Vol.2009-HPC-121 No.7
2009/8/4

バックトラックに基づく負荷分散の T2K 並列環境における評価

平石 拓[†1] 八杉昌宏[†2]
馬谷誠二[†2] 湯淺太一[†2]

マルチコアプロセッサ等を含む並列計算環境が一般的になるに伴い，並列計算向け高生産性言語はより重要になっている．Cilk 言語は共有メモリ環境におけるそのような言語の一つであり，不規則アプリケーションを含む多くのアプリケーションにおいて良好な負荷分散を実現する．すなわち，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ワーカを有効活用する．我々は，論理スレッドフリーなフレームワーク Tascell を提案している．Tascell は Cilk に対して生産性を損なうことなくより高い性能を実現し，クラスタを含む多様な並列計算環境にも対応する．Tascell ワーカは本物のタスクを生成（spawn）するが，それは他のアイドルなワーカから要求されたときであり，一時的バックトラックによる最古のタスク生成可能状態の復元に基づく．本手法により，論理スレッド生成・管理コストを削減でき，作業空間に対する参照局所性の向上させることができる．さらに Tascell では，作業空間の遅延コピーを伴う，すっきりとした効率良いバックトラック探索アルゴリズムが実現できる．本論文では，各ノード計 16 コアを備えるクラスタである T2K オープンスーパーコンピュータにおける本手法の評価結果を報告する．評価において NUMA 環境におけるスケーラビリティの低下等の問題は見られたが，多くのアプリケーションにおいては，多コア環境においても充分な速度向上が得られることが確認できた．

Evaluation of Backtracking-based Load Balancing on T2K Open Supercomputer

Tasuku Hiraishi,[†1] Masahiro Yasugi,[†2]
Seiji Umatani[†2] and Taiichi Yuasa[†2]

High-productivity languages for parallel computing become more important as parallel environments including multicores become more common. Cilk is such a language. It provides good load balancing for many applications including irregular ones; that is, it keeps all workers busy by creating plenty of "logical" threads and adopting the oldest-first work stealing strategy. This paper proposes a "logical thread"-free framework called *Tascell*, which achieves a higher performance than Cilk and supports a wider range of parallel environments including clusters without loss of productivity. A Tascell worker spawns a "real" task only when requested by another idle worker. The worker performs the spawning by temporarily "backtracking" and restoring its oldest task-spawnable state. Our approach eliminates the cost of spawning/managing logical threads, improves locality of reference for workspaces. Furthermore, Tascell enables elegant and highly-efficient backtrack search algorithms with delayed workspace copying. This paper reports performance evaluations of parallel computation on the T2K open supercomputer, a cluster that consists of computation nodes with four quad-core processors. Though there are some problems exposed such as limited scalability in NUMA environments, we can ensure that we can get sufficient scalability even in environments with computation nodes with large numbers of cores in many applications.

1. はじめに

マルチコアプロセッサ等を含む並列計算環境が一般的になるに伴い，並列計算向け高生産性言語はより重要になっている．Cilk 言語はそのような言語の一つであり，バックトラック探索のような不規則アプリケーションを含む多くのアプリケーションにおいて良好な負荷分散を実現する．すなわち，多数の**論理スレッド**を生成して最古優先（oldest-first）のワークスティールを採用することで全ワーカを有効活用する．

これに対し，我々は**論理スレッドフリー**なフレームワーク Tascell[4] を提案している．Tascell ワーカは本物のタスクを生成 (spawn) するが，それは他のアイドルなワーカから要求されたときであり，**一時的バックトラック**による最古のタスク生成可能状態の復元に基づく．この手法は，論理スレッド生成・管理コストの削減，作業空間の再利用促進，参照局所性改善などの利点を持つとともに，作業空間の遅延コピーによるすっきりとした効率良いバックトラック探索アルゴリズムを実現する．また，単一のプログラムを，合理的な効率とスケーラビリティにおいて共有メモリ環境でも分散メモリ環境でも実行できる．

本手法の性能評価は文献 4) でも行っており，上記で挙げた効果が実際に得られることを

†1 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
†2 京都大学情報学研究科
Graduate School of Informatics, Kyoto University

Vol.2009-HPC-121 No.7
2009/8/4

```
int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   s1 = fib(n - 1);
   s2 = fib(n - 2);
   return s1 + s2;
 }
}
```

図 1 C による Fibonacci 数計算プログラム
Fig. 1 C program for Fibonacci.

```
int a[12];   // 作業空間：未使用のピース
int b[70];   // 作業空間：盤面．(6+ 番兵 ) × 10 個のセル

// a[] 中の j0 番目から 12 番目までのピースの設置を試みる
// i 番目（i<j0）のセルは設置済み
// b[k] が盤面中の最初の空きセル
int search (int k, int j0)
{
  int s=0; // 見つかった解の数
  for (int p=j0; p<12; p++) { //未使用のピースについて反復
    int ap=a[p];
    for (each possible direction d of the piece) {
      ... local variable definitions here ...
      if (ap 番目のピースが d の方向に置けるか？);
      else continue;
      ap 番目のピースを盤面 b に設置し，a も更新
      kk = the next empty cell;
      if (no empty cell?) s++;          // 解発見
      else s += search (kk, j0+1); // 次のピース
      ap 番目のピースを取り除き，a も元に戻す（バックトラック）
    }
  }
  return s;
}
```

図 2 Pentomino パズル全解探索の C プログラム
Fig. 2 C program for Pentomino.

