Vol.2012-HPC-135 No.13
2012/8/1

ExaFMMのタスク並列処理系MassiveThreadsによる並列化とその評価

田浦健次朗[1,a)]　中島潤[1,b)]　横田理央[2,c)]　丸山直也[3,d)]

概要：高速なFMMのオープンソース実装であるExaFMMを, タスク並列処理系MassiveThreadsを用いて並列化する方法, および得られた性能について述べる. FMMには木構造を用い, 再帰呼び出しを用いれば見通しよく記述できる部分が多く有る. そのような処理の並列化を生産性よく行うには, 任意の時点でタスクを生成し, それをスケーラブルに負荷分散できるタスク並列処理系が必須かつ有用である. とくに, タスク並列なしでは並列化が困難な以下の部分に焦点を当てる. (1) 粒子間の相互作用を作用反作用の法則に従うよう (従って全運動量を保存するよう) 計算する, (2) 木構造を生成する. BarcelonaおよびNehalemアーキテクチャ上で評価を行い, オリジナルのC++コードにほとんどオーバーヘッドを加えずに並列化を行うことができた. 計算時間の大部分を占める相互作用の計算に関しては, 50万以上の粒子で, Nehalem (24コア) で90%以上 Barcelona (32コア) で75%以上の並列化効率を得た. 相互作用以外の計算は, 現時点では並列化効率が思わしくなく, そのためすべてのフェーズを含めた合計時間については同条件で, それぞれ50%, 35%程度となった.

キーワード：高速多重極展開法, N体問題, 分割統治法, タスク並列

Parallelizing ExaFMM with MassiveThreads Task Parallel Library and Its Evaluation

Kenjiro Taura[1,a)]　Jun Nakashima[1,b)]　Rio Yokota[2,c)]　Naoya Maruyama[3,d)]

Abstract: This paper describes parallelization of a fast, open source FMM implementation of ExaFMM, using a lightweight task parallel library MassiveThreads and reports its performance. FMM contains many operations that can be elegantly expressed with tree structures and recursions. Task parallelism is a useful tool to facilitate parallelization of such algorithms. In particular, we focus on how to elegantly parallelize operations that are otherwise hard to parallelize; (1) "mutual" interactions obeying Newton's law of action and reaction, thereby preserving the total momentum; (2) tree construction. Experiments are conducted on machines with Barcelona and Nehalem microarchitectures. The overhead due to parallelization was negligible. With 500K or more particles, computation of mutual interactions—the dominantly time consuming phase—achieved 90% utilization on Nehalem (24 cores) and 75% utilization on Barcelona (32 cores). As of writing, howerver, utilization of other phases are so low that the utilization as a whole were about 50% and 35% utilization, respectively, under the same conditions.

Keywords: Fast Multipole Method, N-body Problem, Divide and Conquer Method, Task Parallelism

[1] 東京大学
University of Tokyo, 7-3-1 Hongo Bunkyo-ku, Tokyo 113–0033, Japan
[2] サウジアラビア・アブドラ王立科学技術大学
King Abdullah University of Science and Technology
[3] 計算科学研究機構
RIKEN Advanced Institute of Computational Science, 7-1-26 Minatojima-minamimachi, Chuo-ku, Kobe, Hyogo 650-0047, Japan
[a)] tau@eidos.ic.i.u-tokyo.ac.jp
[b)] nakashima@eidos.ic.i.u-tokyo.ac.jp
[c)] Rio.Yokota@kaust.edu.sa
[d)] nmaruyama@riken.jp

1. はじめに

高速多重極展開法 (Fast Multipole Method; 以下 FMM) [2], [9] は, N 体問題の高速アルゴリズムとして, Treecode [1] と並んでよく用いられる. 実際の応用では, 場所によって粒子の密度が異なることが多く, 密度に応じて空間を異なる大きさのセルに分割する適応的なデータ構造 (木構造) が必要になる [7].

FMM では, ある一定個数以下の粒子からなる粒子群同士の全相互作用を計算することが要素となる処理である. 例えば n 個の粒子と m 個の粒子を全対全相互作用させるには, $O(n+m)$ のデータの参照で, $O(nm)$ の計算を行う. そのため, FMM は計算カーネルの性能が得やすく, 局所性が重要な大規模な並列化にも向いたアルゴリズムであると考えられている [5], [10], [12], [16], [23].

一方で, 木構造に対する計算の並列化, その負荷分散など, 並列プログラミング上の課題は多く, SPMD モデルや, OpenMP の parallel for ループのような, 単純な並列ループだけに基づく並列化では見通しをたてにくい. そのような処理を見通し良く並列化し, 並列化するには, 木構造に沿って再帰的に生成されるタスクを自動的に負荷分散する**タスク並列処理系**が有用である. 本論文ではタスク並列処理系の一つとして著者らが提案し公開している MassiveThreads [25] を用いて, オープンソースの高速な FMM 実装である ExaFMM[23] [*1]の並列化及び性能評価を行う. それを通して, FMM に現れる処理の並列化にはタスク並列が有効であることを, 記述面及び性能面から議論する. 特に,

- Dehnen[7] に現れる Generic Tree Walk, それによる, 作用反作用の法則を忠実に守った (運動量を精密に保存する) 計算法の並列化
- 適応的な Treecode や FMM で用いられる木構造の生成

などが, いずれもタスク並列処理系によって見通しよく並列化できることを示す.

