動的負荷分散フレームワーク Tascell の広域分散およびメニーコア環境における評価

平石 拓[†] 八杉昌宏[††] 馬谷誠二[††]

我々は不規則な問題，環境において粒度が大きくバランスのとれた負荷分散を可能にする並列プログラミング／実行フレームワーク Tascell を提案している．Tascell ワーカがタスクを要求した時に一時的にバックトラックを行うことで，並列化のコストを本質的に極小化した遅延分割型負荷分散を実現しており，例えば論理スレッドを利用する Cilk と比較しても優れた性能を示す．Tascell は複数の拠点に設置されたクラスタを WAN で接続した広域分散環境にも適用可能である．本研究では，高並列環境における Tascell の有効性を検証するために，Niagara 2 プロセッサ 2 台を備えたサーバ 128 ハードェアスレッド環境，および InTrigger のうち最大 5 クラスタ 288 コアを用いた環境における評価を行った．その結果，メモリバンド幅や通信遅延，ワークスティーリングの偏り等の影響によるスケーラビリティの制約は観測されたものの，本フレームワークが広域分散環境を含む高並列環境においてもおおむね有効にはたらくことが確認できた．

Evaluation of the Tascell Dynamic Load Balancing Framework in Widely Distributed and Many-Core Environments

Tasuku Hiraishi,[†] Masahiro Yasugi[††] and Seiji Umatani[††]

We proposed a parallel programming/execution framework named Tascell. It enables coarse-grained and well-balanced load balancing for irregular applications and environments. In order to realize load balancing with lazy partitioning that minimizes the cost of parallelization, a Tascell worker performs temporarily backtracking when load balancing is required. In fact, we obtained higher performance than Cilk, which employs logical threads. Tascell supports wide-area distributed environments, such as WAN connected clusters located on multiple sites. In this research, we evaluated Tascell in highly parallel environments; with the server with 128 hardware threads of two Niagara 2 processors, and with up to 288 cores in five InTrigger clusters. In the result, we ensured that Tascell works even in highly parallel environments including widely-distributed environments, though scalability is limited in some cases due to memory bandwidth limitations, communication delay, and biases of task requests.

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[3] 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洗練され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させることができる．実際，クラスタ環境において本手法の性能評価を行い，期待する性能が得られることを確認している[3),10)]．

Tascell フレームワークにおける分散メモリ環境対応は，通信中継サーバ（Tascell サーバ）に複数の計算プロセスを TCP/IP 接続させることによって実現している．ここで，Tascell サーバには計算ノードだけでなく別のサーバプロセスを接続させることもできる．これ