確認しているが，共有メモリ環境の多コア環境における評価は行っていなかった．そこで本論文では，本手法を各ノードに 4 台のクアッドコア Opteron を備えた京都大学の T2K オープンスーパーコンピュータにおける性能評価の結果を報告する．

以降の章では，まず 2–5 章で本提案手法およびその実装の説明を行い，6 章で性能評価を示し，最後に 7 章でまとめと今後の課題を述べる．

2. 率直な負荷分散が困難な例

本章では，3 章の提案手法の説明の準備として，一般に効率的な並列化が困難な 2 つの樹状再帰のアルゴリズムを導入し，それらの並列化がなぜ困難かを説明する．

最初の例は，n 番目の Fibonacci 数を再帰的に求めるアルゴリズムである．C による逐次プログラムは**図 1** のように書ける．各関数呼び出しにおいて，fib$(n-1)$ と fib$(n-2)$ は並列に実行することが可能である．

二つ目の例は，Pentomino パズルの全解探索アルゴリズムである．逐次の C プログラムは**図 2** のように書ける．各関数呼び出しで，未使用のピースについての反復（最外ループ）と，ブロックを置く各方向についての反復（1 つ内側のループ）を行っており，最外ループの並列化が可能である．

Pentomino は 1 ステップ進むごとにピースを盤面に設置しバックトラックの際に取り除くというバックトラック探索を行っているが，そのため，盤面の状態を管理するための**作業空間** が使われていることに注意する．

これらを並列化する率直な方法として，各ワーカがその時々の状態に基づいてタスク生成するかどうかを判断する方法が考えらえる．そのような方法に基づくタスク並列のプログラムを**図 3** に示す．各ワーカは fib(n) の計算において，`fib(n-2)` の計算をタスクとして生成するか，生成せずに自分で計算するかを何らかの基準で判断している．「何らかの基準」としては，例えばタスクを要求しているワーカが存在するときのみタスクを生成する，などが考えられる．効率良い負荷分散のためには，全実行時間にわたって全てのワーカを busy にできるような最小回数のタスク生成が行われることが理想であり，そのため各タスクはできるだけ大きなサイズに分割されることが求められる．つまり，各ワーカは計算の初期段階で適切な数のタスクを生成しておき，その後はずっと（計算終盤の微調整を除き）生成を行わないという戦略を採るのが最善である．しかし，そのような戦略は実行全体についての正確な情報（予測）がなければ実現不可能である．したがって，この戦略はうまくいかない．

同様の戦略を Pentomino の最外ループに適用する場合，各ワーカは，全ての反復を自分で計算するか，半数の反復を新たなタスクとして生成するかを「何らかの基準」で選択する（生成することを選択した場合，残った反復に同様の方法を再帰的に適用する）ことになるが，やはり同様の理由でうまくいかない．

次章では，このようなアルゴリズムに対する「実現可能な」負荷分散の戦略である，我々の提案手法を説明する．

3. 提 案 手 法

我々が提案している Tascell では，**一時的バックトラック**を行うことで，タスクの**遅延分割**を行う．

図 1 や図 2 の逐次計算は，左深さ優先の**呼び出し木**の走査と見ることができる．2 章の

```
// タスクオブジェクトの構造定義
struct tfib {
  int n;  // 入力
  int r;  // 出力
};
// タスクのエントリポイント
void exec_fib_task (struct tfib *pthis)
{ pthis->r = fib (pthis->n); }

int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   if (choose not to spawn?) {
     s1 = fib(n - 1);
     s2 = fib(n - 2);
   } else {
     Allocate a workspace of struct tfib as this.
     this.n = n - 2;  // 入力値をセット
     Send this as a newly spawned task.
     s1 = fib(n - 1);
     Wait and receive the result of this.
     s2 = this.r;      // 出力値を獲得
     Deallocate this workspace.
   }
   return s1 + s2;
 }
}
```

図 3 Fibonacci を率直にタスク並列化したプログラム
Fig. 3 Straightforward task-parallel program for Fibonacci.

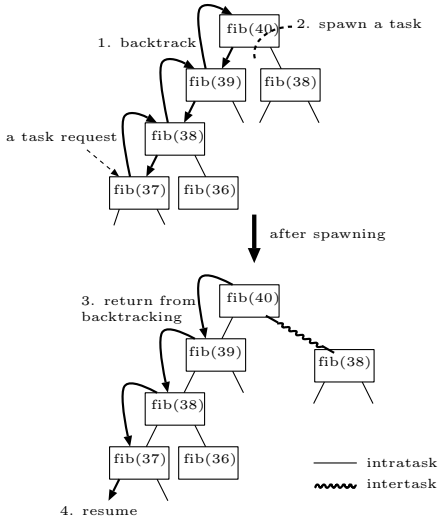

図 4 fib(40) におけるタスクの遅延生成．Tascell のワーカがタスク要求を（fib(37) で）検出すると，(1) まず最古のタスク生成可能時点までバックトラックし，(2) fib(38) のタスクを生成し，(3) バックトラックから復帰し，(4) 自らの計算を再開する．
Fig. 4 Spawning a task lazily while computing fib(40). When a Tascell worker detects a task request (at fib(37)), it (1) backtracks to the oldest task-spawnable point, (2) spawns a task for fib(38), (3) returns from backtracking, and (4) resumes its own computation.

率直な戦略（Fibonacci における図 3）で常に「生成しない」ことを選択した場合も同様である．

Tascell では，ワーカは常に**最初は**「生成しない」ことを選択するが，他のワーカからタスク要求を受けると，**過去の選択を変更したかのように**タスクを生成する．つまり，**図 4・図 5** に示すように，ワーカは (1) まず過去の時点にバックトラックし，(2) タスクを生成し（この時，実行経路をそのタスクの結果を受け取るように変更），(3) 自分が行っていた計算を再開する．

このように，最も古いタスク分割可能時点に遡ることにより，一般には最も大きいタスク（図 4 では木のルートの右側の fib(38)）を生成することができる．

逐次の Pentomino では，ワーカは自らの**作業空間**を持ち，そこにステップごとにピースを置い（do）たり取り除い（undo）たりすることで効率良く探索を行っている．ワーカがタスクを生成する際には， 新しいタスク用の作業空間を allocate した上で，そこに「現在の」状態をコピーする必要がある．提案手法においては,「現在の」内容とは過去の選択時点における内容のことである．そこで，ワーカは図 5 のように (1) のバックトラック順に沿って適切な undo を行うことにより過去の内容を復元し，(2) のタスク生成時にその復元した作業空間のコピーを作成する．その後，(3) のバックトラックからの復帰時に適切な redo を行うことで (4) における自らの計算の再開を可能にする．

