高速版 Barnes-Hut 多体シミュレーションの並列実装

松井 健[†] 平石 拓[††]
八杉 昌宏[†††] 馬谷 誠二[†]

我々は，多体シミュレーションに用いられる高速版 Barnes-Hut アルゴリズムを，近年提案している動的負荷分散に基づく並列プログラミング言語 Tascell を用いて並列化した．一般に動的負荷分散は不規則なアプリケーションの並列化に対して有効であり，Cilk や X10 などのマルチスレッド言語ではワークスティーリングに基づいて効率的な動的負荷分散を提供する．Barnes-Hut アルゴリズムは多体シミュレーションに広く使用されている．Barnes-Hut アルゴリズムには，単一の作業空間を保持し，それを逐次的に更新する特徴を持った，高速なアルゴリズムが存在する．同時実行可能な各論理スレッドの作業空間を注意深く管理しなければならないマルチスレッド言語とは異なり，論理スレッドフリーなワークスティーリングフレームワークである Tascell では，高速版のアルゴリズムを容易に並列化することができた．実際に 128 ハードウェアスレッド環境上で 100 万の質点に及ぼされる力を計算した結果，我々の Tascell プログラムは通常の手法で並列化された Cilk プログラムと比較して 6.95 倍の高速化を達成することができた．

A Parallel Implementation of a Fast Variant of the Barnes-Hut N-Body Simulation

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

We parallelized a fast variant of the Barnes-Hut algorithm for N-body simulations in an emerging parallel programming language with dynamic load balancing called Tascell. In general, dynamic load balancing is effective in parallelizing irregular applications; multithreaded languages such as Cilk and X10 provide efficient dynamic load balancing based on work-stealing techniques. The Barnes-Hut $O(N \log N)$ tree algorithm is widely used for N-body simulations. There is a fast variant of the Barnes-Hut algorithm that maintains and serially updates a single workspace. Unlike multithreaded languages in which each concurrently runnable logical thread requires a carefully managed workspace, our logical-thread-free work-stealing framework Tascell enables us to straightforwardly and successfully parallelize the fast variant. In fact, our Tascell program calculates forces exerted on 1,000,000 bodies 6.95 times faster than a conventionally parallelized Cilk program on a system with two 64-threaded processors.

1. はじめに

マルチコアプロセッサなどを含む並列計算環境が一般的になるに伴い，並列言語の重要性が高まっている．並列言語の目標の 1 つは，幅広いアプリケーションについて高い並列化効果を達成することにある．バックトラック探索などの不規則なアプリケーションでは，タスク全体を同時実行可能な等しい粒度のタスクに静的に分割することが難しい．一般にそのようなアプリケーションの並列化には動的負荷分散が有効であり，Cilk[1] や X10[2] などのマルチスレッド言語はワークスティーリングに基づいて効率的な動的負荷分散を提供する．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並列プログラミング/実行フレームワーク Tascell[3] を提案している．Tascell ワーカは他のワーカからタスクを要求されない限り逐次計算を続けるが，タスクを要求され

† 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
†† 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
††† 九州工業大学大学院情報工学研究院
Department of Artificial Intelligence, Kyushu Institute of Technology

ると一時的なバックトラックによって最古のタスク生成可能状態を復元し，本物のタスクを生成して要求元のワーカに受け渡す．この手法は，論理スレッドの生成や管理に伴うコストを不要とする，同一の作業空間の長期にわたる再利用を可能とし参照局所性を改善する，作業空間の複製を必要があるまで遅延させる，といった利点を持つ．また，Tascell フレームワークは単一のプログラムを共有メモリ環境と分散メモリ環境の双方において実行させることができ，PC クラスタや広域分散環境における性能評価においても，期待される性能が得られることを確認している．[3)4)]

本稿では，重力多体問題における Barnes-Hut アルゴリズム[5)] の Tascell フレームワークを用いた並列実装を提案し，その性能評価の結果を示す．Barnes-Hut アルゴリズムは，多体問題の解を高速に求めるアルゴリズムの 1 つとして知られるが，このアルゴリズムには，力の計算中に単一の作業空間を逐次的に更新する特徴を持った高速版[6)] が存在する．このアルゴリズムを並列化する場合，マルチスレッド言語では各論理スレッドに割り当てる作業空間を注意深く管理しなければ期待される性能が得られないのに対し，我々は Tascell フレームワークを用いることによって，マルチスレッド言語の 1 つである Cilk よりも容易に且つ効率的に各ワーカの作業空間を管理し，良好な性能向上を達成することができた．

