略として優先度選択 [9,16] が提案されている．優先度選択はタスクの大きさを実数値で見積もり，その実数値をワーカ固有の変数に設定して，ワークスティール時に優先度として利用する戦略である．優先度選択によるワークスティールは次のように動作する．

(1) thief はノード内の全 victim 候補の中から κ ワーカをランダムに選択する．

(2) その κ ワーカの各々に設定された優先度を読み取り，最も高い優先度を持つワーカに対してワークスティールを行う．このとき victim がタスク生成が可能であればスティールは成功する．

(3) thief は none メッセージを受け取った場合には，ノード内の別のワーカらを一周 victim として選んでいく形でタスク要求の転送を試みる．

優先度を参照する victim 候補の数は $1 \leq \kappa \leq n$（n は victim 候補の数）の範囲の中から選択する．$\kappa = n$ とすると，常に最高の優先度を持つワーカが victim として選択されるようになり，κ を小さくすると，優先度を考慮しつつランダム性も持った victim の選択を行うようになる．

3.4 タスクの取り返し

Tascell ワーカは，別のワーカに送信したタスクの結果を受け取らないと，実行中のタスクを進められなくなった場合（結果待ち状態）にも，タスク要求を行う．結果待ち状態でのタスク要求では，その結果待ちの原因となっているタスクを送信したワーカをタスク要求先に指定し treq メッセージを送信する．これをタスクの取り返しという．

結果待ちの際のタスク要求を取り返しに限定することで，タスクスタックの最大サイズ，実行スタックの最大サイズそれぞれについて，「単一ワーカタスク生成型実行」の場合の定数倍以下と保証できる．ここでいう単一ワーカタスク生成型実行では，1 ワーカのみで，スティール可能なサブタスクを（Cilk などと同様に，両端キューに）spawn しておく（そのため，

並列 for 構文の反復を分割統治で実行する）ことに加えて，自ら実行する場合にもタスクスタック上で実行するものとする．

4 ワークスティール戦略の改善

4.1 外部ノードへのタスク要求

3.1 節で述べた通り，従来の Tascell では同一ノード内に分割可能なタスクがない場合にのみ，ノード内代表ワーカから外部ノードへのタスク要求を行うという戦略を採用している．この戦略は，ノード間のタスク授受の回数を減らすために有効であると考えられる．しかし一方で，ノード内に小さなタスクしか存在せず，外部ノードに大きなタスクが存在する場合に，ノード内でタスクが細断され小さなタスクが授受される恐れがあるという問題がある．

本研究では，この問題の解決策として，ノード内への any 要求の代わりに外部ノードへ any 要求をどのワーカも定期的に行う戦略を実装した．具体的には，ワーカ固有のカウンタで any 要求を開始した回数をカウントして，カウンタの値が E に達したら，内部ノードへの any 要求の代わりに Tascell サーバを介して外部ノードに any 要求を行い，カウンタの値を 0 に戻すという方式を実装した．$E = 0$ とすると従来実装と同じ戦略になる．

4.2 取り返し制約の緩和

3.4 節で説明した結果待ち状態での取り返し制約により，Tascell ワーカのタスクスタックのサイズの上限は保証されている．しかし一方で，取り返し先のワーカが小さなタスクしか生成できなくても，別のワーカにタスク要求を出すことができないという問題がある．また，取り返し先のワーカが別のワーカへの取り返しを行うことで複数のワーカの間で小さなタスクが授受されるという問題も生じうる．

本研究では，ワーカが取り返し要求を行うべき状況においても一定回数までは any 要求によるスティールを許容することにより，この問題の解決を試みた．具体的には，各ワーカが取り返しの代わりに any 要求を開始した回数をカウントするためのカウンタを用意し，結果待ち状態になった際にそのカウンタの値が τ 未満であれば取り返しの代わりに any 要求によるスティールを開始する方式を実装した．より具体的には，ワーカは以下のように動作する．

(1) ワーカは結果待ち状態になった際，カウンタの値 c_{t} が τ 未満であれば，カウンタをインクリメントし，取り返しの代わりに any 要求を送信する．τ であれば，従来実装における挙動と同様に，取り返しによるタスク要求を行う．結果待ちが解消されたときには，カウンタの値を結果待ち状態になったときの値に戻す．

(2) 取り返しの代わりの any 要求を受け取ったワーカは，タスク生成可能であればタスクを生成し，自身のカウンタの値 c_{v} と共に要求元へ送信する．カウンタ値は `task` メッセージの ⟨分割回数⟩ 部分に付加するようにした．

