Vol.2010-HPC-126 No.25
2010/8/4

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

平石 拓[†1] 河野 卓矢[†2] 八杉 昌宏[†2]
馬谷 誠二[†2] 湯淺 太一[†2]

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．本報告では，そのような環境でどのように負荷分散が実現されているかを，Tascell ワーカ間のメッセージプロトコルを中心に説明する．また本研究では InTrigger のうち最大 4 クラスタを用いた性能評価を行い，256 コアを用いた 18 女王問題の全解探索において逐次の C プログラムに対して 150 倍の速度向上を得ることができた．さらに，共有メモリ高並列環境として，Niagara 2 プロセッサ 2 台を備えたサーバにおける最大 128 スレッドによる性能評価についても報告する．

Evaluation of Backtracking-based Load Balancing in Highly Parallel Environments

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

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 paralellization, 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. This report explains how we support such environments, focusing on the message protocol among Tascell workers. Our 18-queens problem solver achieved a speedup of 150 times compared to its sequential implementation in C, with 256 cores in four InTrigger clusters. We also report a performance evaluation with 128 threads of two Niagara 2 processors as a shared memory highly parallel environment.

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．したがって，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 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

```
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.

1 つの並列計算を行うことも可能である．

本報告では，このように接続された Tascell ワーカ間でどのように動的負荷分散が実現されているかを，タスク要求・応答などのワーカ間のメッセージプロトコルを中心に説明する．

本研究ではまた，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]，および共有メモリ高並列環境である SPARC Niagara 2 を 2 台を備えたサーバにおいて Tascell の性能評価を行った．その結果についても報告する．

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

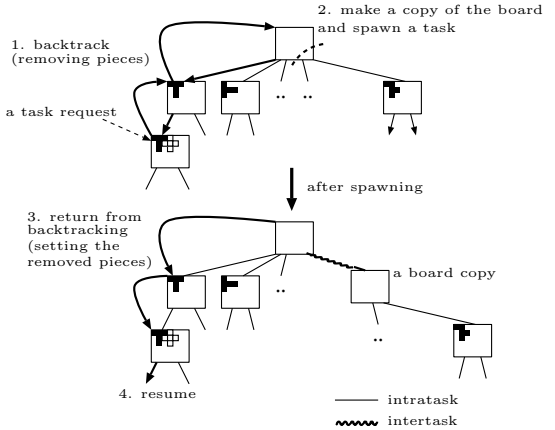


図 2 Pentomino 全解探索におけるタスク遅延生成．Tascell ワーカがタスク要求を（2 つめのピースを置く際に）検出すると，(1) 最古のタスク生成可能時点まで **undo（ピース除去）しつつ**バックトラックし，(2) 反復の半分をタスクとして生成し**（この時，一時的に過去の状態に戻った盤面をコピー）**，(3) **redo（ピースを再設置）しつつ**バックトラックから復帰し，(4) 自らの計算を再開する．
Fig. 2 Spawning a task lazily while performing backtrack search for Pentomino. When 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.

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)[4)] 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．

2.2 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．

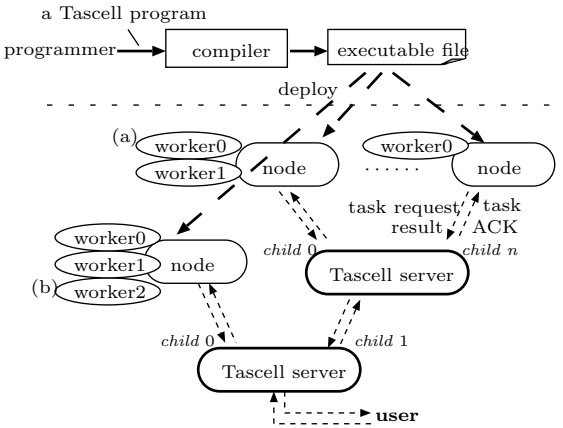


図 3 Tascell フレームワークにおけるプログラムのコンパイル・実行
Fig. 3 Multistage overview of the Tascell framework.

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．ノード間を跨るメッセージは Tascell サーバにより中継される．なお，メッセージはすべて内部的に処理されるものであり，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）．ワーカ間のメッセージプロコトルの詳細は次章で説明する．

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，

⋆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新規に準備可能な作業空間の場合を除くとできない．

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たものをサーバが中継して送信する．

Tascell 言語は，タスク分割可能箇所の指定や，一時的 undo・redo 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 C 言語への変換器として実装した．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 7)，9) で提案した 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．

3. Tascell における動的負荷分散の実現

本章では，Tascell の各ワーカが保持する内部データやワーカ・サーバ間で通信されるメッセージの詳細を示すことで，Tascell におけるワーカ間の負荷分散がどのように実現されているかを説明する．

3.1 ワーカが保持する内部データ

Tascell の各ワーカは，内部データとして以下の種類のスタックおよびキューを保持する．

task スタック　そのワーカ自身が実行するタスクのリスト．各エントリは running（実行中），suspended（他のワーカに渡したサブタスクの結果待ち），done（計算終了）などの状態をとる．

request 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．ただし，一旦要求を受理した後でワーカがタスクを生成できない状態になってしまった場合などは，エントリは subtask スタックに移動せずに破棄される．

subtask スタック　他のワーカに送信したサブタスクのリスト．タスク生成を行った際に，request キューのエントリがこちらに移動する．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（タスクの結果が返信済み）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られる．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から同一のタスクオブジェクトを参照する必要がある．計算ノード内のタスク授受の場合は単に両方のエントリ内に同一のタスクオブジェクトへのポインタをセットし，計算ノード間の授受の場合はそれぞれのノードでタスクオブジェクトの実体を生