本稿の構成は以下の通りである．まず，第 2 章で Tascell 及び Cilk の概要を述べた上で，第 3 章で Barnes-Hut アルゴリズムの逐次実装について触れる．次に，第 4 章で Tascell 及び Cilk を用いた Barnes-Hut アルゴリズムの並列実装を示し，第 5 章で性能評価の結果を示す．第 6 章で関連研究について述べ，最後に第 7 章でまとめる．

2. 並列言語

2.1 Cilk

Cilk では，Cilk procedure と呼ばれる並列実行可能な関数を定義することができ，Cilk procedure の呼び出しの前に `spawn` キーワードを付加することで実行時に新たな論理スレッドが生成され，その Cilk procedure を並列実行することができる．論理スレッドは child-first で実行される．すなわち，生成された子スレッドは直ちに実行され，親スレッドの継続が動的負荷分散のためにスティール可能となる．また，ユーザは Cilk procedure 内で生成されたすべての子スレッドの完了を待つために `sync` キーワードを使用することができる．

Cilk プログラムでは，一般に並列実行が可能な多数の論理スレッドを生成する．通常，各論理スレッドには計算に必要な作業空間を割り当てる必要がある．そのため，マルチスレッド言語を用いてあるアルゴリズムを並列化する際には，それぞれの作業空間が他の作業空間とは独立して使用でき，論理スレッドが実行を終えた後に自身の作業空間を直ちに解放することが可能となるようにすることが望ましい．しかし，アルゴリズムによっては単一の作業空間のみを使用し，それを逐次的に更新するようなものも存在する．例えば，バックトラック探索では親節点での作業空間を部分的に保存した上で子節点においても同じ作業空間を使い続けることがしばしばある．そのようなアルゴリズムをマルチスレッド言語で並列化する場合は，論理スレッドの生成時に親スレッドの作業空間を子スレッドに複製する必要があり，多数の作業空間の複製が作られることによって性能向上が妨げられることがある．この問題は，親子スレッド間で可能な限り同一の作業空間を再利用することで回避可能であるが，各論理スレッドの作業空間をユーザが注意深く管理する必要が生じる．

2.2 Tascell

本稿では，Tascell の概要のみを述べる．詳細は文献 3) 4) を参照されたい．なお，Tascell は分散メモリ環境における並列計算にも対応しているが，本稿では共有メモリ環境における並列計算のみを扱うこととする．☆

Tascell における計算は，タスクを実行する Tascell ワーカによって行われる．Tascell ワーカはタスクを受け取ることによって，そのタスクが表す計算を実行する．また，Tascell ワーカは他のワーカにタスクを要求することができる．Tascell では，アイドルなワーカがタスクを持っているワーカからタスクの一部を横取りするワークスティーリングによって動的負荷分散を実現する．

Cilk では論理スレッドの生成やスティールの実現手法として Lazy Task Creation[7)] を採用している．これに対し，Tascell ワーカは自身が持つタスクを分割するために一時的なバックトラックを行って最古のタスク生成可能状態を復元する．図 2 は，図 1 に示された行列積の C プログラム☆☆を Tascell を用いて並列化した際に，バックトラックに基づくタスクの分割が

☆ 分散メモリ環境へ対応するためには Tascell 言語でやや煩雑な記述をする必要があるが，本稿ではそれらを省略している．

☆☆ 通常は参照局所性改善のために行列 `B` を転置させるといった前処理を行うが，ここでは特に考えていない．

```
void matmul(matrix A, matrix B, matrix C) {
    if (行列 A と B のサイズが十分に小さい) {
        C = AB を計算する;
    } else if (B の列数 >= A の行数) {
        matrix B1, C1 = B, C の左半分の部分行列;
        matrix B2, C2 = B, C の右半分の部分行列;
        matmul(A, B1, C1);
        matmul(A, B2, C2);
    } else {
        matrix A1, C1 = A, C の上半分の部分行列;
        matrix A2, C2 = A, C の下半分の部分行列;
        matmul(A1, B, C1);
        matmul(A2, B, C2);
    }
}
```

図 1 与えられた行列 A, B の行列積を計算する C 関数の擬似コード．アルゴリズムは分割統治法に基づいている．

Fig. 1 A C function (pseudo code) that performs matrix multiplication for given matrices A and B. The algorithm is based on the divide-and-conquer strategy.