Cilk[2] や MultiLisp[3] では load-based inlining（本質的には 2 章で説明した率直な手法）の問題を解決するために，Lazy Task Creation (LTC)[7] と呼ばれる手法を採用している．本手法は LTC に対して，(1) 論理スレッドを一切生成しないので，タスクキューの管理コストが発生しない，(2) マルチスレッド言語では，各（論理）スレッド用に作業空間を確保する必要があるが，本手法では単一の作業空間を再利用し続けることが可能であり，参照局所性が向上する，[⋆1] (3) バックトラック探索をマルチスレッド言語で実装する場合，親ス

⋆1 Cilk では `SYNCHED` という疑似変数を利用することで，子スレッド間については作業空間の再利用が可能だが，親子スレッド間での再利用はできない．

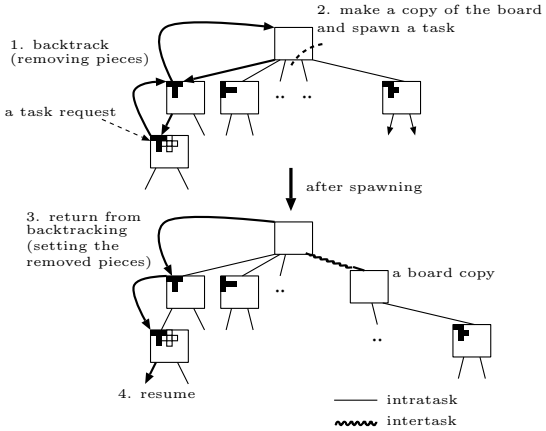

図 5 バックトラック探索 Pentomino におけるタスク遅延生成．図 4 との違いは，(1) バックトラック中の undo 処理（ピース除去），(2) 半分の反復をタスクとして生成する際の，一時的に過去の状態に戻った盤面のコピー，および (3) バックトラックからの復帰中の redo 処理（ピースの再設置）．

Fig. 5 Spawning a task lazily while performing backtrack search for Pentomino. Unlike in Fig. 4, (1) the backtracking step includes undo operations (i.e., removing pieces). (2) The spawning-half-iterations step includes making a copy of the temporarily restored board. (3) The returning-from-backtracking step includes redo operations (i.e., setting pieces).

レッドの作業空間のコピーを各スレッド生成時に用意する必要があるが，本手法では，一時的バックトラックを用いることによりそのようなコピーを遅延できる，(4)（異機種混合環境を含む）分散メモリ環境に分散共有メモリを利用しなくても対応しやすい，という優位点を持つ．

LTC では実際に生成されるタスクの数（タスクスティールの回数）は論理スレッド数に比べて極めて少ないことを仮定している．我々の手法でも同様に，実際に生成されるタスクの数（スティールの回数）は非常に小さいと仮定している．我々の手法は，LTC よりタスクスティールのコストが高くなることを許容し，逐次計算のオーバーヘッドを極めて小さく抑えるものであり，上記の仮定のもとで全体の性能は向上する．

一時的バックトラックや undo，redo 操作をどのように行うかは，我々の拡張言語で追加したコンストラクトを用いて指示する．これについての詳細は 4.2 節で述べる．

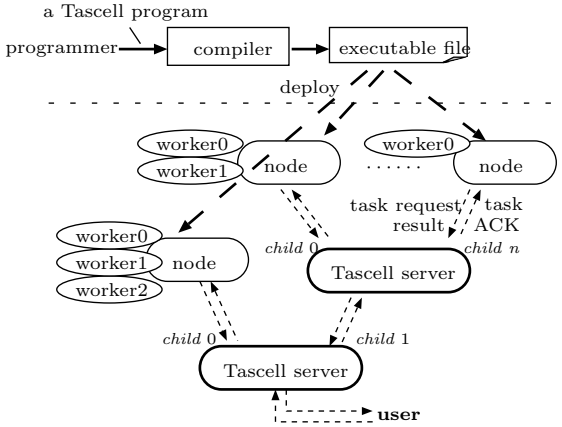

図 6 Tascell フレームワークにおけるプログラムのコンパイル・実行

Fig. 6 Multistage overview of the Tascell framework.

4. Tascell フレームワーク

提案手法を実現するため，我々は **Tascell フレームワーク**を設計・実装した．このフレームワークは Tascell サーバおよび Tascell 言語のコンパイラより構成される．

4.1 フレームワークの概要

図 6 に，Tascell フレームワークにおいてプログラムがどのようにコンパイル，実行されるかを示す．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（起動時に接続先の Tascell サーバを指定する）．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．さらに，Tascell サーバを介して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．

Tascell サーバは，計算ノード間のメッセージの中継や，ユーザインタ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などを行う．なお，図のように Tascell サーバをさらに別の Tascell サーバに接続させることで，木を構成することもできる．これは，一つのサーバに接続されるノード数が増えすぎてサーバの負荷が高くなった場合や，接続に NAT 越えが必要な複数のクラスタで計算を行いたいなどの場合に有効である．

負荷分散はアイドルなワーカが負荷の高いワーカにタスク要求を行うことで実現する．タスク要求メッセージには，相手ワーカを特定するものと，任意のワーカに対するもの（any

```
// タスク tfib の定義
task_tfib {
  in:  int n;  // 入力
  out: int r;  // 出力
};
// tfib のエントリポイント
// タスクオブジェクト this の宣言が暗黙に行われる
task_exec tfib
{ this.r = fib (this.n); }

worker int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   do_two          // Tascell のコンストラクト
     s1 = fib(n - 1);
     s2 = fib(n - 2);
   tfib {  // タスクオブジェクト this の宣言が暗黙に行われる
     // put 部（タスク送信前に実行）
     { this.n = n - 2; }
     // get 部（結果受信後に実行）
     { s2 = this.r; }
   }               // end of do_two
   return s1 + s2;
 }
}
```

図 7 Fibonacci の Tascell プログラム
Fig. 7 Tascell program for Fibonacci.