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

```
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 1 Pentomino パズル全解探索の C プログラム
Fig. 1 C program for Pentomino.

により，各クラスタの代表ノードにサーバプロセスを配置し，サーバおよび計算ノードからなるツリー状のネットワークを構成することで，WAN で接続された多拠点のクラスタに跨って 1 つの並列計算を行うことも可能である．

しかし，これまでに行った性能評価は最大 16 コアのノードからなる単一クラスタによるものであり，多コア環境や広域分散環境で実際に並列計算を行った場合に，十分なスケーラビリティを得られるかという評価は不十分であった．そこで本研究では，共有メモリ環境，(広域）分散メモリ環境双方において Tascell による高並列計算の評価を行った．具体的には，共有メモリ環境として SPARC Niagara 2 サーバ，広域分散メモリ環境として日本国内の多拠点のクラスタを相互に接続した計算環境である InTrigger[8] の最大 5 クラスタを用いて性能測定および結果の考察を行った．

以下，2 章で Tascell フレームワークの概要，3 章で Tascell におけるタスクスティーリング戦略の説明を行う．次に 4 章で評価を示し，最後に 5 章でまとめる．

2. Tascell の概要

本章では，Tascell の概要を説明する．詳細は文献 3)，10) を参照されたい．

2.1 提 案 手 法

例として，Pentomino パズルの全解探索アルゴリズムの並列化を考える．逐次の C プログラムは**図 1** 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）を行っているが，最外ループについての並列化を考える．

素朴な並列化の方法としては，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

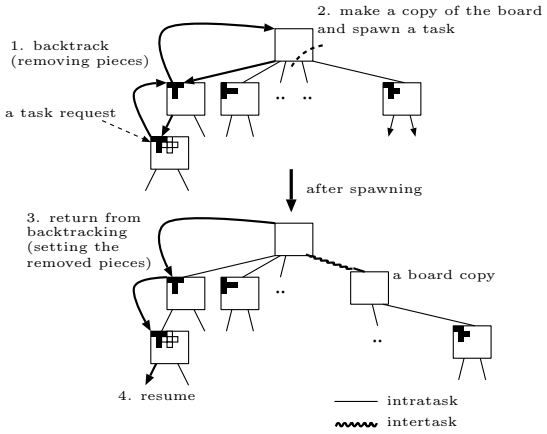


図 2 Pentomino 全解探索におけるタスク遅延生成．Tascell ワーカがタスク要求を（2 つめのピースを置く際に）検出すると，(1) 最古のタスク生成可能時点まで **undo（ピース除去）しつつ**バックトラックし，(2) 反復の半分をタスクとして生成し（**この時，一時的に過去の状態に戻った盤面をコピー**），(3) **redo（ピースを再設置）しつつ**バックトラックから復帰し，(4) 自らの計算を再開する．
Fig. 2 Spawning a task lazily while performing backtrack search for Pentomino. When a Tascell worker detects a task request, it (1) backtracks to the oldest task-spawnable point *with performing undo operations (i.e., removing pieces)*, (2) spawns a task for the half of the remaining iterations *with making a copy of the temporarily restored board*, (3) returns from backtracking *with performing redo operations (i.e., setting pieces)*, and (4) resumes its own computation.

ことが考えらえる．すなわち，各ワーカが最外ループの全ての反復を自分で計算するか反復の一部をタスクとして生成するかを，何らかの基準（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）で判断する．効率良い負荷分散のためには，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）生成を行わないような判断基準が最善である．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）がない限り実現することは不可能である．

我々の提案手法では，**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．すなわち，ワーカは，常に**最初は**「タスクを生成しない」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．そのため，**図 2** 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，
(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）最も大きいタスクを生成することができる．

逐次の Pentomino では，ワーカはステップごとにピースを置いたり取り除い（undo）たりするための**作業空間**を持つ．よって，タスク生成の際には，新しいタスク用の作業空間を allocate した上で，そこに**過去の選択時点における**状態をコピーする必要がある．そこで，ワーカは図 2 に示すように，

(1) のバックトラックの際に **undo する**ことにより過去の内容を復元し，
(2) のタスク生成時に復元した作業空間をコピーした後，
(3) のバックトラックからの復帰時に **redo する**ことで
(4) における自らの計算の再開を可能にする．

Cilk[1] や MultiLisp[2] では load-based inlining（本質的には上で説明した「素朴な」手法）の問題を解決するために，Lazy Task Creation (LTC)[5] と呼ばれる手法を採用している．本手法は LTC に対して，

- 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．
- 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．
- 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．
- （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．

2.2 Tascell フレームワーク

Tascell フレームワークは，上記の提案手法を実現するために実装したランタイムおよび Tascell 言語コンパイラからなるフレームワークである．

図 3 に，Tascell フレームワークにおけるプログラムのコンパイル，実行の様子を示す．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．この仕様

☆ 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新規に準備可能な作業空間の場合を除くとできない．

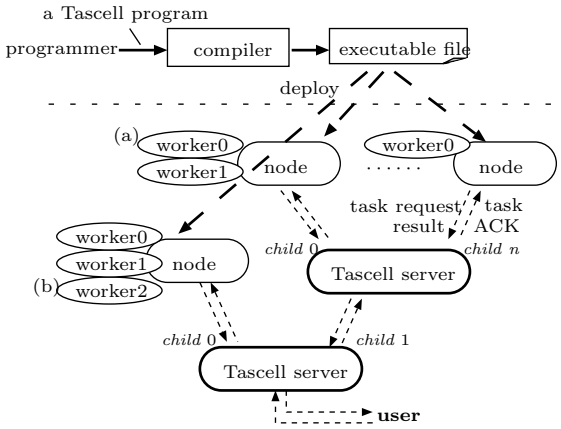


図 3 Tascell フレームワークにおけるプログラムのコンパイル・実行
Fig. 3 Multistage overview of the Tascell framework.

によって，クラスタの代表ノードのみが直接外部と通信可能な環境にも対応できる．また，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に複数のサーバに負荷を分散させることもできる．

ワーカ間のタスクやその結果の授受は**タスクオブジェクト**を介して行われる．タスクオブジェクトは，タスクの入出力の値をセットするためのメンバを持つ構造体であり，その具体的な構造は Tascell プログラムで定義される．オブジェクトの受け渡しは，共有メモリ内のワーカ間であればポインタの受け渡しで行い，ノードを跨る場合はオブジェクトをシリアライズして Tascell サーバを介して送信する．

Tascell 言語は，タスク分割可能箇所の指定や，一時的 undo および redo の処理を記述できるようにした拡張 C 言語である．プログラマは**図 4** のように，既存の逐次プログラムをベースにして，Tascell ワーカのプログラムを書くことができる．

Tascell コンパイラは Tascell 言語から XC-cube 言語[7),9)] への変換器として実装した．XC-cube は我々が提案している拡張 C 言語であり，拡張の一つとして，入れ子関数（関数の中で定義する関数）を定義することで “L-closure” あるいは “Closure” と呼ぶ軽量なクロージャを生成するための機能を備えている．この機能を用いることで，Tascell ワーカは実行スタックの底で「眠っている」フレームにアクセスする（stack walk）ことができ，タスクの要求時生成や undo/redo 処理を実現できる（詳細な実現方法は文献 3) を参照されたい）．

入れ子関数は GCC も C 言語への拡張として提供しているが，L-closure，Closure は機能に一部制限を課すことでクロージャの生成・維持コストを削減している．L-closure は Closure より大きな呼び出しコストを