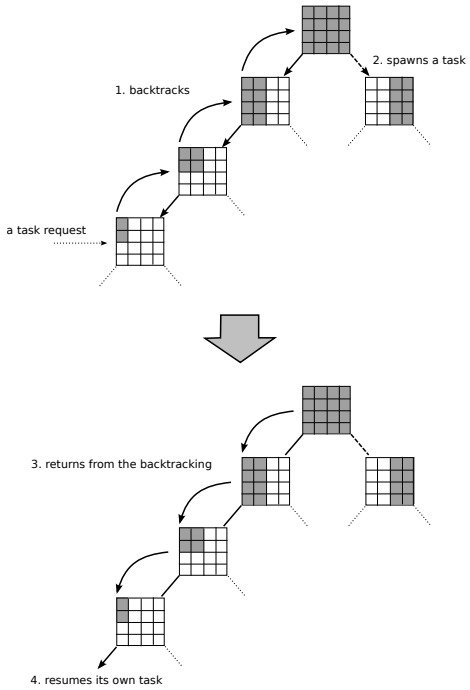

図 2 一時的なバックトラックを利用したタスク生成．格子は計算している行列を表す．Tascell ワーカは計算木の各節点において灰色で示された部分行列の計算を担当する．

Fig. 2 An illustration of task spawning by employing temporary backtracking. The squares represent calculating matrices. A Tascell worker should calculate shaded submatrices in each state.

どのように行われるかを示している．図 2 が示すように，Tascell ワーカは計算中に他のワーカからタスク要求を受信すると，

(1) まず，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分割し，
(3) バックトラックから元の計算状態に戻り，
(4) 自身の行っていた計算を再開する．

つまりワーカは，最初は常にタスクを分割しないことを選択し逐次的に計算を行うが，他のワーカからタスク要求が生じ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．

本手法では潜在的なタスクとしての論理スレッドは生成されず，キューを用いて論理スレッドを管理するためのコストは不要となる．また次に挙げる特徴を活かすことによって，本手法では単一の作業空間を逐次的に更新するアプリケーションを効率的に並列化することができる:

- マルチスレッド言語では各論理スレッドに作業空間を割り当てる必要があるが，本手法ではワーカが逐次計算を行っている間は単一の作業空間を再利用することができる．これにより参照局所性の改善による性能向上が期待できる．
- マルチスレッド言語では子スレッドの生成時に親スレッドの作業空間を複製する必要があるが，本手法では同様の複製をタスク要求が生じるまで遅延させることができる．そのため，マルチスレッド言語による同等の実装と比較して作業空間の複製回数が少なく済ませられる．

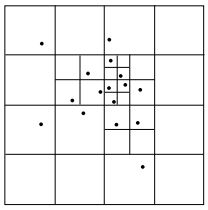

図 3 階層化された空間の分割を図示したもの (簡単のため 2 次元で示している)．黒点は質点を，それらを囲む正方形はセルを表している．

Fig. 3 An illustration of hierarchical boxing (presented in two dimensions for simplicity). Each black dot represents a body, and a bounding box represents a cell.

3. 重力多体問題

3.1 Barnes-Hut アルゴリズム

Barnes-Hut アルゴリズムは近似的な計算によって多体問題の解を高速に求めるアルゴリズムである．現実の科学技術計算では非常に多数の粒子の運動をシミュレートする必要があるため，Barnes-Hut アルゴリズムを用いた多体シミュレーションの高速な実装に関する研究は今日でも広く行われている．

本稿では，特に重力多体問題を考える．質点の数を

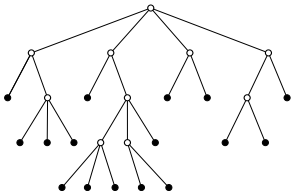

図 4 図 3 の空間構造を表現する木構造．黒と白の節点はそれぞれ質点とセルを表す．
Fig. 4 A hierarchical tree representing the spatial structure in Figure 3. Black and white nodes represent bodies and cells, respectively.

N とすると，ニュートン力学に基づいてそれぞれの質点に作用する万有引力を求めるための計算時間は通常 $O(N^2)$ となる．Barnes-Hut アルゴリズムでは，力の計算を始める前に，空間をセルと呼ばれる部分空間に再帰的に分割し (図 3)，その空間構造を表す木構造を生成する (図 4)．そして，生成された木を探索しながら各々の質点に及ぼされる力を計算する．その際に，質点間の距離が十分に離れている場合は，力を及ぼす複数の質点をセル単位でまとめ，これを 1 つの質点と見なして力の計算を行う．この手法により，Barnes-Hut アルゴリズムは計算量を $O(N \log N)$ に削減している．