要求）の 2 種類がある．メッセージは計算ノードの実行時システムや Tascell サーバにより中継されるが，any 要求の転送先にはなるべく負荷の高いワーカ・計算ノードを選択する．このような一連のメッセージはすべて内部的に処理され，プログラマは個々のメッセージを明示的に扱う必要はない（扱えない）．

各タスクおよびその結果は**タスクオブジェクト**としてやりとりされる．オブジェクトの構造は Tascell プログラム中で定義される．ノード内ワーカ間のオブジェクトの受け渡しは，共有メモリを介してポインタの受け渡しだけで高速に行える．ノードを跨る場合は，オブジェクトをシリアライズし，⋆1 Tascell サーバを介して送信する．

4.2 Tascell 言語

Tascell 言語は C の拡張言語である．**図 7** と**図 8** に Tascell プログラムの例を示す．（下線部が C に対する拡張部分．）プログラマは既存の逐次プログラムをベースにして，Tascell

⋆1 Tascell コンパイラは 4.2.1 節で説明するタスク定義の in:，out:の情報に基づき，デフォルトのシリアライザ・デシリアライザの定義を自動的に生成する．ただし，オブジェクトの一部の要素のみの選択的な転送（通信量を減らすため）や，動的に生成されたデータの転送などには，自動生成では対応できないため，プログラマ自身がシリアライザ・デシリアライザを定義することもできる．

```
task pentomino {
  out: int s;    // 出力
  in: int k, i0, i1, i2;
  in: int a[12]; // 作業空間：未使用のピース
  in: int b[70]; // 作業空間：盤面．(6+ 番兵 ) × 10 個のセル
};
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; // 見つかった解の数
 // Tascell の並列 for 文
 for (int p : j1, j2)
  {
   int ap=tsk->a[p];
   for (each possible direction d of the piece) {
    ... local variable definitions here ...
    if(ap 番目のピースが d の方向に置けるか？);
    else continue;
    dynamic_wind // アンドゥ・リドゥ操作指示コンストラクト
     { //  dynamic_wind のドゥ・リドゥ操作
       ap 番目のピースを盤面 tsk->b に設置し，tsk->a も更新
     }
     { // dynamic_wind 本体
       kk = the next empty cell;
       if (no empty cell?) s++; // 解発見
       else                     // 次のピース
        s += search (kk, j0+1, j0+1, 12, tsk);
     }
     { // dynamic_wind のアンドゥ操作
       ap 番目のピースを取り除き，tsk->a も元に戻す
       （バックトラック）
     } // end of dynamic_wind
   }
  }
 pentomino (int i1, int i2) // this の宣言および
            // 範囲（i1-i2）の設定が暗黙になされる
  {
   // put 部（タスク送信前に実行）
   { // 反復の前半分に相当するタスクの入力を設定
     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 部（結果受信後に実行）
   { s += this.s; }
  }  // end of parallel for
 return s;
}
```

図 8 Pentomino の Tascell プログラム
Fig. 8 Tascell Program for Pentomino.

で追加されたコンストラクトを用いてワーカのプログラムを書くことができる．Tascell で追加された主なコンストラクトを以下で説明する．

4.2.1 タスク定義

タスク定義は下記のトップレベル宣言により行う．

`task` *task-name* `{`[`in:`|`out:`] *struct-declaration* ⋯`};`

例として，図 7 中の “`task tfib {in: int n; out: int r};`” は，`tfib` というタスクのオブジェクト構造を定義している．構文は C の構造体の定義とほぼ同様だが，`in:` または `out:` の属性を各フィールドに指定することができる．コンパイラはこれらの属性を参照して，外部ノードへタスクを送受信する手続きを生成する．

タスク *task-name* が実際に行う計算は，

`task_exec` *task-name* `{` *body* `}`

のトップレベル宣言により定義する．*body* 中では，図 7 中の “`this.r =fib(this.n);`” のように `this` を用いてタスクオブジェクトを参照できる（タスクの入力の値が含まれている），タスクの計算結果を適切なフィールドにセットすることをプログラムに要求する．また，*body* では以下で説明する**ワーカ関数**を呼び出すことができる．

4.2.2 ワーカ関数

通常の関数定義に `worker` キーワードを付加すると，その関数は**ワーカ関数**となる．以下で説明するタスク分割コンストラクト（およびワーカ関数の呼び出し）の使用は，ワーカ関数および `task_exec` の本体でのみ可能である．(この制限は Cilk の `cilk` 手続きと同様.)

4.2.3 タスク分割指示コンストラクト

Tascell で追加された文 (statement)

`do_two` $statement_1$ $statement_2$ *task-name* `{` $statement_{put}$ $statement_{get}$ `}`

により，$statement_2$ の計算（図 7 では “`fib(n-2)`”）を $statement_1$ （“`s1=fib(n-1);`”）の計算中に生成してもよいことを指示する．より正確には以下の処理が行われる．(1) ワーカはまず暗黙の**タスク要求ハンドラ**を有効にし，$statement_1$ を実行する．ハンドラが起動すると，ワーカは新しいタスク *task-name* を生成し，$statement_{put}$ （図 7 の “`this.n=n-2;`”）によりタスクオブジェクトの入力フィールドに値をセットし，そのタスクオブジェクトをタスク要求元に送信した後，バックトラックから復帰する．(2a) $statement_1$ の実行中にタスク要求ハンドラが起動していなければ $statement_2$ がそのまま実行される．(2b) ハンドラが起動していた場合，$statement_2$ の実行は飛ばされ，ハンドラが生成・送信したタスクの終了および結果の受信を待ち合わせる．その後，$statement_{get}$ （図 7 の “`s2=this.r;`”）で，受信したタスクの結果を取り込むことで，最終的に $statement_2$ の実行と同等の結果を得る．なお，結果の待ち合わせ中のワーカはアイドル状態にならないようにするため，別のタスクの要求・実行を行うが，この際，結果待ち合わせ中のタスクを実行しているワーカをタスクの要求先にしている（タスクの取り返し）．これは leapfrogging[10] と呼ばれる，実行スタックサイズの肥大を抑えるための手法である．

生成されるタスクの型は識別子 *task-name* により指定する．$statement_{put}$ や $statement_{get}$ 中では `this` キーワードによりタスクオブジェクトを参照できる．$statement_{put}$ ではこのオブジェクトの入力フィールドにタスクの入力をセットすること，$statement_{get}$ ではオブジェクトの出力フィールドを参照してタスクの結果を獲得し $statement_2$ と等価な処理を行うことが，それぞれプログラマに要請される．