```
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
   }
  }
 handles pentomino (int i1, int i2) // this の宣言および
           // 範囲（i1-i2）の設定は暗黙になされる
  {
   // 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 4 Pentomino の Tascell プログラム
Fig. 4 Tascell Program for Pentomino.

受け入れる代わりに，コンパイラによるレジスタへの変数割当を妨害しないなど，さらに小さな維持コストを実現している．この維持コストの削減効果は，レジスタを多く持つアーキテクチャにおいては大きく，そうでないアーキテクチャでは小さくなる．Tascell におけるクロージャの呼び出しはタスク生成のたびに発生し，その頻度は実行する Tascell プログラムにも左右されるが，総合的には，SPARC においては L-closure，IA-32 や AMD64 においては Closure を採用したほうが良い性能を示すことが多い．SPARC における比較はこの後の性能評価の節で具体的に示す．

なお，XC-cube 言語の実装はアーキテクチャごとに GCC コンパイラを改造して行っているが，L-closure と Closure に関しては（やや性能を犠牲にするが）標準 C 言語への変換による実装も行っている[4] ため，C コンパイラを使える環境であれば Tascell の利用は可能である．

3. Tascell におけるタスクスティーリング

Tascell におけるワーカ間のタスクスティーリングは以下のように行われる．

- 各ワーカは，自らが実行する残りタスクをエントリとする「タスクスタック」を持つ．
- タスクスタックが空であるワーカは，タスク要求メッセージを（特定の宛先を指定せずに）出し，応答を待つ．
- 要求メッセージは，まず同一ノード内の他の適当なワーカに送られる．受信したワーカは，可能であれば自らが計算中のタスクを図 2 の要領で分割することで新たなタスクを生成し，返信する．タスク生成が不可能であれば（自分のタスクスタックも空であるなど），ノード内の別のワーカに要求を転送する．
- ノード内のどのワーカもタスクを返せなかった場合は，自分が接続している Tascell サーバに要求を転送する．ただし，1 つの計算ノードからは同時に最大 1 つの要求メッセージしか出さないよう抑制している☆．
- 要求メッセージを受信した Tascell サーバは，別の接続しているノード（計算ノード，親・子サーバ）から，タスクを持たないことが確実であるもの☆☆を除外し，その残りからランダムに選択したノードに要求を転送する．転送できるノードが 1 つもなければ，以降の状況変化で転送できるノードが現れるまでメッセージを保留する（この時点で拒否メッセージを返すことも可能だが，再び要求メッセージが発行されるだけであり，無駄な通信が発生してしまう）．
- 外部からのタスク要求メッセージを受け取った計算ノードは，ノード内のワーカに順に問い合わせ，タスク生成が可能であればタスクを返信する．どのワーカも生成が不可能であれば拒否メッセージを返信する．拒否メッセージは Tascell サーバを経由し，もとのタスク要求ワーカに送られる．
- 拒否メッセージを受け取ったワーカは，しばらく sleep した後，再び要求メッセージを出す．
- 要求が受理され，タスクを受け取れたワーカはそれをタスクスタックに push し，計算を開始する．
- 計算中，他のワーカに依頼したタスクの計算結果の待ち合わせが発生した場合，単に待つことはせずに，再びタスク要求メッセージを出す．ただし，このときの要求先は，待ち合わせの原因となるタ