3.2 Barnes-Hut アルゴリズムの逐次実装

本研究では重力多体問題の逐次シミュレーションプログラムである treecode 1.4[8] を並列化し，評価を行った．以下ではこの treecode の実装の概要を述べる．

treecode は次の手順を繰り返し実行することで，質点の運動をシミュレートする:

(1) 質点の位置を読み込み，木構造を生成する．
(2) 各質点に及ぼされる力を計算し，質点の加速度とポテンシャルを更新する．
(3) (2) で得られた計算結果を元に，各質点の位置と速度を更新する．

シミュレーションは逐次的に実行されることから，treecode には並列実行時にワーカ間の負荷分散を行うための Orthogonal Recursive Bisection (ORB)[9] のような空間分割手法は実装されていない．

treecode における力の計算には文献 5) で示されているアルゴリズムとはやや異なる実装がなされており，文献 6) で提案されているより高速なアルゴリズムが使用されている．このアルゴリズムは，空間上で近傍に位置する質点は類似した “interaction list” を持つという性質を利用して，木探索のコストを削減する．ここで interaction list とは，ある質点に力を及ぼす全ての質点またはセルのリストである．本来の Barnes-Hut アルゴリズムでは，各々の質点ごとに木探索を行い，質点ごとに固有の interaction list を求めて及ぼされる力を計算する．しかし高速化されたアルゴリズムでは，木探索は 1 度だけ行い，その間単一の interaction list を保持して，それを逐次的に更新する．そして探索が葉，つまり質点に到達した時に，その時点の interaction list を用いてその質点に及ぼされる力を計算する．

```
void gravcalc() {
    list A = 木の根節点を要素とするリスト;
    list I = 空リスト;
    node p = 木の根節点;
    walktree(A, I, p);
}
void walktree(list A, list I, node p) {
    list nextA
      = (リスト A の末尾を先頭ポインタに持つ) 空リスト;
    for (A に含まれる各節点 a) {
        if (a はセルである) {
            a と p の距離を計算する;
            if (距離が十分に離れている)
                a を I に追加する;
            else
                a のすべての子節点を nextA に
                追加する;
        } else if (a != p) {
            a を I に追加する;
        }
    }
    if (nextA に要素が追加されていない)
        I の各節点要素が p に及ぼす力を計算する;
    else
        walksub(nextA, I, p);
}
void walksub(list A, list I, node p) {
    if (p はセルである)
        for (p の各々の子節点 q)
            walktree(A, I, q);
    else
        walktree(A, I, p);
}
```

図 5 treecode 1.4 で利用されている高速な力の計算のアルゴリズムの擬似コード．
Fig. 5 Pseudo code for the fast force calculation algorithm employed in treecode 1.4.

treecode における力の計算の実装の擬似コードを図 5 に示す．力の計算は関数 `gravcalc` を呼び出すことによって始まる．ここで初期化しているリストは，実際には十分な大きさの配列と，使用中の配列要素の先頭と末尾を指すポインタから構成される．リストは以後関数呼び出しの引数として渡されるが，配列の複製が行なわれるわけではないことに注意されたい．

実際の力の計算は 2 つの関数 `walktree` と `walksub` が相互再帰的に呼び出されることによって行われる．関数 `walktree` は節点 `p` の中に含まれるすべての質点に及ぼされる力を計算する．リスト `I` は根節点から節点 `p` に探索が到達するまでに求められた interaction list を，リスト `A` は `p` についてまだ探索が終了していない節点のリストを表す．また，リスト `nextA` はリスト `A` の末尾ポインタ以降に新たに追加される要素を記

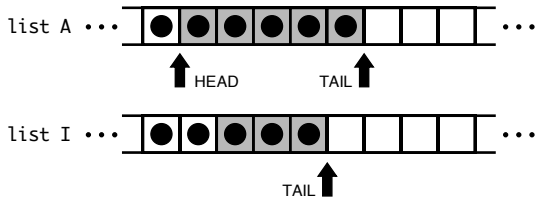

図 6 ある関数 walksub 呼び出しの呼び出し直後のリストの状態．灰色で示された要素は直前の関数 walktree の実行中に追加された節点を表す．子節点の探索中は未使用の配列領域に新たな要素が書き込まれるが，探索終了後には再び使用可能になる．

Fig. 6 Lists right after a function call walksub. Shaded items represent nodes added at the previous function call walktree. Although new items are added to the unused parts of the arrays while traversing child nodes, they will be available again after each traversal.