```
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, i2, 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.

成し，タスクや結果の送受信時に必要なデータの同期をとることでこれを実現している．

なお，Tascell は他のワーカから要求を受けて初めてタスクを生成する方式を採用しているため，Cilk などの LTC 実装を含むワークスティーリング方式の一般的な実装とは異なり「他のワーカがスティール可能なタスクのキュー」（task キュー）に相当する内部データは存在しない．

3.2 ワーカ・サーバ間のメッセージ

Tascell ワーカ・サーバ間で通信されるメッセージの主な種類は以下の通りである．

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）．any 要求が存在するにも関わらず宛先を明示的に指定した treq メッセージも必要な理由は後で説明する．このメッセージを受け取ったワーカは次のいずれかの処理を行う．

- その要求を（一旦）受理し，ワーカの request キューにエントリを追加する．
- その要求を拒否し，要求元ワーカに `none` メッセージ（後述）を返す．
- その要求メッセージを別のワーカに転送する（any 要求の場合のみ）．

task メッセージ 別のワーカにタスクを送信するメッセージ．ワーカは，request キューにエントリがある状態でポーリングを行う[⋆1] と，バックトラックを行ってそのリクエストに対するタスクオブジェクトを生成し，`task` メッセージにより送信する．メッセージの送信後，request キューの当該エントリは subtask スタックに移動する．task 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あり，その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，1 つのワーカが生成したサブタスク間でユニークな ID（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（図 4 の例では `pentomino` タスクしか存在しないが，一般には複数のタスク型が 1 つのプログラム内に存在し得る）．⟨データ⟩ はそのタスクの入力の値である．このメッセージを受け取ったワーカは，task スタックにエントリを追加し，⟨データ⟩ の値でタスクオブジェクトを初期化した後，そのタスクを実行する．

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．一旦受理して request キューに登録されたタスク要求を後から拒否することになった場合にも送信される．結果として，`treq` メッセージに対しては `task` か `none` のいずれかのメッセージが返信されることが保証される．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） タスクの結果を送信するためのメッセージである．ワーカは，`task` メッセージで受け取ったタスクの計算を完了するとこのメッセージを送信し，task スタックからエントリを取り除く．（その後の task スタックのトップにまだタスクのエントリが存在し，かつサブタスクの結果待ち状態でなければそのタスクを再開する．そうでなければ `treq` メッセージにより仕事の獲得を試みる．）`rslt` 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⟩ がそのタスクの出力値（結果）である．このメッセージを受け取ったワーカは，subtask スタック内の当該エントリから参照されているタスクオブジェクトに受け取った ⟨データ⟩ の値を書き込み，そのエントリの状態を done にする．