☆ この直後に説明する「盗み返し」は除く．
☆☆ 具体的には，タスクを送った回数とタスクの計算結果メッセージを受け取った回数が同数のノードは，確実に実行中のタスクを持たないと判定している．

表 2 chiba クラスタとその他の拠点のクラスタ間のバンド幅およびラウンドトリップタイム
Table 2 Inter-cluster bandwidths and round-trip times between chiba and other clusters.

拠点	バンド幅（Mbps）	RTT（ms）
hongo	1390	4.57
mirai	65.9	26.9
kobe	387	20.8
keio	94.5	7.77

スクを送信した先に限定する（盗み返し）[☆]．タスクを受信できれば，タスクスタック（の待ち合わせ中のタスクの上）にそれを push し計算を新たに開始する．

- 1 つのタスクの計算が完了したら，その結果を計算結果メッセージとしてタスクの送信元に返信し，タスクスタックを pop する．まだスタックにタスクが残っていれば（つまり，先ほど終了したタスクが盗み返しによるものであれば），待ち合わせ状態が完了しているかを確認し，そうであればそのタスクを再開する．スタックが空，あるいは待ち合わせ状態が完了していない場合は，再びタスク要求メッセージを出す．

一般的なタスクスティーリングシステム（たとえば Cilk）とは異なり，各ワーカは要求を受けてから初めてタスクを生成するため，いわゆるタスクキュー（送信可能なタスクのキュー）は持たない．そのため，単にタスクキューのサイズからワーカ／計算ノードの負荷を見積もるということはできない．ワーカは計算中のタスクを持っていれば，(特に木の並列探索のようなアプリケーションにおいては）ほぼ際限なくタスクを生成することが可能であるが，その反面，特定のワーカに要求が集中して細粒度のタスクが多く生成されてしまう危険性もある．計算中のタスクの分割履歴などから負荷を推定し，サーバのタスク要求転送先の決定に利用する仕組みも簡易的に実装しているが，少なくとも単一サーバと複数ノードにおける並列計算においては目立った性能変化が見られないため，現状ではランダムに転送先を決定する戦略を採用している．

4. 性 能 評 価

InTrigger のうち，chiba，hongo, mirai, kobe，keio の最大 5 拠点 36 ノードを使用した性能評価，および SPARC Niagara 2 プロセッサを 2 台を備えた計算サーバにおける性能評価を行った．Niagara 2 サーバにおいては，Cilk との比較[☆☆]，および Closure と L-closure 2 種類の入れ子関数[7),9)] による Tascell コンパイラ実装の比較も行った．評価に用いたハードウェア，コンパイラの詳細を**表 1** にまとめる．また，InTrigger での評価におけるクラスタ間の接続図を**図 5** に，クラスタ間のネットワークのバンド幅およびラウンドトリップタイム（それぞれ iperf，ping で測定した値）を**表 2** に示す．InTrigger の評価においては，chiba（14 ノード），hongo（4 ノード）, mirai（4 ノード）, kobe（5 ノード），keio（9 ノード）の順で計算に参加するクラスタを追加していった．

ベンチマークプログラムには説明で用いた Pentomino(n) のほかに，n 番目の Fibonacci 数を doubly recursive に求めるプログラム（Fib(n)），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 は，Pentomino と同様，並列 `for` と `dynamic_wind` の組み合わせによる実装である．LU と Comp は，cache-oblivious な再帰アルゴリズムで実装されている．つまり，たとえば Comp においては，サイズ n と m $(n \geq m)$ の配列を比較する Comp タスクが，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，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，全てのアプリケーションにおいて，粒度を上げるための恣意的な閾値を設定しない細粒度並列の実装を行っている．

4.1 Niagara 2 での評価

測定結果の基づく Niagara 2 サーバにおける速度向上のグラフを**図 6** に示す．

逐次性能の Cilk との比較については，既存の性能評価[3),10)] と同じ傾向が確認できる．すなわち，Tascell は Cilk に比べて論理スレッドの生成・管理コストが存在しないことに加え，1 つの作業空間を再利用し続けることによる参照局所性の向上（Nqueens，Pentomino，Grav），また作業空間のコピーの手間が省けること（Nqueens，Pentomino）によって，よりよい逐次性能を得ることができる．