(3) 取り返しの代わりの any 要求によりタスクを獲得したワーカは，自身のカウンタ値 c_{t} とタスクとともに受け取ったカウンタ値 c_{v} を比較し，$c_{\mathrm{t}} < c_{\mathrm{v}}$ であれば，カウンタを c_{v} に更新する．

表 1 評価環境

	Xeon Phi
Host processor(s)	Intel Xeon E5-2697 v2 12-core × 2
Host memory	64GB (shared)
Co-processor(s)	Intel Xeon Phi 3120P 57-core × 4 (four hardware threads per core)
Co-processor Memory	6GB (for each co-processor)
Network	Each co-processor is connected to the host via PCIe 3.0×16 (Bandwidth = 15.6 GB/s)
OS	CentOS 6.5 (64bit)
Compiler	Intel Compiler 13.1.3 with -O3 optimizers
Closure	Trampoline-based implementation (compatible with the GCC extension [1])
Tascell server	Steel Bank Common Lisp 1.2.7 (runs on the host processors)

5 性能評価

本章では提案手法の分散メモリ環境における性能評価について述べる．評価環境は表 1 の通りである．

性能評価では，優先度選択と 4 章で説明したスティール戦略を組み合わせて測定を行った．また比較のために，一様ランダム選択，優先度選択に関しても測定を行った．

評価には次のベンチマークプログラムを用いた．

- **Fib(n)** n 番目のフィボナッチ数を再帰的に求めるプログラム．優先度 w は $w = n$ で与えられる．

- **Nq(n)** n 女王問題の全解探索をバックトラックを用いて行うプログラム．優先度 w は $w = n - (j + 1)$ で与えられる．ここで j は設置済みの女王駒の数である．
- **Pen(n)** n ピースを用いたペントミノパズル（$n > 12$ は追加のピースと拡張ボードを使用する）の全解探索をバックトラックを用いて行うプログラム．優先度 w は $w = n - j$ で与えられる．ここで j は設置済みのピースの数である．
- **Cmp(n)** 2 つの n 要素の配列間の全要素のペア（$0 \leq i, j < n$ として，(a_i, b_j)）について比較演算を行うプログラム．優先度 w は $w = n$ で与えられる．
- **Histogram($n, \boldsymbol{N_R}, \boldsymbol{d}$)** 2 以上 $2 + n$ 未満の d 個の自然数からなる n^d 組の最大公約数のマルチセット
  $\{gcd(i_1, i_2, \ldots, i_d) \mid 2 \leq i_1, i_2, \ldots, i_d < 2 + n\}$
  をヒストグラムとして集計するプログラム．優先度 w は $w = nd$ で与えられる．

プログラムの入力については，Fib(50)，Nq(17)，Pen(15)，Cmp(150000)，Histogram(50,50,7) とした．

評価結果を図 4 から図 8 に示す．グラフには 22 回測定したときの第一四分位数と第三四分位数をエラーバーとして表示している．グラフの縦軸は逐次 C プログラムに対する性能比であり，t_S/t_P で計算される．（t_S は逐次 C プログラムの実行時間，t_P は P ワーカ使用したときの Tascell プログラムの実行時間.）

グラフ中の凡例の詳細については以下の通り．

- **random** は一様ランダム選択を用いたときの速度向上を示す．
- **priority-all** は優先度選択（$\kappa = P$）を用いたときの速度向上を示す．全 victim の中から最大の優先度を持つワーカへスティールを行う方式である．
- **remote-10** は優先度選択（$\kappa = P$）で $E = 10$ としたときの速度向上を示す．ただし，ノード数 1 のときは $E = 0$ とした．
- **remote-20** は優先度選択（$\kappa = P$）で $E = 20$ としたときの速度向上を示す．ただし，ノード数

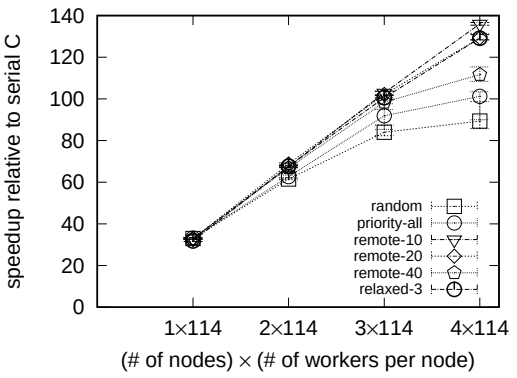

図 4　Fib(50) の分散メモリ環境における速度向上 (逐次 C プログラムに対する性能比)