2. 背景

2.1 多体シミュレーションと高速多重極展開法

多体シミュレーションでは, N 個の粒子の, 位置 $x_0, x_1, \ldots, x_{N-1}$ や, 問題に応じて決まるその他の属性 (質量や電荷) から, それらの間の作用 (力やポテンシャル):

$$\phi(x_i) = \sum_{j=0}^{N-1} \mu_j g(x_i - x_j) \qquad (1)$$

をすべての i $(0 \le i < N)$ に対して計算することが問題の中心である. ここで, $g(x)$ は, x だけ離れた 2 粒子間の作用を表す関数であり, 例えばクーロンポテンシャルであれば,

$$g(x) = \frac{1}{|x|}$$

となる. そして, μ_j は粒子 j の電荷量である.

式 (1) をすべての i について計算することは, そのままでは $O(N^2)$ の計算量を要するが, FMM はそれを $O(N)$ の計算量で行う. 鍵となる考え方は, 互いに遠く離れた二つの粒子群 A と B の間の相互作用を, 「$x \approx x'$ ならば $g(x) \approx g(x')$」ということを利用してまとめて効率よく計算することである. 以下は Dehnen[6], [7] によるもので, アルゴリズムの数学的な導出は省略し, 計算手順の概要を述べる. なお, [7] では提案アルゴリズムを, FMM とは異なる $O(N)$ の N 体シミュレーションアルゴリズムであると位置づけているが, 本稿では特に区別せずこれも FMM の一種と呼ぶ.

Dehnen の FMM アルゴリズム全体は 4 つのフェーズから構成される.

(1) **Build:** 木を生成する
(2) **Upward:** 各セルに対し多重極展開を計算する
(3) **Interact:** 多重極展開をセル間で相互作用させる
(4) **Downward:** interact で求まった相互作用を各粒子へ適用する

生成される木は, シミュレーション領域全体を根とする木構造である. 木構造の各ノードを「セル」と呼び, 各セルは空間内のある正方領域 (2 次元なら正方形, 3 次元なら立方体) を表している. 以下では 3 次元を仮定して説明を続ける. その場合あるセル C の子は, C (が表す立方体) の各辺を 2 等分してできる, 8 個の立方体 (を表すセル) である.

Build フェーズが生成する木は, 木のリーフセルに含まれる粒子の個数が, 1 個以上かつある一定値 N_{crit} 以下になるように作られる. 従って一般には, 粒子の密度に応じて, 場所によって異なる深さを持つ木が作られる.

この作り方から, 粒子数が N 個であれば, よほど粒子が極端に偏らない限り, セルの数は $O(N)$ となる. 仮に, 子を持つセルは常に 2 個以上の子を持つ (言い換えれば子を一つしか持たないセルは存在しない) と仮定すれば, 明らかにセルの数は $(2N-1)$—各リーフに 1 つずつの粒子が入った 2 分木のセル数—で抑えられる.

Upward フェーズでは, 各セルに対して, それに含まれる粒子のみから定まる固有の係数 (多重極展開) を計算する. Barnes ら [1] によるオリジナルの Treecode における, 質量合計や, 重心の一般化のようなものである. リーフセルにおいてはそこに含まれる粒子から直接計算され, 非リーフセルにおいては, 子ノードの多重極展開から親ノードのそれを計算することができる. 従って計算自体は木構造を, 帰りがけ順序 (postorder) で走査することで求めることができる. 各セルでの計算は $O(1)$ で, 従って Upward フェーズ全体として $O(N)$ の計算量である.

Interact フェーズは計算時間の中心を占めるもので, 遠くのセル間で多重極展開を相互作用させる. ある二つのセルが「それらの大きさに比べて十分離れて」いれば, それ

[*1] http://www.bu.edu/exafmm/

らのセルの多重極展開を利用して近似的にそのセル間の計算を行う．その場合, その子供のセルは計算に関与しない．そうでないときは各セルの子供のセル同士の相互作用を再帰的に行う．このフェーズの計算量も $O(N)$ となる．

Downward フェーズは, 各セルに対して, interact フェーズで得られた, 遠くのセルからの相互作用を, その子供セルに伝搬・蓄積させ, 最終的にはリーフセルの各粒子に作用させる．Upward フェーズ同様, 各セルでの計算は $O(1)$ で, 従って Downward フェーズ全体として $O(N)$ の計算量である．