憶するためのリスト (初期値は空リスト) であり，要素を格納する配列はリスト A と共有している．関数 walktree では，はじめにリスト A の各要素 a に対して p との距離を求め，a を I に追加するか，a の子節点をリスト nextA に追加する．すなわち，a の子節点の探索が不要であるか否かを，距離に基づいて判断している．その後，nextA に要素が追加されていなければその時点で保持している interaction list を用いて p に及ぼされる力を求め，そうでなければさらに探索を続ける．後者の場合は関数 walksub が呼び出され，もし p がセルならばその子節点に及ぼされる力をそれぞれ計算する．

関数 walksub が呼び出された直後のリスト A, I の状態を図 6 に示す．末尾ポインタ以降にある未使用の配列領域は新たな要素を追加するための領域として子節点の探索中に更新されるが，先頭/末尾ポインタの初期位置は関数引数の値として保存されるため，子節点の探索が終了するとこれらのポインタの位置は探索前の状態に戻り，新たな子節点の探索のために再び未使用の配列領域となる．このように，treecode における力の計算の実装は，単一の作業空間を保持し続け，それを逐次的に更新するという特徴を持っている．[8)]

4. 並列実装

この章では，Tascell 及び Cilk を用いた treecode の並列実装を示す．なお，本稿では紙面の都合上，一般に Barnes-Hut アルゴリズムで最も計算時間を要する力の計算の並列実装のみを示すこととする．木構造の生成などその他の並列実装については，別論文で発表する予定である．

```
// タスク twalk の定義
task twalk {
    list A; list I; node p;
    int j1; int j2;
};
// twalk タスクを受信した Tascell ワーカのエントリポイント
// タスクオブジェクト this が暗黙に宣言される
task_exec twalk {
    walksub(this.A, this.I, this.p, this.j1, this.j2);
}
worker void walksub(
  list A, list I, node p, int i1, int i2) {
    if (p はセルである) {
        // Tascell の並列 for 文を利用
        for (int i: i1, i2) {
            node q = p の i 番目の子節点;
            walktree(A, I, q);
        }
        // タスク要求ハンドラ
        // タスクオブジェクト this が暗黙に宣言される
        handles twalk(int j1, int j2) {
            /* "put" 部 (タスクの送信前に実行される) */ {
                リスト A と I を this.A と this.I に
                複製する;
                this.p = p;
                this.j1 = j1;  this.j2 = j2;
            }
            /* "get" 部 (計算結果を受信後に実行される) */ {
                this.A と this.I に割り当てられた
                メモリ領域を解放する;
            }
        }
    }
    else
        walktree(A, I, p);
}
```

図 7 Tascell 言語で書かれた，並列化された力の計算アルゴリズムの擬似コード．

Fig. 7 Pseudo code of the parallelized force calculation algorithm written in the Tascell language.

4.1 Tascell を用いた実装

実際に並列化を行った箇所は，図 5 における関数 walksub の内部にある．関数 walksub では，p がセルの時に p の各々の子節点について関数 walktree を呼び出している．この walktree の反復呼び出しに Tascell 言語の並列 for 文を用いることで，反復の一部をタスクとして分割できるようにした．

図 7 に Tascell 言語による並列化された関数 walksub の擬似コードを示す．まず，木の探索を表すタスクとして twalk という名前のタスクを先頭で定義した．twalk タスクの内部では，関数 walksub を呼び出す際にその引数として与える変数が定義されている．これには，関数 walksub 内部の並列 for 文の開始/終了インデックスを表す変数 j1 と j2 も含まれる．twalk タスクを受信した Tascell ワーカは task_exec を実行し，受信したタスクオブジェクト this に保存された変数を引数として関数 walksub を呼び出す．Tascell の並列 for 文では，Tascell ワーカは他のワーカからタスク要求を受信しない限り並列 for 文の各反復を逐次的に実行するが，タスク要求を受信すると未実行

```
cilk void walksub(list A, list I, node p) {
    if (p はセルである) {
        for (p の各々の子節点 q) {
            list copyA, copyI = 複製されたリスト A 及び I;
            spawn walktree(copyA, copyI, q);
        }
        sync;
        copyA と copyI に割り当てられたメモリを解放する;
    }
    else
        spawn walktree(A, I, p);
}
```

図 8 Cilk 言語で書かれた，並列化された力の計算アルゴリズムの擬似コード.

Fig. 8 Pseudo code of the parallelized force calculation algorithm written in the Cilk language.