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．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` を返す）．この仕組みは不必要なタスク分割を防ぐためのものだが，詳細は後で説明する．

3.3 宛先ワーカの指定

各メッセージに含まれる ⟨送信元アドレス⟩ や ⟨送信先アドレス⟩ は，計算に参加している

⋆1 図 4 のプログラムでは並列 `for` 文を実行する前にポーリングが行われる．

全計算ノードの中からワーカを特定できなければならない．これは，アドレスを送信元から送信先への相対アドレスとして指定することで実現している．具体的には，Tascell におけるアドレスは整数または ‘p’ を ‘:’ で区切った文字列である．整数 n は，Tascell サーバにおいては n 番目の子ノード（計算ノードまたは子サーバ），計算ノードにおいては n 番目のワーカを意味し，‘p’ は自分が接続している親サーバを意味する．

メッセージを中継する Tascell サーバは，受け取ったメッセージの ⟨送信元アドレス⟩ や ⟨送信先アドレス⟩ を適切に加工してから次の送信先に転送する．たとえば，図 3 において (a) のワーカが (b) のワーカに treq メッセージを送信する場合のメッセージ処理は以下のように行われる．

1. ワーカ (a) が接続している Tascell サーバに `treq 0 p:0:2` を送信.
2. 受け取った Tascell サーバはその親サーバに `treq 0:0 0:2` を送信.
3. 受け取った Tascell サーバは 0 番目の計算ノードに `treq 1:0:0 2` を送信.
4. 受け取った計算ノードは `treq` の宛先が,（2 番目のワーカである）ワーカ (b) であり，送信元が（そのメッセージの直接の送信元である Tascell サーバから見た相対アドレスが `1:0:0` である）ワーカ (a) であることを認識できる.

3.4 実行スタックの上限の保証

Tascell ワーカがアイドルになり `treq` メッセージを送信するのは，以下の 2 つのいずれか場合である.

- task スタックが空になった場合.
- 別のワーカに送信したタスクの結果を受け取らないと，実行中のタスクを先に進められなくなった場合（結果待ち状態）.

前者の場合，Tascell ワーカは any 要求を送信する．しかし，結果待ち状態になったことにより `treq` メッセージを送信する場合は，any 要求ではなく，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送信先に指定して `treq` メッセージを送る（取り返し）.

結果待ちの際のタスク要求を取り返しに限定する理由は，ワーカの実行スタックの上限を保証するためである．結果待ち状態のワーカは，残りの計算に必要な情報を実行スタックに残しており，この状態から新たに大きなタスクを獲得して，その後さらに結果待ちになり新たに大きなタスクを獲得する，ということが繰り返されると，実行スタックのサイズが膨れ上がってしまう.

タスク要求を取り返しに限定することで，獲得できるタスクは必ず，自らが実行していたタスクのサブタスクに由来するものとなり，そのサイズを抑えることができる．この手法は，本質的には Leapfrogging[6] と同等であり，実行スタックの最大サイズは逐次実行における最大の実行スタックサイズの定数倍以下になることが保証される.

ところで，取り返しの `treq` メッセージの送信とほぼ同時に，取り返し先で仕事待ちの原因となっているタスクが終了した場合，タイミングによっては `treq` メッセージと `rslt` メッセージが入れ違いになることが有り得る．この `rslt` メッセージを受け取ったワーカは，結果待ち状態を抜けることができるため，先ほど送信したタスク要求はもはや必要ではなく，取り返し先のワーカは `none` メッセージを返信することが期待される．これを実現するため，`rslt` を受け取ったワーカは，今後取り返しの `treq` メッセージを送らないことを伝えるために `rack` メッセージを返信し，一方 `rslt` を送信したワーカは `rack` メッセージを受け取るまで全ての `treq` メッセージを拒否することとしている．[⋆1]

3.5 any タスク要求の処理

計算を正常に終了させるというのみの観点では，`treq` メッセージの any 要求をどのワーカに転送するかは，そのメッセージが有限時間内に処理されることが保証される限りにおいては自由である（タスクを持たないワーカに送ってしまっても `none` メッセージが返却され，再び `treq` メッセージが送信されることになる）．しかし性能面においてはどのように要求先を決定するかは，特に広域分散環境においては非常に重要であり，様々な戦略が考えられる．基本的には，なるべく粒度の大きいタスクを近くのワーカから獲得できるような戦略がよい.