Tascell では反復計算を分割するために，並列 `for` ループ構文を用意している．構文は以下の通りである．

`for(int` *identifier* `:` $expression_{from}$`,` $expression_{to}$`)`
$statement_{body}$
task-name `(int` $identifier_{from}$`, int` $identifier_{to}$`)` `{` $statement_{put}$ $statement_{get}$`}`

図 8 の Pentomino プログラムでは並列 `for` ループ “`for (int p: j1, j2) {...} pentomino (int i1,int i2) {{...} {s+=this.s}}`.” により最外ループを並列化している．

この文は，$expression_{from}$ 以上 $expression_{to}$ 未満の整数に対して $statement_{body}$ を繰り返す．この繰り返し実行の間，暗黙のタスク要求ハンドラが有効になる．このハンドラが起動すると未処理の反復の前半分が新しいタスクとして生成される．実際に新しいタスクに割り当てられる範囲は $statement_{put}$ 中で，$identifier_{from}$ と $identifier_{to}$ で参照できる．生成されたタスクの結果は $statement_{get}$ で処理する．

Tascell はまた，undo，redo 操作を定義するため，Scheme 言語[6] と同様の `dynamic_wind` コンストラクトを備えている．構文は以下の通りである．

`dynamic_wind` $statement_{before}$ $statement_{body}$ $statement_{after}$.

この文は基本的には，$statement_{before}$ （図 8 ではピースを置く “do” 処理），$statement_{body}$，$statement_{after}$ （ピースを取り除くという “undo” 処理）の順で実行される．これに加え，$statement_{after}$ は $statement_{body}$ の実行中，この文よりも古いタスク要求ハンドラの起動**前**にも（“undo” 処理として）実行される．また $statement_{before}$ はそのようなハンドラの起動終了**後**にも（“redo” 処理として）実行される．

do_two，並列 for，dynamic_wind の各文は $statement_1$ や $statement_{body}$ 内で動的にネストさせることができる．この時，図 4 や図 5 のように複数のタスク要求ハンドラおよび undo，redo 節が同時に有効になる．各ワーカは do_two と並列 for の先頭においてポーリングによりタスク要求を検出する．検出すると一時的なバックトラックにより最も古い要求ハンドラを起動することで，できるだけ大きいタスクを生成しようとする．バックトラックの途中に undo，redo 節があった場合は，全ての undo 節がバックトラックの経路順に実行され，ハンドラ起動後，全ての redo 節がその逆順に実行される．

5. Tascell コンパイラの実装

5.1 入れ子関数による stack walk の実現

高い可搬性を確保するため，Tascell コンパイラは C 言語へのトランスレータとして実装した．ただし，Tascell の一時的バックトラック機構を実現するためには “stack walk”（実行スタックの底で眠っているフレームへのアクセス）が必要となり,「標準の」C レベルでの実装は困難である．我々は，**入れ子関数**を利用することでこの問題を解決した．

入れ子関数とは，関数定義の本体で定義される関数のことである．入れ子関数の定義を評価すると，生成時環境における lexical スコープの変数にアクセス可能なクロージャが得られる．このクロージャを間接的に呼び出すことにより，合法的な実行スタックへのアクセスが可能となる．

入れ子関数は，標準の C 言語の仕様には含まれていないが，GCC[1),8)] に拡張仕様として含まれている．しかし，GCC の実装ではクロージャの維持・生成コストが高いため，我々の以前の研究で **L-closure** に基づく入れ子関数の実装を提案し，それらのコストの大部分を削減することに成功した．L-closure の実装には，標準の C 言語への変換に基づくもの (**LW-SC**)[5)] と GCC の拡張に基づくもの (**XC-cube**)[11)] がある．性能は XC-cube のほうが優れているが，LW-SC にはプラットフォームごとに実装を用意する必要がないという利点がある．

5.2 入れ子関数を含む C コードへの変換

図 7 の Tascell プログラムは**図 9** の（入れ子関数付きの）C プログラムに変換される．各ワーカ関数には，最新の do_two，並列 for，dynamic_wind のハンドラに対応するクロージャを渡すための引数_bk0 が追加される． 各 do_two 文は，入れ子関数の定義（図 9 の _bk1_do_two）を持つコード断片に変換される．この関数はポーリングによってタスク要求が検出されたときに呼ばれる．入れ子関数の本体では，まず二番目に新しいハンドラに対応する入れ子関数_bk0 を呼び，より大きなタスクの生成を試みる．（その呼び出し先では，さらに大きなタスクが生成できないかを同様に試みる．）その試行の後，まだタスク要求が残っている場合のみ，新しいタスクが生成されタスク要求元に送信される．送信後，ワーカは入れ子関数の呼び出し元にリターンし，自らの計算を再開する．

```
int fib(void (*_bk0) (void), struct thread_data *_thr,
        int n)
{
  if (n <= 2)
    return 1;
  else {
    int s1, s2;
    { /*---------------- do_two ----------------*/
      struct tfib pthis[1];               // 作業領域
      int spawned = 0; //statement2 のタスクが生成済？
      {
        void _bk1_do_two (void) // 入れ子関数
        {
          if (spawned) return;
          _bk0(); // 後戻り継続
          if (task request exists?) {
            pthis->n = n - 2;     //statement_put
            spawned = 1;
            // タスク生成・送信
            make_and_send_task(_thr, 0, pthis);
          }
        }
        if (_thr->req)    // ポーリング
          _bk1_do_two (); // 一時的後戻り開始
                          // （入れ子関数の呼び出し）
        {
          s1 = fib(_bk1_do_two, _thr, n-1); //statement1
        }
      }
      if (spawned) {
        // 生成しタスクの結果をマージ
        wait_rslt(_thr);
        s2 = pthis->r;                  // statement_get
      } else {
        s2 = fib(_bk0, _thr, n - 2);    // statement2
      }
    } /*---------------- do_two ----------------*/
    return s1 + s2;
  }
}
```

図 9 図 7 の fib 関数の変換結果（do_two 文の変換を含む）
Fig. 9 Translation result from the worker function fib in Fig. 7, including translation of a do_two statement.