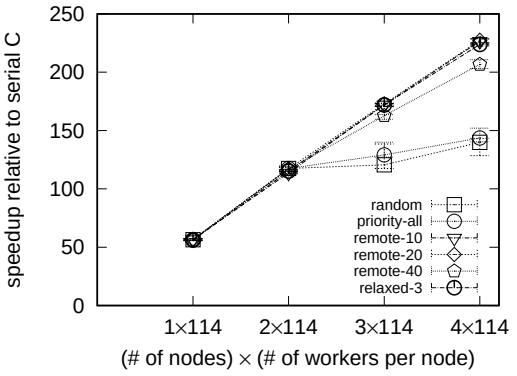

図 5　Nq(17) の分散メモリ環境における速度向上 (逐次 C プログラムに対する性能比)

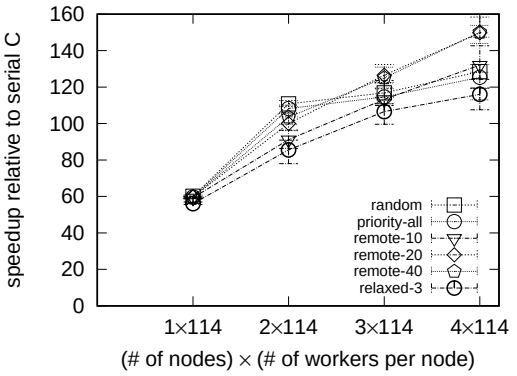

図 6　Pen(15) の分散メモリ環境における速度向上 (逐次 C プログラムに対する性能比)

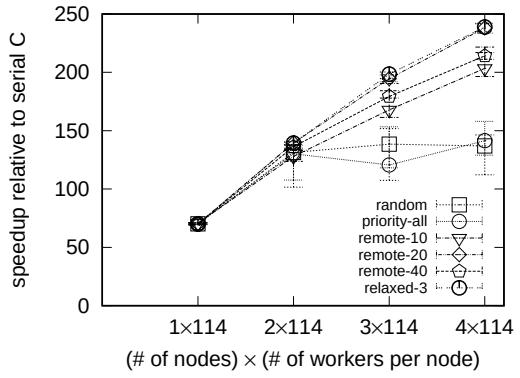

図 7　Cmp(150000) の分散メモリ環境における速度向上 (逐次 C プログラムに対する性能比)

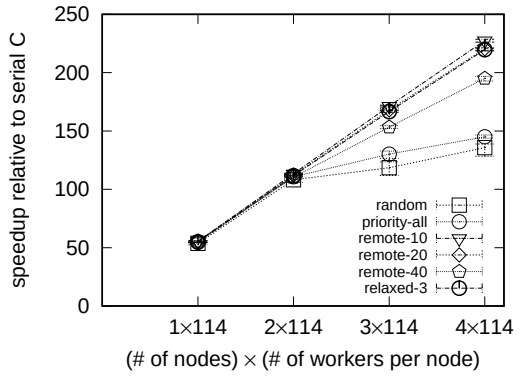

図 8　Histogram(50,50,7) の分散メモリ環境における速度向上 (逐次 C プログラムに対する性能比)

1 のときは $E = 0$ とした．

- **remote-40** は優先度選択（$\kappa = P$）で $E = 40$ としたときの速度向上を示す．ただし，ノード数 1 のときは $E = 0$ とした．
- **relaxed-3** は優先度選択（$\kappa = P$）で $E = 20$, $\tau = 3$ としたときの速度向上を示す．ただし，ノード数 1 のときは $E = 0$ とした．

図 4 から図 8 より，**remote-20** で顕著な性能向上が得られていることが分かる．4 ノード × 114 ワーカを用いた場合では，**priority-all** と比較すると **remote-20** は Fib(50) において 44.7%，Nq(17) において 62.5%，Pen(15) において 16.5%，Cmp(150000) において 73.8%，Histogram(50,50,7) において 62.4% の性能改善が得られた．

relaxed-3 については **remote-20** と比較すると，Fib(50)，Nq(17)，Cmp(15000)，Histogram(50,50,7) では性能比に大きな変化は見られず，Pen(15) では-22.5%悪化している．この評価では，**remote-20** によりすでに十分な性能改善がなされていたことが原因として考えられる．