以上から, 木の生成以降の計算は全体として $O(N)$ で実行できる．

通常の Treecode や他の FMM の定式化に無い特徴として, 力の適用を常に対称的 (相互) に行い, 結果として作用反作用の法則を忠実に守った計算ができる, という特徴がある．というのは, セル A とセル B が十分離れており, 近似計算が可能であった場合, B が A に及ぼす作用 (力やポテンシャル) と, A が B に及ぼす作用を一度に計算し, それらを双方に適用するからである．計算量が約半分になる上, それらの力は自然に逆向き (作用・反作用の関係) になる．これは Treecode のように, 「セルから 1 粒子」という一方向の作用を基本に計算する方式では, 自然に達成できないことに注意されたい．なんとなれば, 粒子 $b \in B$ が粒子 $a \in A$ に及ぼす力と, その逆向きの (粒子 $a \in A$ が粒子 $b \in B$ に及ぼす) 力とは別々に計算され, しかも同じ近似方法を用いて計算されるわけではないからである．これは計算時間という観点からも, 計算精度 (運動量の保存) という観点からも好ましくない．

並列化という観点からは, 木構造生成の並列化と, 上で述べた対称的な相互作用の計算の並列化が興味深い．残りの部分は計算時間としても少ない上, 基本的には木構造に沿って全セルを走査するだけのもので, タスク並列を用いれば記述上の困難はあまりない．しかしこれですら, セル配列全体に対する for all ループではなく, 単純に OpenMP の並列 for で並列化できるわけではないことに注意しておく．

2.2 ExaFMM

ExaFMM は, ボストン大学およびサウジアラビア・アブドラ王立科学技術大学の横田, Barba らによって開発が進められているオープンソースの FMM コードであり, 世界最速の FMM 実装の一つであり [23], 以下のような特徴がある．

- 複数の木構造生成方式を選択できる．(a) 密度に適応した木構造を作る topdown 方式, または (b) 深さ一様の bottomup 方式
- 複数の多重極展開方法を選択できる．(a) テイラー展開, または (b) 球面調和関数展開
- Dehnen[7] による generic tree walk を用いた相互作用の計算
- 1 コア, マルチコア, GPU, MPI 分散メモリ環境など, 多様な環境で高性能を発揮することを目指した開発

本稿では, 1 コア用の逐次コード[*2]を元にして, ノード内の並列化およびその性能評価を行った．

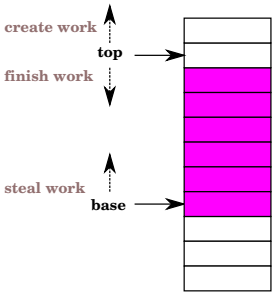

図 1 ワークスティーリングのための deque

2.3 再帰的な処理の並列化, タスク並列処理, ワークスティーリング

FMM では多くの部分で木構造に対する再帰的な処理が行われる．このような処理を並列化するには, 再帰呼び出しに対してタスクを生成し, それを動的に負荷分散できる, タスク並列処理系が有用である．構文としてタスク生成をサポートするだけでは不十分で, 大量のタスク生成が行え, それらをなるべく多くな部分木を単位としてプロセッサ間で負荷分散させるアルゴリズムが必要である．そのような負荷分散アルゴリズムの代表的なものに, ワークスティーリング [15] がある．

ワークスティーリングでは, ある数 (典型的にはコア数) のワーカが, 互いにタスクを盗み合いながら負荷分散する．各ワーカは自分が管理しているタスクを収容する deque を持つ (図 1)．初期状態では唯一のタスク (main 関数を実行するタスク) だけが存在しており, 従って一つのワーカだけがタスクを実行している．タスクが生成されるにつれ, それらが他のワーカに盗まれることで, 負荷分散が行われる．スケジューリング方式は以下の 3 つのキーワードにまとめられる．

Work first 実行: タスクが生成される際, 直ちに生成されたタスクの実行を開始する．つまり, 親の実行を継続せず, あたかも通常の逐次実行のようにタスクの実行が開始される．

LIFO 実行: タスクが終了したら, その親タスクが他のワーカにスチールされていない限り, その親の実行に戻る．

FIFO スチール: タスクを持たないワーカがタスクを盗む際は, ワーカをランダムに選び, そのワーカのタスクスタック中の最も底にある (タスクの親子関係の先祖に近い) タスクを盗む．

[*2] ソースコード中の, fast/ ディレクトリ下のコード

表 1　タスク並列の高性能なオープンソース実装

処理系	実装形態	表面言語	work first
Cilk	トランスレータ	Cilk	yes
TBB	ライブラリ	C++	no
Java Fork Join	ライブラリ	Java	no
MassiveThreads	ライブラリ	C/C++	yes

これらの方式により, タスクの親子関係からなる木構造が, 大きな粒度で負荷分散 (木の根に近いところで, 分割) され, かつ個々のワーカ内ではあたかもタスク生成が通常の逐次関数呼び出しであったかのような順序でプログラムが実行される. これは, 個々のコアがメモリへアクセスする順序が, ワークスティーリングによる負荷分散がおきない限り, 逐次の場合と変わらないということを保証する意味で非常に重要である.