並列 for 文も同様に変換できる．ただし，入れ子関数の本体で，ワーカは新しいタスクのためのループ範囲を計算し，さらに自分が計算するループ範囲を更新する必要がある．dynamic_wind の変換では，$statement_{body}$ を入れ子関数の定義を含むコード断片に変換する．この関数本体には，$statement_{after}$ の複製（undo 処理），次に新しい入れ子関数の呼び

出し式，$statement_{before}$ の複製（redo 処理）が置かれる．

6. T2K における評価

T2K オープンスーパーコンピュータで本手法の性能評価を行った（評価環境の詳細を**表 1** にまとめる）．ベンチマークプログラムとして，説明で用いた Fib(n)，Pentomino(n) のほかに n 女王問題の全解探索（Nqueens(n)），$n \times n$ の行列の LU 分解（LU(n)），2 つの n 要素の配列間の全要素ペア（(a_i, b_j) for all $0 \leq i, j < n$）について比較演算を実行するプログラム（Comp(n)），$(2n+1)^3$ 個の同一質量の質点からある点にかかる総引力を計算するプログラム（Grav(n)）を用いた．

Tascell の Nqueens プログラムは，Pentomino と同様，並列 `for` と `dynamic_wind` を組み合わせて書かれている．LU と Comp は，`do_two` を用いて cache-oblivious な再帰アルゴリズムで書かれている（サイズ n と m ($n \geq m$) の配列を比較する Comp タスクは，サイズ $n/2$ と m の配列を比較する 2 つのタスクに分割される）．Grav は反復アルゴリズムであり，Tascell では並列 `for` の三重ネスト（各ネストが座標軸に対応）で実装されている．なお，全てのプログラムは細粒度並列の実装である．

まず，逐次実行のオーバーヘッドを評価するため，Tascell を 1 ワーカで実行したときの実行時間を，C と Cilk のそれぞれのプログラム（Tascell とほぼ同じアルゴリズムで実装[⋆1]）の実行時間と比較した．Cilk の Nqueens，Pentomino，Grav では各スレッドは計算のための作業空間を配列として保持する．さらに Nqueens と Pentomino では，各スレッドの生成時に親スレッドの作業空間をコピーしてもらう必要がある（`SYNCHED` を用いた場合でも）．そのため，大きなオーバーヘッドが生じる．Tascell のワーカは，全てのアプリケーションで一つの作業空間を再利用し続けることができる．

表 1 評価環境
Table 1 Evaluation environment.

CPU	AMD Quad Core Opteron 8350 Barcelona 2.2GHz×4（計 16 コア）
OS	Redhat Enterprise Linux AS V4
コンパイラ	GCC 3.4.3 （`-O2` 最適化） + LW-SC (Tascell)，Cilk 5.4.6 (Cilk)
ワーカ	`pthread_create` で生成 （`PTHREAD_SCOPE_SYSTEM` スケジューリング）
サーバ	Allegro Common Lisp 8.1 で実装（`(speed 3) (safety 1) (space 1)` でコンパイル）． 計算ノード（のうち一台）と同一のノードで起動．
ネットワーク	InfiniBand（各計算ノードが単一のサーバに TCP/IP 接続）

⋆1 Nqueens と Pentomino では `SYNCHED` による子スレッド間の作業空間の再利用（3 章の脚注参照）を行っている．

性能測定の結果を**表 2** に示す．Tascell のオーバーヘッドの主な要因はポーリングと入れ子関数の維持コストだが，ほぼ全てのアプリケーションで Cilk より非常に低く抑えられている．特に Fib では Cilk の論理スレッドの生成頻度が非常に高いため，差が顕著に現れる．Nqueens が Pentomino よりオーバーヘッドの差が大きいのは，Nqueens のほうが作業空間コピーの頻度が高いためであると考えられる．Tascell でも Cilk でも LU のオーバーヘッドはほとんどない．これは，タスク分割可能な箇所の発生頻度が少ないためである．

Cilk のみにある逐次オーバーヘッドの要因としては，(a) spawn の実行ごとに allocate される明示的フレームの管理コスト，(b) THE プロトコル[2]（一貫性のある論理スレッドキューへのアクセスのための手法）のコスト，(c) 各論理スレッドにかかる作業空間の間のデータコピーのコスト（Pentomino と Nqueens の場合）が挙げられる．このうち (c) のオーバーヘッドを見積もるため，表 2 には，タスク生成可能時点で人工的に作業空間のコピーを行うようにした Tascell プログラムの実行時間も示している．

1 ノード内（共有メモリ環境）で複数のワーカで並列計算を実行したときの性能測定の結

表 2 1 ワーカ実行時の逐次 C プログラムとの実行時間の比較．Tascell の「コピー有」は，各生成可能なタスクごとに，人工的に作業空間のアロケーションとコピーを行うようにした場合の測定結果．
Table 2 Execution time (and relative time to sequential C programs) with one worker. In「コピー有」(“w/ copying”) of Tascell, we performed artificial workspace allocation and copying between workspaces for each spawnable task.

	Elapsed time in seconds (relative time to plain C)			
	C	Cilk	Tascell	Tascell コピー有
Fib (40)	0.596 (1.00)	5.71 (9.59)	1.52 (2.54)	—
Nqueens (15)	8.10 (1.00)	22.2 (2.74)	11.9 (1.46)	17.2 (2.13)
Pentomino (12)	1.14 (1.00)	1.69 (1.47)	1.64 (1.43)	2.04 (1.78)
LU (2000)	9.21 (1.00)	9.30 (1.01)	9.24 (1.00)	—
Comp (30000)	3.32 (1.00)	6.02 (2.08)	3.99 (1.20)	—
Grav (200)	2.85 (1.00)	5.80 (2.03)	4.29 (1.50)	—