6　関連研究

ワークスティーリングフレームワークは一般的に，マルチスレッド言語 [3,4,6,8,10] やライブラリ [7] として実装されている．マルチスレッド言語とは異なり，Tascell [5] は論理スレッドフリーかつオンデマンドな並列実行を提供する．

Duran らはタスク並列向けに adaptive cut-off を提案している [2]．この手法では実行時にアプリケーションから収集された情報を用いてどのタスクを枝切りするか決定する [2]．Wang らは，タスクを実行中のワーカがアイドルな他ワーカを検出した時点からも，指定されたカットオフに応じた数の分割可能なタスクを生成するという手法により動的なタスク粒度調整を実現している [11]．Tascell はあらかじめ論理スレッド (あるいは OpenMP 用語におけるタスク) を生成しない．また，Tascell では粒度制御はスティール時に適用することができ，生成時のカットオフとは異なり，常にスティールする機会があり，仮想確率的ガード [14,15] や優先度および重み付選択 [9,16] を用いたときには，thief は victim 上の大域的な情報を利用することができる．

八杉らは、階層的計算省略に基づく並列実行モデル HOPE [13] を提案しており、これは主に耐障害性を実現している．

7　まとめと今後の課題

本研究では，Tascell に定期的に外部ノードへのタスク要求を行うスティール戦略と取り返しでないタスク要求を許容するという取り返し制約を緩和したスティール戦略を実装した．性能評価では，定期的に外部ノードへのタスク要求を行うことで性能向上が得られることを確認した．また，取り返し制約の緩和による性能改善は得られていないため，より多くの場合

の評価も行っていきたい．

参 考 文 献

[1] Breuel, T. M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference*, 1988.

[2] Duran, A., Corbalán, J., and Ayguadé, E.: An Adaptive Cut-off for Task Parallelism, *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, 2008, pp. 36:1–36:11.

[3] Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proceedings of the International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, No. 748, Springer-Verlag, 1993, pp. 94–107.

[4] 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.

[5] 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.

[6] IBM Research: X10: Performance and Productivity at Scale. `http://x10-lang.org/`.

[7] Intel Corporation: *Intel Threading Building Block Reference Manual*, 2007. `http://threadingbuildingblocks.org/`.

[8] Mohr, E., Kranz, D. A., and Halstead, Jr., R. H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3(1991), pp. 264–280.

[9] Nakashima, R., Yoritaka, H., Yasugi, M., Hiraishi, T., and Umatani, S.: Extending a Work-Stealing Framework with Priorities and Weights, *Proceedings of the 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3 2019) (held in conjunction with SC 2019)*, November 2019, pp. 9–16.

[10] Taura, K., Tabata, K., and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP'99)*, May 1999, pp. 60–71.

[11] Wang, L., Cui, H., Duan, Y., Lu, F., Feng, X., and Yew, P.-C.: An Adaptive Task Creation Strategy for Work-stealing Scheduling, *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, 2010, pp. 266–277.

[12] 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, Mar 2006, pp. 170–184.

[13] Yasugi, M., Muraoka, D., Hiraishi, T., Umatani, S., and Emoto, K.: HOPE: A Parallel Execution Model Based on Hierarchical Omission, *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*, August 2019, pp. 77:1–77:11.

[14] Yoritaka, H., Matsui, K., Yasugi, M., Hiraishi, T., and Umatani, S.: Extending a Work-Stealing Framework with Probabilistic Guards, *Proceedings of the 45th International Conference on Parallel Processing Workshops (ICPPW 2016) (Ninth International Workshop on Parallel Programming Models and Systems Software for High-End Computing P2S2 2016, held in conjunction with ICPP 2016)*, August 2016, pp. 171–180.

[15] Yoritaka, H., Matsui, K., Yasugi, M., Hiraishi, T., and Umatani, S.: Probabilistic guards: A mechanism for increasing the granularity of work-stealing programs, *Parallel Computing*, Vol. 82(2019), pp. 19–36. (Available online 22 June 2018).

[16] 寄高啓司, 八杉昌宏, 平石拓, 馬谷誠二: 優先度ならびに重みを用いたワークスティールフレームワークの性能改善, The 1st. cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming (xSIG 2017), April 2017. (口頭発表，ヤング・リサーチャー枠).

[17] 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.

[18] 平石拓, 河野卓矢, 八杉昌宏, 馬谷誠二, 湯淺太一: バックトラックに基づく負荷分散の高並列環境における評価, 情報処理学会研究報告 *(SWoPP 2010)*, Vol. 2010-HPC-126, No. 25(2010), pp. 1–11.