逐次性能の差がそのまま並列計算の結果にも反映され，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 128 並列の計算では，Tascell は Cilk に対して 4.51 倍（＝16.1s/3.57s）の速度向上を達成している．

また，Closure 版より L-closure 版の Tascell コンパイラのほうがほとんどのベンチマークにおいて良い性能を示している．これは，L-closure 版のほうが Tascell のバックトラックの実現のために利用している入れ子関数の生成・維持コストが小さいためである．

ワーカ数増加に伴う速度向上については，Fib や

[☆] この限定を行わないと，ワーカの実行スタックのサイズが膨れ上がる可能性がある[6)]．

[☆☆] 標準の Cilk は共有メモリ環境しかサポートしていないため，InTrigger においては Tascell のみの評価を行った．

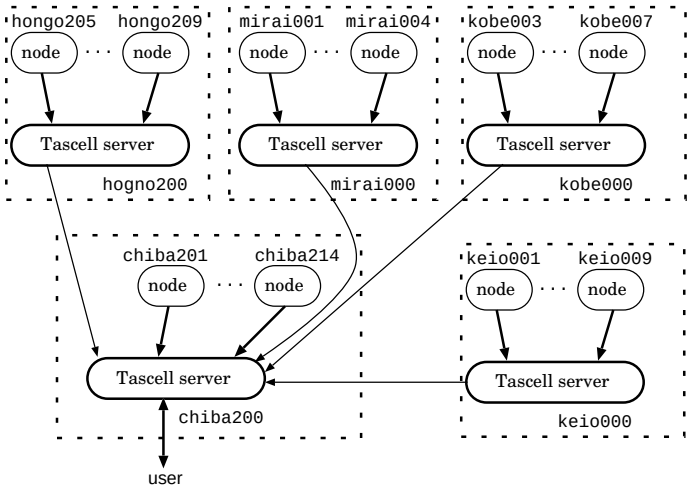
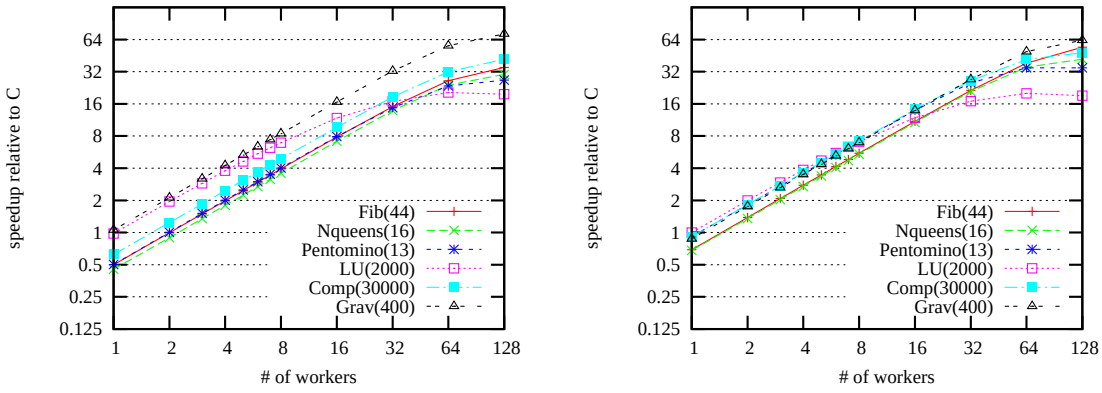
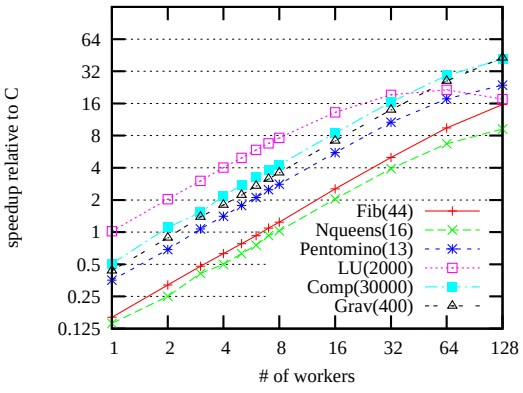


図 5 性能測定時の InTrigger クラスタ間の接続.
Fig. 5 Connection among InTrigger clusters for the performance measurements.