ワークスティーリングや, それを近似した処理系は数多く研究・開発されてきた [3], [8], [11], [13], [17], [20], [25]. 現在オープンソースとして容易に利用可能な処理系としては, C 言語の拡張である Cilk*3, C++用のライブラリである Intel Threading Building Block (TBB) *4, Java 用のライブラリである Java Fork Join フレームワーク*5, C/C++用のライブラリである MassiveThreads *6などがある. このうち work first を実現しているものは Cilk と MassiveThreads だけである. また, OpenMP 3.0 以降の仕様には tasks 構文が取り入れられ, C/C++/Fortran でタスク並列が記述可能となっているが, 現在までの GCC (version 4.6) に実装されている tasks 構文の実装は, スケーラビリティが悪く, 並列実行性能としては前述の処理系と比肩しうるものではない.

2.4 MassiveThreads

MassiveThreads[25] は, pthread と互換の API およびブロッキング IO のセマンティクスを持った, 軽量スレッドライブラリである. したがって, 通常の C/C++から pthread API を呼び出すだけでタスク並列処理が実現できる. 例えば pthread_create がタスク生成, pthread_join がスレッドの終了待ちを行う API である. MassiveThreads は pthread の代わりにリンクするだけで利用でき, コンパイラやトランスレータを必要としないため, タスク並列をサポートする言語の実行時システムを実装するのに適している. 現在 MassiveThreads を用いた Chepel [4]*7 のタスク並列実装が行われている.

ただし, pthread API はそのままエンドユーザが呼び出すライブラリとしては低レベルである. そこで, TBB のタスク並列 API のひとつである, task_group クラスと同等のクラスを MassiveThreads 上に実現した. task_group クラスは以下のシグナチャを持つ.

```
1  class task_group {
2    void run(closure); // closure を実行するタスクを生成・登録
3    void wait();           // 登録されたタスクすべての終了を待つ
4  }
```

使用例は以下で, C++0x 標準のクロージャ (ラムダ式) のシンタックスを用いて, 任意の文を簡単にタスクとして生成することができる. [=,&x] は変数 x を親子間で共有 (参照渡し) し, それ以外の変数はコピーして渡すことを示している.

```
1  int fib(int n) {
2    if (n < 2) return 1;
3    else {
4      task_group tg;
5      int x, y;
6      tg.run([=,&x] { x = fib(n - 1); });       // タスクを生成
7      y = fib(n - 2);  // 並行してもうひとつの再帰呼び出しを実行
8      tg.wait();                                 // 両タスクの終了待ち
9      return x + y;
10   }
11 }
```

MassiveThreads 上での実装は非常に単純である. run メソッドで MassiveThreads の pthread_create を呼び出すとともに, 作られたスレッドの名前を task_group 構造体の中に登録する. wait メソッドでは task_group 構造体の中に登録されているスレッドすべての終了を MassiveThreads の pthread_join を呼び出して待つだけである. そして, TBB の実装と異なり, 忠実に work first 実行を実現している.

3. 関連研究

これまで FMM や treecod の実装は数多く報告されている [19],[22],[21],[18] が, MPI や CUDA などの低レベルなプログラミングモデルを用いて明示的なデータ分割, 負荷分散を行っているものが殆どで, FMM や Treecode が持っていた自然で再帰的な記述を保ちながらタスク並列処理計を用いて実装・評価した例は少ない. 松井ら [24] は Barnes らが公開している Treecode の高速版の並列化を, タスク並列処理言語 Tascell を用いて並列化している. Treecode は FMM と異なり, もともとセル (多数の粒子) から一粒子への作用が, 計算の基本となる単位である. 単純な実装であれば, 異なる粒子に対する計算を並列に実行することは容易であるが, 高速版 Treecode では, 「粒子 a と粒子 b が近くにある場合, a への作用を計算すべきセルの集合と, b への作用を計算すべきセルの集合が似通っている」という性質を利用して, それを差分更新によって計算する. このため異なる粒子間の計算に依存関係が生じ, 並列化が困難になる. 松井ら [24] はこれを Tascell のバックトラックに基

*3 http://supertech.csail.mit.edu/cilk/
*4 http://threadingbuildingblocks.org/
*5 http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
*6 http://code.google.com/p/massivethreads/
*7 http://chapel.cray.com/

づく並列化を利用して解決している．これに対し FMM では, 最初からセル同士の相互作用を計算の基本とする．「粒子 a と粒子 b が近くにある場合, 計算すべきセルの集合が似通っている」という性質は, 近くのセルの及ぼす力 (ポテンシャル) を多重極子としてまとめ, それを遠くのセルに一度の計算で作用させるということで, 自然に利用されている．結果として我々の FMM の並列化は元々の逐次プログラムとの乖離が少ない, 素直なものになっている．

また我々の並列化は, 2.1 節で述べた, ある作用 (粒子 $a \to$ 粒子 b) とその逆向き ($b \to a$) の作用を一度しか計算しない, 対称的な相互作用の計算を並列化している．また, 相互作用の計算のみならず木構造生成部分も並列化している, などの違いがある．

タスク並列処理計を ExaFMM に適用した先行研究として, Ltaief および Yokota [14] がある．QUARK [*8] という, DAG スケジューリングを行う処理系を利用して, ExaFMM の相互作用の計算部分を並列化している．10^6 粒子に対する 16 コアまでの台数効果を報告しているが, 総じて, QUARK スケジューラのボトルネックが現れ, QUARK に渡すタスクの粒度を大きくする必要があると報告している．[14] では, 多重極展開の次数は 6, QUARK には 100, 1000, 10000 個などの相互作用を一つのタスクとして渡している．それに対し本稿では, 相互作用の計算では, 多重極展開の次数は 3, 木のリーフまでタスクを生成し続けるなど, プログラマにとって自然な細粒度のタスク生成を行っても良好な台数効果が得られている．