現在の Tascell の実装では，以下のような戦略を採用している.

- any 要求を出すワーカは，最初に計算ノード内の別のワーカからのタスクの獲得を試みる．ノード内のどのワーカに最初に要求を出すかは,
  - ワーカ番号順
  - ランダムに選択

  の 2 つの戦略を要求を出すたびに交互に用いる.
- ノード内にタスクを生成できるワーカがいない場合，0 番ワーカが代表して接続している Tascell サーバに any 要求を送信する．また，この状態で受け取った外部からの any 要求に対しては `none` を返す.
- Tascell サーバは，親サーバに接続している場合，1/(子ノードの数 + 1) の確率で any 要求を親に転送する．親サーバに転送しなかった場合,

⋆1 この戦略が期待通りにはたらくためには，メッセージが送信した順序で受信されることを保証する必要がある.

– ランダム
– 最後に送信した `task` メッセージの ⟨分割回数⟩ が最も小さい子ノード

のいずれかの戦略（サーバ起動時に選択）で転送先の子ノードを決定する．ただし，

– `treq` メッセージの送信元
– `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

には，親サーバ・子ノードに関わらずメッセージを転送しない．後者は，該当するノードにはタスクが存在しないことが明らかなためである．

- 例外として，ルートの Tascell サーバも最初の 1 回のみ any 要求を親（＝ユーザインタフェース）に転送する（最初のタスクはユーザから送信してもらう必要があるため）．

この戦略では，なるべく近くのワーカからタスクを獲得でき，遠くのワーカにしか大きなタスクがない場合にもある程度対応できると考えられる．

4. 性能評価

InTrigger のうち，chiba，kobe，keio，kyushu の 4 拠点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[⋆1]，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，kobe，keio，kyushu の順で計算に参加するクラスタを追加していった．

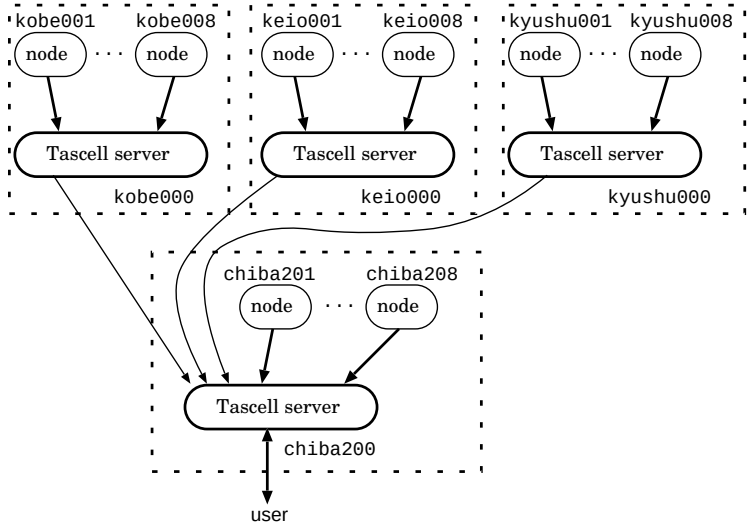


図 5 性能測定時の InTrigger クラスタ間の接続．
Fig. 5 Connection among InTrigger clusters for the performance measurements.