Vol.2009-HPC-121 No.7
2009/8/4

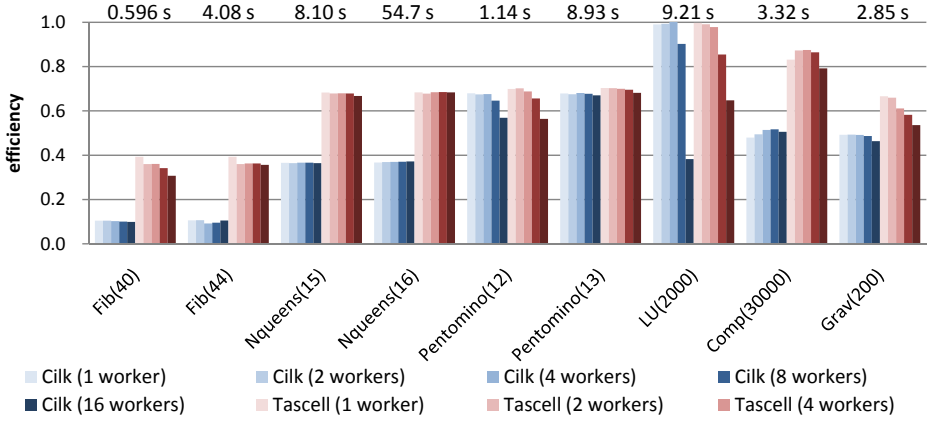

図 10 1 ノード内の複数ワーカによる並列計算の性能測定結果．縦軸の efficiency は S/n_w（S: 逐次 C プログラムに対する速度向上，n_w: ワーカ数）で定義した値（つまり，efficiency = 1 で理想の速度向上）．各ベンチマークの上部の数値は C の実行時間．

Fig. 10 Efficiency with multiple workers in a shared memory environment within one node. Efficiency is defined as S/n_w where S is a speedup to a sequential C program and n_w is the number of workers. (Efficiency = 1 means an ideal speedup.) The number above each group of bars shows the execution time in C.

果を**図 10** に示す．逐次性能の差がそのまま並列計算の結果にも反映され，LU 以外の全てのベンチマークで Tascell が Cilk より高い性能を示している．例えば，Nqueens(16) の 16 並列の計算では，Tascell は Cilk に対して 1.84 倍（=0.684/0.372）の速度向上を達成している．

ただし，Fib や Grav で顕著に見られるように，ワーカ数増加に伴う速度向上は Tascell のほうが鈍くなっている．これは，Tascell のノード内のワーカ間通信のオーバーヘッドが大きいためであると考えられる．

また Tascell ・Cilk ともに，LU において 8 並列以上で性能が急に低下している．これは，メモリバンド幅の飽和のほか，メモリが NUMA 構成である T2K ノードにおいて，全ての CPU から一つのメモリモジュールに配置された行列へのアクセスを多発させてしまっていることが原因と考えられる．この対策としては，タスクスティール時に行列の必要な部分を，ワーカが割り当てられた CPU に近いメモリにコピーしておくことも考えられるが，

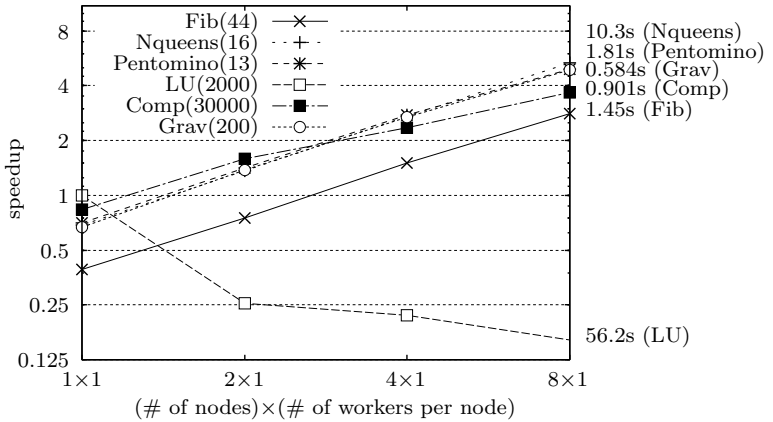

図 11 複数の計算ノードを用いた並列化による逐次 C プログラムに対する速度向上（各ノード 1 ワーカ）．各グラフ右の秒数は 8 ノードでの実行時間．

Fig. 11 Speedups with multiple computation nodes relative to sequential C (one worker running in each node). The numbers on the right side of each graph show execution time with 8 nodes.

ワークスティーリング方式の動的負荷分散においては頻繁にコピーが起こってしまうため，充分な性能向上を得るのは困難である（後述する，分散環境で速度向上を得るのが困難なのと同様の問題）．より現実的な解決策としては，LU 分解における行列のアクセスパターンに合致するように行列要素の配置順序を工夫し，キャッシュ性能を向上させることにより遠くのメモリへのアクセス回数を減らすことが考えられる．

次に，複数の計算ノードを単一のサーバに接続して，並列計算を行ったときの性能評価の結果を**図 11**，**図 12** に示す．[⋆1]

各ノードでワーカを 1 つだけ動かした場合（図 11），LU 以外の各タスクの計算コストが通信コストに対して比較的大きいベンチマークでは良い速度向上が得られている．Tascell ではタスクの分割回数が少なく抑えられているため，サーバによるボトルネックもここではあまり問題になっていない．一方，LU では全く速度向上を得ることができなかった．これは，タスク分割のたびに，タスクの入力および結果として部分行列の送受信が発生するためである．文献 9) でも考察されているように，LU のように大きなサイズの共有データを要するアプリケーションで，ワークスティーリング方式の動的負荷分散で充分な速度向上を得