の反復の半数をタスクとして分割し，タスク要求元のワーカに送り返す．タスクとして受け渡す反復のインデックスは，タスク要求ハンドラの中で変数として参照することができ，それらは Tascell フレームワークによって自動的に設定される．図 7 では送信するタスクを準備するための “put” 部の実行中に，現在の作業空間であるリスト A, I といった変数とともに，反復のインデックス j1 と j2 を複製してタスク要求元のワーカに送信している．タスク要求ハンドラの “get” 部には，分割されたタスクの計算結果を受信した後で，それを現在のワーカの (中間の) 計算結果に統合するための処理を記述するが，treecode では力の計算結果を大域変数を経由したヒープメモリ領域に書き込むため，図 7 ではメモリ管理の処理のみが記述されている．

ワーカの作業空間であるリスト A と I は，他のワーカからタスク要求が生じた時にのみ複製されることに注意されたい．タスク要求が生じない限り Tascell ワーカは並列 for 文を逐次的に実行するため，タスクの分割は行われず作業空間の複製は発生しない．

4.2 Cilk を用いた実装

Cilk においても，Tascell における実装と同等の並列化を行った．すなわち，for 文中の関数 walktree の呼び出しの前に spawn キーワードを付加することによって，関数 walktree の各呼び出しを並列実行できるようにした．図 8 に Cilk 言語による実装の擬似コードを示す．

マルチスレッド言語を用いて関数 walksub の並列化を行う場合，より性能を向上させるためには論理スレッドに割り当てる作業空間の管理方法についてさらに考慮する必要がある．木の探索中は作業空間内のデータ，すなわちリスト A と I の要素が逐次的に更新されるため，生成された子スレッドは親スレッドの作業空間を再利用するのが効率的である．しかし，同時実行可能な複数の子スレッドが親スレッドの作業空間

```
cilk void walksub2(
  list A, list I, node p, int i1, int i2) {
    if (i2 - i1 == 1)
        spawn walktree(A, I, p の i1 番目の子節点);
    else if (i2 - i1 > 1) {
        spawn walksub2(
          A, I, p, i1, (i1 + i2) / 2);
        if (SYNCHED) {
            spawn walksub2(
              A, I, p, (i1 + i2) / 2, i2);
        } else {
            list copyA, copyI = 複製されたリスト A 及び I;
            spawn walksub2(
              copyA, copyI, p, (i1 + i2) / 2, i2);
            sync;
            copyA と copyI に割り当てられたメモリを
            解放する;
        }
    }
}
```

図 9 SYNCHED 変数と分割統治法を用いて改良された walksub 関数の Cilk における実装.

Fig. 9 An improved Cilk implementation of walksub function using the SYNCHED variable and the divide-and-conquer strategy.

に同時にアクセスすることを避けるため，図 8 の実装では子スレッドを生成するたびに親スレッドの作業空間を毎回複製している．この実装は Tascell の実装と比較して多数の作業空間の複製を発生させ，性能向上を妨げる原因となる．

作業空間の再利用を促進するために，Cilk ではスケジューラ変数 SYNCHED を利用することができる．SYNCHED 変数は，Cilk procedure 内でそれまでに生成されたすべての子スレッドが実行を終えている時にのみ真の値をとる擬似変数である．SYNCHED 変数の値を論理スレッドの生成後に確認することによって，以下のようにして親子スレッド間での作業空間の再利用が可能となる:

- 値が真ならば親スレッドの作業空間を使用する子スレッドは存在しないため，次に生成される子スレッドには親スレッドの作業空間を (複製することなく) 再利用させる．
- 値が偽ならば既にある子スレッドによって親スレッドの作業空間が使用されているとみなし，次に生成される子スレッドには親スレッドの作業空間を複製して割り当てる．

SYNCHED 変数と分割統治法を用いて関数 walksub 内の for ループを並列化した擬似コードを図 9 に示す．

Cilk 以外のマルチスレッド言語は，一般に SYNCHED 変数に相当する機能を持たないことに注意されたい．そのような言語では，例えば mutex 変数を用いるなどの方法で，ユーザ自身で SYNCHED 変数を用いた実装に相当する実装を行わなければならない．

表 1 評価環境
Table 1 Evaluation environments

	Linux PC Server (x86-64)
CPU	Xeon X5650 2.67GHz 6-Core×2 Hyper-threading を有効化 (計 24 スレッド) Turbo Boost を無効化
メモリ	24GB
OS	CentOS 5.4 (64bit)
コンパイラ	Tascell コンパイラ + GCC 4.1.2 Cilk コンパイラ 5.4.6 + GCC 4.4.2 最適化オプション `-O3` を有効化