4. ExaFMM の MassiveThreads を用いた並列化

4.1 ExaFMM で用いられるデータ構造

中心的なデータ構造の定義として以下の 4 つを用いる．

- `Body` クラス: 1 つの粒子を表すクラス
- `vector<Body>`: 粒子の配列
- `Cell` クラス: セル (木構造のノード) を表すクラス
- `vector<Cell>`: セルの配列

中心的なデータとして,

- `bodies`: 全粒子を格納する, `vector<Body>`のインスタンス
- `cells`: 全セルを格納する, `vector<Cell>`のインスタンス

が作られる．

`vector<T>`は, C++の標準テンプレートライブラリに含まれる, 動的に拡張可能な配列であるが, 実体は C の配列・ポインタに近い．`vector` 上で連続したインデクスを持つ要素は, 実際のアドレスも連続している．vector の途中の要素を参照するには, vector と添字の組を用いてもよいが,

[*8] http://icl.cs.utk.edu/quark/index.html

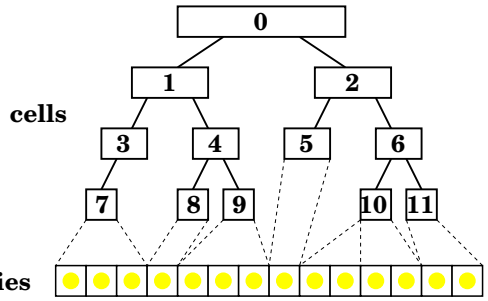

図 2 ExaFMM における粒子およびセルの表現．木のノード内の番号は, cells 配列内で格納される位置 (可能な一例) を示している

直接配列の途中を参照する, いわばポインタの代替となるような手段として, `vector<T>::iterator` というクラスが用意されている．つまり, 動的に拡張可能な点以外は, ほぼ C の配列・ポインタと同じである．

そこで以下の説明では C 言語のポインタ表記を用いる．

セルの子ノードへのポインタは, `cells` 内でのインデクス (unsigned int) として表されている．「木構造」は実際にはこの `cells` のことであり, 木構造が生成されたあと, `bodies` と `cells` は以下の条件を満たす．

- (C1) 全セルが `cells` 内に, 密に (隙間なく) 格納されている．
- (C2) セルは `cells` 内で, トポロジカル順序もしくはその逆順に並んでいる．
- (C3) 各セルに所属する粒子は, bodies 上で連続した位置に並んでいる．

どれも, 必ずしもこれらを満たしていなくても, 必要な計算を実行することは可能であるが, メモリ量の節約, メモリ参照の局所性や連続性を高める上で重要である．

条件 (C3) は, 木構造末端のセル (リーフセル) のみならず, 全セルに対して維持されており, 結果的に bodies 配列は, 木を深さ優先探索した際に遭遇する順序— 実際には 4.3 で述べる Morton 順序—で整列されている．

木が生成された後の cells 配列, bodies 配列の状態を図 2 に表した．ただし図示を簡単にするため, 1 次元 (2 分木) を仮定して表示している．

以降ではまず, 木が完成した後の interact フェーズの並列化を先に述べ (4.2 節), その後に木構造の生成 (4.3 節) について述べる．

4.2 相互作用 (interact) フェーズおよびその並列化

4.2.1 逐次版 ExaFMM の相互作用フェーズ

以下は 2 つのセル同士を作用させる関数である．

```
1 void interact2(Cell * A, Cell * B) {
2   if (A と B が十分遠い) {
3     approximate(A, B); // 近似して終了
4   } else if (A と B がともにリーフ) {
5     P2P_2(A, B);  // 粒子間で直接相互作用
```

```
6   } else if (A が B より大きい) {
7     for each child a of A {
8       interact2(a, B);  // A を分割しそれぞれ B と作用
9     }
10  } else {
11    for each child b of B {
12      interact2(A, b);  // B を分割しそれぞれ A と作用
13    }
14  }
15 }
```

セル A とセル B がその大きさに比べて十分離れている場合は, すでに求めてあるセルの多重極展開を用いて相互作用が計算され, それ以上の計算は行われない (2-3 行目). A 内にある各粒子へ B が及ぼす力を伝搬させたり, B 内にある各粒子へ A が及ぼす力を伝搬させるのはこの後の downward フェーズで行われる.

十分離れていない場合, 両者がリーフであれば直接計算を行う (4-5 行目). そうでなければどちらかのノードの子ノードに対して再帰呼び出しを行う (6-14 行目). つまり, A $\leftrightarrow$ B の相互作用を, その子セル間の相互作用に分解していく.

ここで特徴的なのは, interact2(A, B) を行うと, B が A に及ぼす力と, A が B に及ぼす力の両方が (一回の計算で) 計算されることである. 結果として, interact2(A, B) はセル A, セル B の**両方**を書き換える.