(a) Tascell (Closure)

(b) Tascell (L-closure)



(c) Cilk

図 6 Niagara 2 における並列計算の性能測定結果.
Fig. 6 Performance evaluation on Niagara 2.

表 1 評価環境
Table 1 Specifications of evaluation platforms.

	UltraSPARC T2 Plus サーバ	InTrigger chiba，kobe，keio，kyushu クラスタ
プロセッサ	UltraSPARC T2 Plus 1.4GHz 8-core × 2 コアあたり 8 スレッド（計 128 スレッド）	Xeon E5410 2.33GHz Quad-Core × 2
メモリ	24GB	32GB（chiba，hongo） 16GB（mirai，kobe，keio）
OS	SunOS 5.10 (64bit)	Linux 2.6.x (64bit)
コンパイラ （Tascell）	XC-cube（SPARC 32bit GCC 3.4.6 ベース） `-O2 -mcpu=ultrasparc` Closure および L-closure に基づく入れ子関数[7),9)]	XC-cube（X86-64 GCC 3.4.6 ベース） `-O2` Closure に基づく入れ子関数[7),9)]
コンパイラ （Cilk）	Cilk 5.4.6 + SPARC 32bit GCC 4.4.2 `-O2 -mcpu=niagara2`	 —
Tascell サーバ	 —	Steel Bank Common Lisp 1.0.39 `(speed 3) (safety 3) (space 1)`

Grav で顕著に見られるように，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．L-closure 版の入れ子関数の呼び出しコストは Closure 版より大きいため，タスク分割にかかるコストも大きいが，総合的な性能では L-closure 版のほうが上回っている．(全体の実行時間が少ないと，並列化のオーバーヘッドが相対的に大きくなってしまうことも注意する必要がある．)

また，Tascell，Cilk いずれも 64 並列から 128 並列に並列度を上げたときの速度向上があまり得られていない．この原因の 1 つとしては，Niagara 2 のハードウェアマルチスレッディングによる並列化効果の限界が現れてきたことが考えられる．

LU では速度向上の頭打ちが顕著であり，特に 128 並列ではいずれの実装においても実行時間が増加してしまっているが，この原因としてはメモリバンド幅が飽和したことが考えられる．また，Cilk のほうがやや速度向上が得られにくい原因としては，Cilk のワーカがタスクキューに排他的にアクセスするために用いている THE プロコロル[1)] においてメモリバリア命令を用いており，これがメモリシステムに負荷をかけていることが考えられる．

4.2 InTrigger での評価

測定結果に基づく InTrigger における台数効果のグラフを**図 7**，実行時間および速度向上を数値でまとめたものを**表 3** に示す．

InTrigger においては，タスクオブジェクトのサイズに対して計算量が大きいベンチマークのみ評価を行った．それ以外のベンチマーク（LU，Comp）は文献 3)，10) で考察済みの通り速度向上が見込みにくいためである．この解決は今後の課題ではあるが，今回のベンチマーク対象からは省いた．

ワーカあたり十数秒程度以上の計算が必要となるサイズの問題においては，おおむね良好な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，クラスタ間のネットワーク性能による影響は小さいものと考えられる．

また，総クラスタ数の増加にともなって速度向上が鈍くなっているが，その原因の 1 つとして，特定のクラスタに多くのタスク要求が偏り，そのクラスタ内のワーカに十分な粒度のタスクが行き渡らなくなってしまったということが考えられる．それを確かめるため，Pentomino(16) の 288 ワーカでの計算において，各クラスタ間に跨るタスクスティールの回数を計測した．測定結果を**表 4** に示す．この表から，hongo クラスタへのタスクスティールが集中しており，また hongo から外部のクラスタへのタスクスティールも目立って少ないことが確認できる．このことから，hongo クラスタ内部では，ワーカ間で粒度が小さいタスクを取り合う状態になっていると予想される．

測定環境において，hongo はユーザから最初のタスクを受け取るクラスタである．そのため，計算の初期において他クラスタから hongo へのタスクスティールが集中するほか，この最初のタスクが分割可能である限り hongo クラスタ内でタスクが枯渇することがない．このことが，タスクスティール行き先が偏った原因であると考えられる．各ワーカに割り当てられるタスクの粒度を適切に維持するためには，タスクスティールが全てのワーカに均等に行き渡るようにするのがよい．そのためには，たとえば各 Tascell サーバが一様ランダムにタスク要求の転送先を決定するだけでは不十分で，トポロジーやタスクスティール履歴などの情報を利用するよう改良する必要があると考えられる．