	UltraSPARC T2 Plus Server (SPARC)
CPU	UltraSPARC T2 Plus 1.4GHz 8-Core×2 コア当たり 8 スレッド (計 128 スレッド)
メモリ	24GB
OS	SunOS 5.10
コンパイラ	Tascell コンパイラ + GCC 4.4.2 Cilk コンパイラ 5.4.6 + GCC 4.4.2 最適化オプション `-O3` を有効化

5. 性能評価

前章で示した並列実装の性能評価の結果を示す．評価を行ったプログラムは次の 3 つである:

Tascell 図 7 の実装に基づく Tascell プログラム．

Cilk 図 8 の実装に基づく，論理スレッドの生成のたびに親スレッドの作業空間を複製する Cilk プログラム．

Cilk (SYNCHED) 図 9 の実装に基づく，`SYNCHED` 変数を用いて作業空間の複製回数を抑制した Cilk プログラム．

これらのプログラムに加え，本研究では 3.1 節で述べた (高速化されていない)Barnes-Hut アルゴリズム[5] を並列化した Cilk プログラムも実装し，評価を行った．この Cilk プログラムは，3.2 節の高速化された Barnes-Hut アルゴリズムが元の Barnes-Hut アルゴリズムよりも高速であることを確認するために，参考として実装したものである．

評価環境は表 1 に示すとおりであり，x86-64 アーキテクチャ及び SPARC アーキテクチャの 2 つの計算機上で評価を行った．これらの計算機上で，ランダムに生成された 100 万の質点☆からなる多体シミュレーションを実行し，逐次版の treecode[8] との計算時間の比較を行った．なお，本稿では第 4 章にて力の計算の並列実装のみを示したため，測定結果についても力の計算の結果のみを掲載することとした．

☆ 逐次版の treecode にテスト用の質点データを生成するための機能が備わっているため，本研究ではそれを用いた．

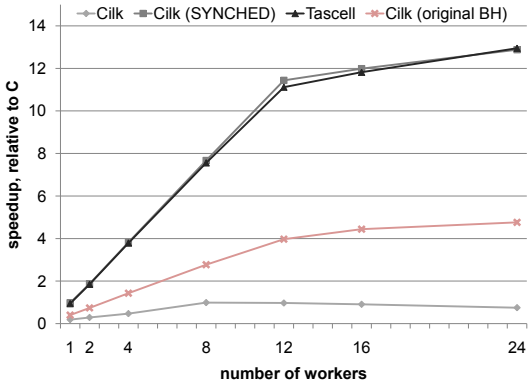

図 10 x86-64 アーキテクチャマシン上で測定された，逐次版の力の計算時間に対する性能向上率．
Fig. 10 Relative speedup to the serial force calculation time measured on the x86-64 architecture machine.

図 11 SPARC アーキテクチャマシン上で測定された，逐次版の力の計算時間に対する性能向上率．
Fig. 11 Relative speedup to the serial force calculation time measured on the SPARC architecture machine.

2 つの計算機上で測定された力の計算の計算時間について，逐次版 treecode からの性能向上率を示した結果を図 10(x86-64) 及び図 11(SPARC) に示す．いずれの計算機上のいずれのワーカ数の結果においても，高速化された Barnes-Hut アルゴリズムを並列化した **Tascell** 及び **Cilk (SYNCHED)** の性能が，元の Barnes-Hut アルゴリズムを並列化した **Cilk (original BH)** を上回っていることが分かる．実際に，**Tascell** は **Cilk (original BH)** と比較して，x86-64 マシン上で最大 2.72 倍，SPARC マシン上で最大 1.73 倍の性能を達成している．

論理スレッドの生成のたびに親スレッドの作業空間を複製する **Cilk** は，どちらの計算機上でも性能が思

うように伸びておらず，**Cilk (original BH)** の性能をも下回ってしまっていることが分かる．対照的に，親子スレッド間で作業空間を可能な限り再利用する **Tascell** 及び **Cilk (SYNCHED)** は良好な性能向上率を達成することができている．実際に，**Tascell** は **Cilk** と比較して，x86-64 マシン上で最大 13.05 倍，SPARC マシン上で最大 6.95 倍の性能を達成している．この結果から，単一の作業空間を逐次的に更新するアルゴリズムをマルチスレッド言語を用いて並列化する際に，期待される性能を得るためには，各論理スレッドの作業空間を適切に管理することが重要であると言える．