また, 上記に加え, A と B が同じセルである場合の関数 interact1 を別途作っておく. つまり以下は, セル A とそれ自身の相互作用を計算する.

```
1  void interact1(Cell * A) {
2    if (A がリーフまたは含まれる粒子数が少ない) {
3      P2P_1(A); // 粒子間で直接相互作用
4    } else {
5      for each child a of A {
6        for each child b of A {
7          // A の子供同士を作用
8          if (a == b) interact1(a);
9          else if (a < b) interact2(a, b);
10       }
11     }
12   }
13 }
```

やっていることは interact2 の場合とほぼ同じだが, A と A が十分離れていることはありえないのでその検査が省略されている. また, 再帰呼び出し時に注意が 2 点あり,

- ある子ノードと, それと同じ子ノード同士を interact させる場合は interact1 を呼ぶ
- x と y が異なる子ノードの場合に, interact2(x, y) という再帰呼び出しと, interact2(y, x) という再帰呼び出しの片方だけを行う.

9 行目で行っている a < b という検査は後者を達成するためである.

4.2.2 並列化

一見すると, interact1/interact2 の並列化は非常に単純で, 再帰呼び出し (interact1 の 5, 6 行目および interact2 の 7, 11 行目の for ループ) を並列に実行すればよいだけと思えるかも知れないがそうではない.

前述したように, interact2(A, B) の呼び出しは, B が A に及ぼす力と, A が B に及ぼす力の両方を (一回の計算で) 求める. そして interact2(A, B) はセル A とその子孫, セル B とその子孫の**両方**を書き換え得る. この性質により, interact2 の再帰呼び出しは並列に行えない. interact1 にも類似の問題がある.

基本に立ち戻って問題を考えると, ここで達成したいのは, A の子ノードと B の子ノードの全組み合わせ (a, b) に対して, interact2(a, b) を呼び出すことである. そこで, セルのリストを二つ受け取り, それらに含まれるセルの組み合わせに対して interact2 を呼ぶ補助関数 interact_list2 を作る. そしてそれらの呼び出しが互いに衝突しないように並列実行を行う. やや模式的になるが, intreract_list2 は以下のように書ける.

```
1  void interact_list2(CellList C, CellList D) {
2    if (C と D のどちらかが 1 要素)
3      // 逐次に
4      for each a in C {
5        for each b in D {
6          interact2(a, b);
7        }
8      }
9    } else {
10     // 並列に
11     C0, C1 = C の前半/後半;
12     D0, D1 = D の前半/後半;
13     task_group tg;
14     // 前半同士, 後半同士を並列に
15     tg.run([=] { interact_list2(C0, D0); });
16     interact_list2(C1, D1);
17     tg.wait();
18     // 前半と後半, 後半と前半を並列に
19     tg.run([=] { interact_list2(C0, D1); });
20     interact_list2(C1, D0);
21     tg.wait();
22   }
23 }
```

CellList は, セルの集合を表しているが, 実際には cells 配列内の一部分を, その先頭・終端のアドレスで表している.

ここで 15 行目と 16 行目の呼び出しはそれぞれ異なるセルを書き換えるため, 安全に並列実行できることに注意されたい. 19 行目と 20 行目の呼び出しについても同様である.

この補助関数を用いて interact2 自身は以下のようになる.

```
1  void interact2(Cell * A, Cell * B) {
2    if (A と B が十分遠い) {
3      approximate(A, B); // 近似して終了
4    } else if (A と B がともにリーフ) {
```

```
    P2P_2(A, B);  // 粒子間で直接相互作用
  } else {
    // A の子供と B の子供を相互作用
    intereract_list2(children of A, children of B);
  }
}
```

interact1 の方の並列化も同様で, セルのリストを一つ受け取り, それに含まれる二つのセルの組み合わせに対して interact1 または interact2 を呼ぶ補助関数 interact_list1 を作る.

```
void interact_list1(CellList C) {
  if (C が 1 要素) interact1(C[0]);
  else {
    C0, C1 = C の前半/後半;
    task_group tg;
    // 前半同士, 後半同士を並列に作用
    tg.run([=] { interact_list1(C0); });
    interact_list1(C1);
    tg.wait()
    // 前半と後半を作用
    interact_list2(C0, C1);
  }
}
```

interact1 自身は以下のようになる.

```
void interact1(Cell * A) {
  if (A がリーフまたは含まれる粒子数が少ない) {
    P2P_1(A); // 粒子間で直接相互作用
  } else {
    // A の子供同士を相互作用
    interact_list1(children of A);
  }
}
```

4.3 木構造生成 (build) フェーズおよびその並列化

4.3.1 逐次版 ExaFMM の木構造生成

木構造生成の目標は, 4.1 節で述べた条件 (C1)-(C3) を満たす, セルの配列を作ることである. 元々の逐次版 ExaFMM では, 木構造の作成に二つの方式が実装されており, コンパイル時にどちらかを指定できるようになっている.