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 クラスタ） 16GB（その他）
OS	SunOS 5.10 (64bit)	Linux 2.6.9 (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)` any 要求に対する子ノードの決定戦略は「ランダム」

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.

	kobe	keio	kyushu
バンド幅（Mbps）	570	94.5	113
RTT（ms）	17.7	7.77	26.7

⋆1 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．

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，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．また，全てのアプリケーションにおいて，粒度を増やすための恣意的な閾値を追加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

Niagara 2サーバにおける並列化効果のグラフを**図6**，実行時間と速度向上の値をまとめたものを**表3**に示す．

逐次性能のCilkとの比較については，これまでに行った評価[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考慮する必要がある．）

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．これは，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ためであると考えられる．

LUでは並列化効果の制限が顕著であり，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．この原因はメモリバンド幅の飽和であると考えらえる．Cilkのほうがやや速度向上が得られにくいのは，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担を強いているためであると考えられる．

4.2 InTriggerでの評価

InTriggerにおける並列化効果のグラフを**図7**，実行時間と速度向上の数値をまとめたものを**表4**に示す．

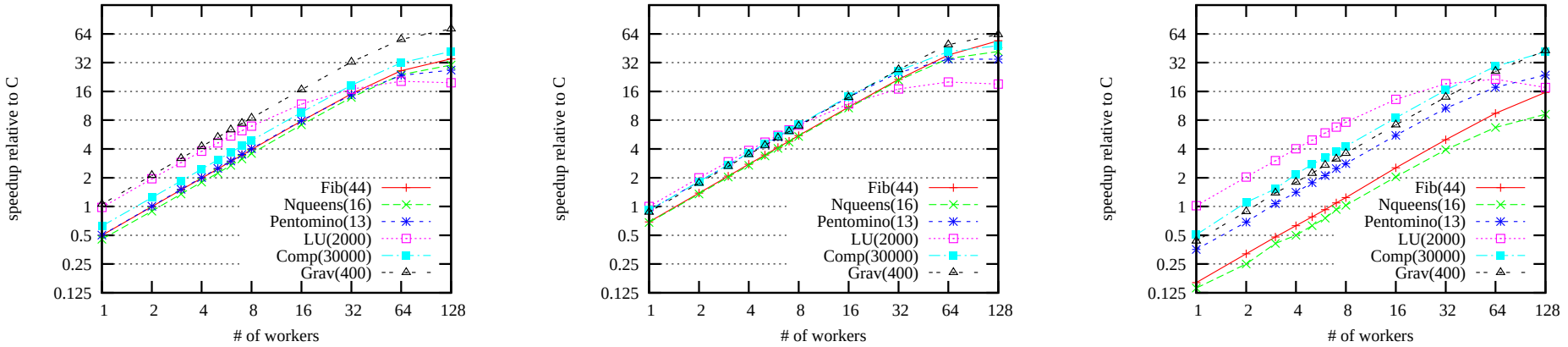
InTriggerにおいては，計算量に対してタスクオブジェクトのサイズに対して大きいベンチマークのみ評価を行った．それ以外のベンチマーク（LU，Comp）は文献3)，10)で考察済みの通り速度向上が見込みにくいためである．この解決は今後の課題ではあるが，今回は評価を見送った．

クラスタ数が1から3までの間は十分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．しかし，クラスタ数を3から4に増加させたときの速度向上はやや小さくなっている．この原因としては，図5のワーカ・サーバの接続関係において3.5節で挙げたanyタスク要求の戦略を適用した結果，chibaクラスタ内のワーカにタスク要求が集中してしまい，このクラスタ内のワーカに十分な量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．

また，Pentomino(16)では速度向上が得られているが，Pentomino(15)では3クラスタ以降で速度向上が得られなくなっていることから，速度向上を得るためには少なくともワーカあたり20秒〜30秒程度のサイズのタスクを与える必要があることがわかる．

5. まとめと今後の課題

本報告では，我々が提案しているバックトラックに基づく負荷分散フレームワークTascellにおいて，ワーカ間の負荷分散がどのように実現されているかを説明した．また，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



(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 3 Niagara 2 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 Niagara 2.