いずれの計算機上においても，**Tascell** と **Cilk (SYNCHED)** はほぼ同等の性能向上率を達成している．これは，**Cilk** と比較してどちらのプログラムも作業空間の複製回数を同程度に抑えることができているためと考えられる．ただし，**Tascell** は逐次版の Barnes-Hut アルゴリズムに変更を加えることなくこれらの性能を達成している点に注意されたい．**Cilk (SYNCHED)** では，`SYNCHED` 変数を用いて作業空間の再利用を適切に行うために，逐次版の Barnes-Hut アルゴリズムに変更を加えている．また **Cilk (SYNCHED)** の実装は自然な形で容易に得られるものではなく，**Tascell** の実装を真似る形で作業空間の再利用を試みた結果として得られるという点にも注意されたい．

なお，すべてのプログラムについて，一定のワーカ数以上の設定で実行させると性能向上率の傾きが減少する傾向が見られる (x86-64 マシン上での 16, 24 ワーカ，SPARC マシン上での 64, 128 ワーカなど)．この原因は，同時マルチスレッディングによる性能向上の限界や，メモリバンド幅の飽和などが考えられる．

6. 関 連 研 究

分散メモリ環境における Barnes-Hut アルゴリズムの並列化に関する研究は数多くある．分散メモリ環境では，計算ノード間で効果的な負荷分散を実現するため，一般には空間を計算ノードの数だけ適当な手法で分割し，それぞれの部分空間内の質点に及ぼされる力の計算を各ノードに割り当てることが多い．空間の分割手法としては，再帰的な 2 分割を繰り返す手法[9] や，モートン順序に基づく手法[10] などがある．一方，我々の並列実装ではワークスティーリングに基づく実行フレームワークを用いて動的な負荷分散を行っている．加えて，Tascell フレームワークは単一のプログラムを共有メモリ環境と分散メモリ環境の双方において実行させられることから，本稿で示した並列実装は分散メモリ環境上においても実行可能であり，計算ノード間の負荷分散は Tascell フレームワークによって実現される．

多体問題は，かつてはベクトル型プロセッサや GRAPE などの専用ハードウェアを用いて，計算時間を要する浮動小数点演算を高速に計算する手法が主流であったが，近年では GPGPU を用いた計算手法が主流となりつつある．一方，我々の並列実装における力の計算部では，計算の一部をタスクとして分割して並列化するのみに留まっているため，GPGPU の利用により更なる性能向上が見込まれる．

7. 終 わ り に

本稿では，高速版 Barnes-Hut アルゴリズムを我々の提案しているワークスティーリングフレームワーク Tascell を用いて並列化した実装と，その性能評価の結果を示した．高速版 Barnes-Hut アルゴリズムは力の計算中に単一の作業空間のみを保持し，それを逐次的に更新する特徴を持っており，マルチスレッド言語を用いて並列化する場合は各論理スレッドに割り当てる作業空間を注意深く管理しなければ期待される性能は得られなかった．しかし Tascell では，タスクはアイドルなワーカから要求された時にのみ生成され，ワーカは逐次計算を行っている間単一の作業空間を再利用することができる．こうした論理スレッドフリーの枠組みを用いることにより，我々は高速版 Barnes-Hut アルゴリズムの並列化において，ユーザによる注意深い作業空間の管理を必要とすることなく良好な性能向上を達成することができた．

今後は，GPGPU を利用した高速化や分散メモリ環境における実装の研究を進め，より大規模な多体シミュレーションを実行して評価を行っていく予定である．

謝辞 本研究の一部は，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．

参 考 文 献

1) Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp.212–

223 (1998).
2) IBM Research: X10. http://x10-lang.org/.
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*, pp. 55–64 (2009).
4) 平石拓, 八杉昌宏, 馬谷誠二: 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, 先進的計算基盤システムシンポジウム（SACSIS2011）, pp. 55–63 (2011).
5) Barnes, J. and Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm, *Nature*, Vol. 324, pp. 446–449 (1986).
6) Barnes, J. E.: A Modified Tree Code: Don't Laugh; It Runs, *Journal of Computational Physics*, Vol. 87, pp. 161–170 (1990).
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, pp. 264–280 (1991).
8) Barnes, J. E.: Treecode Guide. http://www.ifa.hawaii.edu/~barnes/treecode/treeguide.html.
9) Warren, M. S. and Salmon, J. K.: Astrophysical N-body simulations using hierarchical tree data structures, *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pp. 570–576 (1992).
10) Warren, M. S. and Salmon, J. K.: A Parallel Hashed Oct-Tree N-Body Algorithm, *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pp. 12–21 (1993).