以下で現れる, Morton 順序については付録 A.1, [22] などを参照.

topdown: Barnes ら [1] によるものと同じ方式で, ルートのみからなる木構造を作り, それに粒子を一つずつ追加していく方式. 粒子を加える際, まずその粒子の位置を含むリーフセルを見つける. それによりそのリーフセルに含まれる粒子数が一定値を超えたら, そのリーフセルに子セルを追加する. このようにして木を根から葉へ向けて成長させていく. 新しいセルが生まれる際, それは配列 (STL の vector) の最後尾に追加され, 結果としてここで配列の拡張が起きうる. すべての粒子を追加し終えたところで, 4.1 節で述べた条件を満たすべく, できた木構造を深さ優先で操作し, セルおよび粒子を並べ替える.

ここで, セルを並び替えているのは条件 (C1)-(C3) を満たすためである. なお, 実は並び替える前の木構造には, セルが備えるすべての情報が必要というわけではないため, そのノードは Cell クラスを軽量化したクラス (Node クラス) のインスタンスになっている.

bottomup: 最小の木構造を作ることにはこだわらず, 全体の粒子数 N を元に適当な経験則で木の深さを確定させる. 例えば 8 分木で, リーフに含まれる粒子数を 100 程度にしたければ, 最大深さを $1+\log_8(N/100)$ とする. 次に全粒子を Morton 順序で整列し, その順に「その粒子を含むリーフセルが, 生成されていなければ生成する」ことを行う. 粒子を事前に Morton 順序で整列させているので, ある一つのリーフセルに含まれる粒子は, 連続して現れることが保証されているため, 「その粒子を含むリーフセルが, 生成されて」いるかどうかの検査は簡単に行える. リーフができたら, それらのリーフセルを同様に Morton 順序でスキャンしてリーフの親となるセルを作る. それができたらそれらの親セルをスキャンしてその親を作る, ... ということを木の深さ分繰り返してすべてのセルを生成する. セルを新しく作る際は, cells 配列の末尾にそのセルが挿入される. このやり方で作られた cells 配列は自動的に 4.1 節で述べた性質を満たしている.

Topdown の方法では, 「リーフセルには 100 個以下の粒子しか含まれない」という条件のもとで, 最小の木ができる. すなわち, 非リーフセルは必ず 100 よりも多くの粒子を含む. Bottomup の方法では, どちらとも, それが満たされるという保証はない. その意味で, topdown だけが真に適応的なデータ構造である.

4.3.2 並列化

ExaFMM に実装されている, topdown, bottomup のどちらのやり方も, それらを微修正して直接並列化することは難しい. 困難な原因は, 表面的にはどちらも, 木構造のノードのメモリを確保するために vector の動的な拡張を用いている点にある.

我々はここでも再帰的なアプローチを取る. つまり, 「ある立方体領域に対する木構造は, それを 8 分割した立方体領域それぞれの木構造を組み合わせたものである」という事実を素直に再帰呼び出しを使って表現する.

そのために, 「空間のある立方体領域と, そこに含まれる粒子の集合を受け取り, それに対応する木構造を返す」関数 build_nodes を定義する.

```
// B に含まれる粒子からなる木を作る
// それぞれの粒子のMorton key ∈ [R0, R8)
Node * build_nodes(BodyList B, int R0, int R8) {
  if (B が空) return NULL;
```

```
  else if (B の要素数 <= 100) {
    // リーフノードを作って終了
    return new Node(B);
  } else {
    task_group tg;
    Node * n = new Node(B);
    // 空間を 8つに分割 ≡ Morton キーの範囲を 8 分割
    // それぞれの範囲の木を再帰的に生成
    for (i = 0; i < 8; i++) {
      tg.run([=,B] {
        R  = (R0 * (8 - i) + R8 *  i     ) / 8;
        R' = (R0 * (7 - i) + R8 * (i + 1)) / 8;
        Bi = B 中で Morton key が[R, R') の範囲にある粒子
        n->child[i] = build_nodes(Bi, R, R');
      }
    }
    tg.wait();
    // 後に備え，全ノード数を記録
    for (i = 0; i < 8; i++) {
      if (n->child[i] != NULL)
        n->n_nodes += n->child[i]->n_nodes;
    }
    return n;
  }
}
```

いくつかの注意としては,

- 粒子の集合 B は実際には配列の一部分で, 先頭/終端アドレスとして表現されている
- 17 行目で, 8 分割した立方体領域それぞれに含まれる粒子の集合を作っているが, これは, B が Morton 順序で整列していることを前提として, 2 分探索を行っている.
- 木構造のノードは, n_nodes というフィールドを持ち, それは自分を根とする部分木にいくつのノードが含まれているかを示している.