5. まとめと今後の課題

本研究では，共有メモリ高並列環境である Niagara 2 サーバと，広域分散環境である InTrigger プラットフォームそれぞれにおいて Tascell の性能評価を行った．Tascell はこれらの高並列環境においても有効にはたらくことは確認できたが，広域分散環境においてより十分な並列化効果を得るためには，どのワーカにタスク要求を出すかの戦略がより重要になる．オーバー

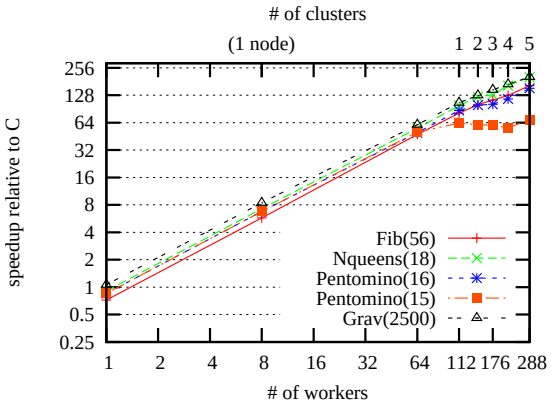


図 7 InTrigger クラスタにおける並列計算の性能測定結果.
Fig. 7 Performance evaluation on InTrigger clusters.

表 3 InTrigger におけるプログラムの実行時間（s）および速度向上
Table 3 The elapsed times (s) and the speedups for the benchmark programs on InTrigger.

# of workers	C	Tascell (Closure)				
	1	1	256	speedup		
	t_S	t_{TC1}	t_{TC288}	t_S/t_{TC1}	t_S/t_{TC288}	t_{TC1}/t_{TC288}
Fib(56)	2835	3763	17.3	0.721	164	218
Nqueens(18)	3574	3933	17.7	0.909	202	222
Pentomino(16)	7304	8564	47.7	0.853	153	180
Pentomino(15)	707	827	10.3	0.855	68.9	80.3
Grav(2500)	4526	4259	22.2	1.06	204	192

表 4 Pentomino(16) の計算におけるクラスタ間のタスクスティール回数
Table 4 The number of task steals among clusters in the execution of Pentomino(16).

thief / victim (ワーカ数)	chiba	hongo	mirai	kobe	keio	スティールされた回数計	ワーカあたり
chiba(112)	—	126	427	207	560	1320	11.8
hongo (32)	618	—	82	110	146	956	29.9
mirai (32)	304	12	—	24	37	377	11.8
kobe (40)	313	27	36	—	69	445	11.1
keio (72)	387	31	47	39	—	504	6.88
スティールした回数計	1622	196	592	380	812	3602	—
ワーカあたり	14.5	6.13	18.5	5.28	11.3	—	12.5

ヘッドのより詳細な分析を行いつつ，上記の戦略について本格的な比較・検討を実施していくことが今後の課題である．また，データ通信にサーバを介さないようにする，MPI を利用するなどより効率的なデータ通信の実装を行うことも予定している．

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」），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）ならびに，科学研究費若手研究（B）「後戻りに基づく動的負荷分散による並列化技法の実用化」（22700030）の助成を得て行った．

また，性能測定に利用させていただいた InTrigger の管理者の方々に感謝いたします．

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, pp. 212–223 (1998).
2) 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).
3) 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), pp. 55–64 (2009).
4) 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 Transactions on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50-67.).
5) 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).
6) 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).
7) 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).
8) 斎藤秀雄, 鴨志田良和, 澤井省吾, 弘中健, 高橋慧, 関谷岳史, 頓楠, 柴田剛志, 横山大作, 田浦健次朗: InTrigger：柔軟な構成変化を考慮した多拠点に渡る分散計算機環境, 情報処理学会研究報告-ハイパフォーマンスコンピューティング（HPC）, Vol. 2007, No. 80, pp. 237–242 (2007).
9) 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.11, No. SIG 1 (PRO 13), pp. 118–132 (2008).
10) 平石拓, 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 T2K 並列環境における評価, 情報処理学会研究報告-ハイパフォーマンスコンピューティング（HPC）, Vol. 2009-HPC-121, No. 7, pp. 1–11 (2009).