	C	Tascell (Closure)					Tascell (L-closure)					Cilk				
# of workers	1	1	128	speedup			1	128	speedup			1	128	speedup		
	t_S	t_{TC1}	t_{TC128}	t_S/t_{TC1}	t_S/t_{TC128}	t_{TC1}/t_{TC128}	t_{TL1}	t_{TL128}	t_S/t_{TL1}	t_S/t_{TL128}	t_{TL1}/t_{TL128}	t_{C1}	t_{C128}	t_S/t_{C1}	t_S/t_{C128}	t_{C1}/t_{C128}
Fib(44)	42.7	85.7	1.22	0.498	35.0	70.2	62.0	0.787	0.689	54.3	78.8	268	2.73	0.159	15.6	98.2
Nqueens(16)	149	329	4.94	0.453	30.2	66.6	219	3.57	0.680	41.7	61.3	1069	16.1	0.139	9.25	66.4
Pentomino(13)	30.4	60.6	1.37	0.502	22.2	44.2	33.3	0.873	0.913	34.8	38.1	85.3	1.28	0.356	23.8	66.6
LU(2000)	86.2	88.2	4.37	0.977	19.7	20.2	86.3	4.51	0.999	19.1	19.1	84.4	4.91	1.02	17.6	17.2
Comp(30000)	16.8	26.9	0.402	0.624	41.8	66.9	18.5	0.349	0.908	48.1	53.0	33.2	0.405	0.506	41.5	82.1
Grav(400)	121	114	1.67	0.942	72.5	68.3	138	1.92	0.877	63.0	71.9	277	2.82	0.437	42.9	98.2

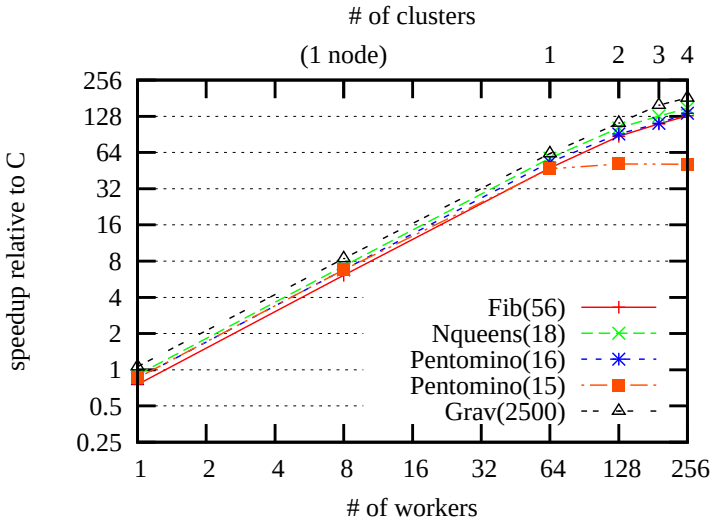


図 7 InTrigger クラスタにおける並列計算の性能測定結果.
Fig. 7 Performance evaluation on InTrigger clusters.

表 4 InTrigger におけるプログラムの実行時間（s）および速度向上
Table 4 The elapsed times (s) and the speedups for the benchmark programs on InTrigger.

	C	Tascell (Closure)				
# of workers	1	1	256	speedup		
	t_S	t_{TC1}	t_{TC256}	t_S/t_{TC1}	t_S/t_{TC256}	t_{TC1}/t_{TC256}
Fib(56)	2835	3763	22.0	0.753	129	171
Nqueens(18)	3574	3933	23.9	0.909	150	165
Pentomino(16)	7304	8564	53.8	0.853	136	159
Pentomino(15)	707	827	13.8	0.855	51.2	59.9
Grav(2500)	4526	4259	24.9	1.06	182	183

においてそれぞれ性能評価を行った.

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．今後は，そのような戦略についてより本格的な比較・検討を実施していくほか，グラファ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」），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.

性能測定や考察においては，InTrigger プラットフォームのほか，GXP[5]，VGXP[11] の各ツールを利用させていただきました．管理者，作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) 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).
5) Taura, K.: GXP : An Interactive Shell for the Grid Environment, *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2004. Proceedings*, pp.59–67 (2004).
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 Legit-

imate 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.SIG1 (PRO13), 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).

11) 鴨志田良和，金田憲二，遠藤敏夫，田浦健次朗，近山　隆：低負荷で多数の計算機をリアルタイムに監視するシステム VGXP の実装，電子情報通信学会技術研究報告. CPSY, コンピュータシステム，Vol.106, No.199, pp.19–24 (2006-07-26).