⋆1 標準 Cilk は共有メモリ環境しかサポートしていないため，Tascell のみ評価を行った．

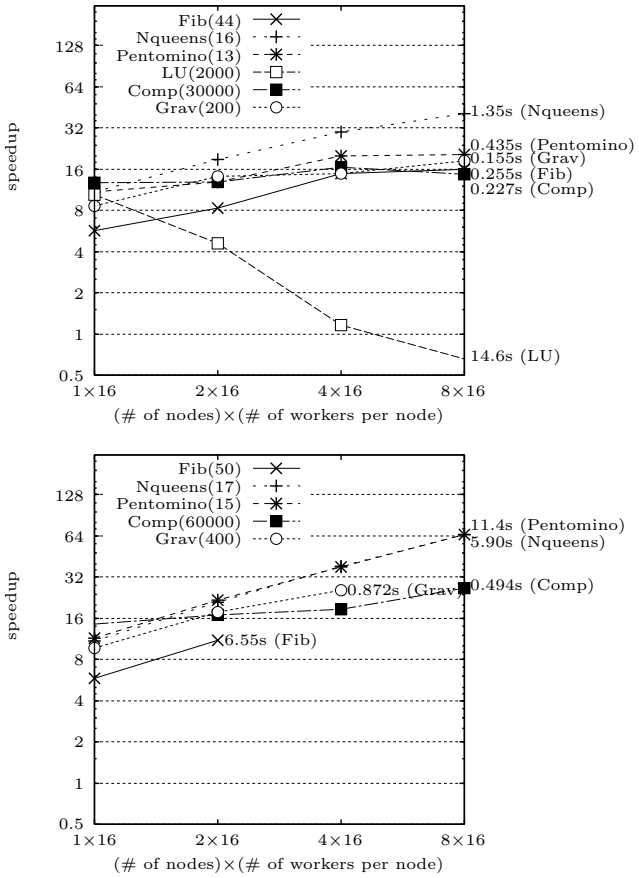

図 12 複数の計算ノードを用いた並列化による逐次 C プログラムに対する速度向上（各ノード 16 ワーカ）．各グラフ右の秒数は測定できた最大ノード数での実行時間．下のグラフは大きなサイズの問題での評価結果．

Fig. 12 Speedups with multiple computation nodes relative to sequential C (16 workers running in each node). The numbers on the right side of each graph show execution time with the maximum number of nodes. The graph shown below is the results for larger problems.

るのは非常に困難である．

16 個のワーカを各ノードで動かした場合（図 12）では，各ワーカはノード内・ノード間の二種類の通信を行いながら計算を進める．そのような環境においても，図 12 下のグラフが示すように，総ワーカ数に見合うサイズの問題（ワーカあたり数秒程度の計算量）を与えれば充分な速度向上を得ることができる．Comp では，$O(n^2)$ の計算コストに対して $O(n)$ の通信コストがかかるため，n が小さい場合は，スケーラビリティは特に制限されてしまう．

Fib(50) と Grav(400) の計算をあるノード数以上で行うと，途中で計算の進行が止まってしまい，測定結果を得ることができなかった．ノード間のメッセージは飛び交い続けているため，ワーカ・サーバ間の通信プロトコルの設計の問題によりライブロックが起こった可能性があるが，以下の理由よる大幅な性能低下の可能性もあると考えている．現在の Tascell の実装においては，ワーカはまずノード内のワーカに仕事を要求し，ノード内のどのワーカからも仕事が受け取れなかったときのみそのノードの代表ワーカが外部に仕事要求メッセージを出す．このような階層方式はノード間のタスク授受を減らすために一見有効であると考えらえるが，ノード内のワーカ数が多い環境においては非常に小さなタスクがノード間を飛び交う原因となり，逆にノード間通信を増やしてしまう可能性がある．すなわち，ノードの代表ワーカが受け取ったタスクは他の 15 個のワーカにより細断され，そのような小さなタスクがノード間で授受されてしまう．Fib や Grav は最小のタスク単位が非常に小さなアプリケーションなので，各ノードのワーカ数が増えたことにより，問題が顕在化してしまった可能性がある．この問題を解決するため，今後，階層方式以外の要求先決定戦略も実装・評価していく予定である．スタックサイズ上限の問題を無視すれば，leapfrogging （4.2.3 節を参照）を行わないようにすることでも問題を解決できる可能性がある．

なお，以前の評価[4] とは異なり，本評価では Tascell サーバを計算ノードのうち一台と同一のノードで動作させているが，それによる目立った性能低下は観測されなかった．

7. まとめと今後の課題

本論文では，我々が提案しているバックトラックに基づく負荷分散方式の，計 16 コアの Opteron プロセッサを備える T2K 並列環境での性能評価を示した．特にバックトラック探索のアルゴリズムでは，我々の手法により作業空間の間のコピーを遅延できることにより，大幅なオーバーヘッド削減を実現できる．それ以外のアプリケーションについても，論理スレッドを使わないことや，単一の作業空間を再利用し続ける効果により高い性能を得ることができる．本論文の評価により多コア環境でも十分な速度向上を得られることが確認できた

が，一方で NUMA 環境においてスケーラビリティが制限されることや，各ノードが多数のワーカを持つ分散環境において問題が発生することも観測された．

今後は，本評価で顕在化した問題の検証を行うほか，グラフアルゴリズムを用いたより実用的なアプリケーションに本手法を適用し，効果を評価していく予定である．

謝辞 本研究の一部は，「並列分散計算環境を安定有効活用する要求駆動型負荷分散」（21013027）（科学研究費特定領域研究「情報爆発時代に向けた新しい IT 基盤技術の研究」）ならびに科学研究費基盤研究（B）「安全な計算状態操作機構の実用化」（21300008）の助成を得て行った．

参 考 文 献

1) Breuel, T. M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).
2) 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, pp. 212–223 (1998).
3) Halstead, Jr., R.H.: New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, *Parallel Lisp: Languages and Systems* (Ito, T. and Halstead, R.H., eds.), Lecture Notes in Computer Science, Vol.441, Sendai, Japan, June 5–8, Springer, Berlin, pp.2–57 (1990).
4) 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)*, Raleigh, U.S. (2009). (to appear).
5) Hiraishi, T., Yasugi, M. and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol.2, pp.262–279 (2006). (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50-67.).
6) Kelsey, R., Clinger, W. and Rees, J.: Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices*, Vol.33, No.9, pp.26–76 (1998).
7) 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, pp.264–280 (1991).
8) Stallman, R.M.: Using and Porting GNU Compiler Collection (1999).
9) van Nieuwpoort, R.V., Kielmann, T. and Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications, *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, New York, NY, USA, ACM, pp.34–43 (2001).
10) 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)*, pp.208–217 (1993).
11) Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proceedings of 15th International Conference on Compiler Construction (CC2006)*, Lecture Notes in Computer Science, No.3923, Springer-Verlag, pp.170–184 (2006).