ここで作られる木構造は, 各ノードを new を用いて割り当てており, 当然のことながら, 最終的に得たい (C1)-(C3) を満たす木構造ではない. いわば, 得たい木構造のトポロジーだけを求めたものに相当する. ノードのデータ構造としても, Cell とは異なる Node を用いている. そこで上記で得られた Node の木構造をもう一度走査し, cells 配列の適切な位置にセルを作り直す. いわば, (C1)-(C3) を満たすためのコンパクションを行う. この際, 上記で各ノードの子孫にどれだけのノードが存在するかを, フィールド n_nodes に記録することで, cells 配列のどこに部分木を作ったらよいかがわかる.

```
/* n を根とする Node の木を Cell の木に変換.
   Cell は[C0, C1)に隙間なく割り当てる */
void nodes2cells(Node * n, Cell * C0, Cell * C1) {
  *C0 = node2cell(n);
  Cell * P = C0 + 1;
  task_group tg;
  for (i = 0; i < 8; i++) {
    Node * c = n->child[i];
    if (c != NULL) {
      Cell * Q = P + c->n_nodes;
      tg.run([=] { nodes2cells(c, P, Q); });
      P = Q;
    }
  }
  tg.wait();
}
```

最終的に, 木構造生成の全体像は以下のようになる. 以下で L は, 事前に定めた木の最大の深さであり, $3 + \log_8(N)$ が 9 以下であればその値, 9 以上であれば 9 としている.

```
sort bodies in Morton order;
Node * n = build_nodes(bodies, 0, (1 << (3 * L)));
nodes2cells(n, cells, cells + n->n_nodes);
```

5. 性能評価実験

5.1 実験環境

以下の二つの環境を用いた.

- マシン N (Nehalem):
  - Intel(R) Xeon(R) CPU E7540 2.0GHz
  - Nehalem マイクロアーキテクチャ
  - キャッシュサイズ L1 32KB, L2 512KB, L3 18MB
  - 6 コア × 4 ソケット (合計 24 コア)
- マシン B (Barcelona):
  - AMD Opteron(tm) Processor 8354 2.2GHz
  - Barcelona マイクロアーキテクチャ
  - キャッシュサイズ L1 64KB, L2 512KB
  - 4 コア × 8 ソケット (合計 32 コア)

FMM に関わるパラメータの設定は以下のとおり.

- 粒子の初期配置: 半径 1 の球面付近に一様に生成
- 多重極展開の方法: Taylor 展開
- 多重極展開の次数: 3
- 近似許容パラメータ (tolerance parameter [7])θ : 0.6
- リーフセルの最大粒子数: 100

interact フェーズにおいては, 特に粒度調整をしていない. すなわち木の末端まで, 再帰呼び出しと共にタスクが生成される. もちろん粒子数が 100 以下になったらそれがリーフセルになっているので, それ以上の再帰呼び出しは行われない.

5.2 オリジナル ExaFMM と比較した際の逐次性能

図 3 に, マシン N および B の 1 コア上で, 10^6 粒子の相互作用の計算にかかった時間を示す.

x 軸のラベルの意味を説明する.

- C++は並列化を施していないコード, MassiveThreads はタスク生成プリミティブを挿入したコードである.
- recursive は interact フェーズの木の操作を再帰呼び出しを用いて行うコード, stack は再帰呼び出しを用いず, 明示的にスタックを管理して行うコードである.

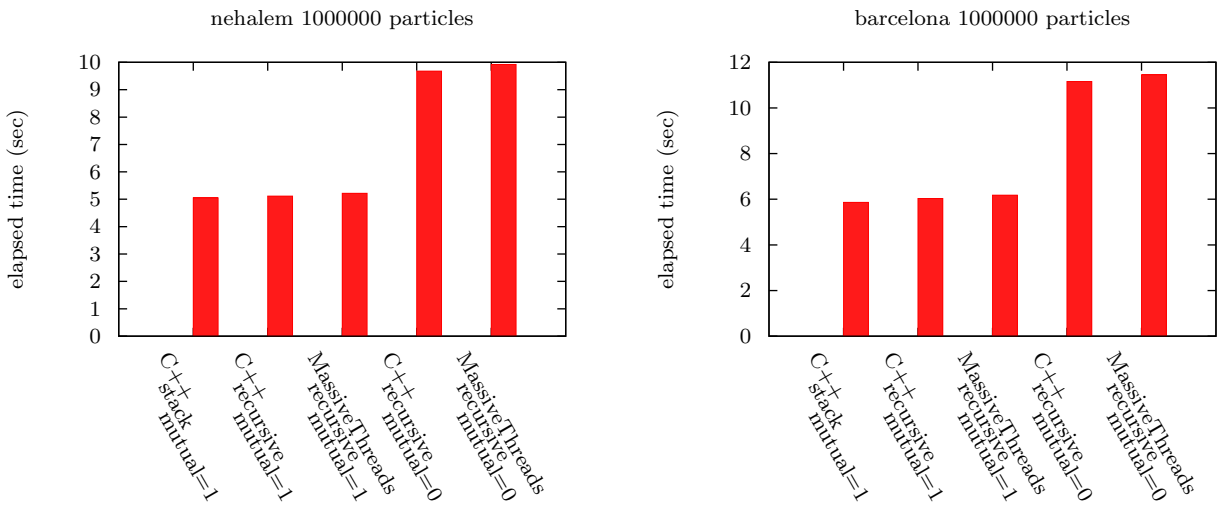

図 3 interact ステージの逐次実行時間. 上: マシン N, 下: マシン B

- mutual = 1 は, 「相互」作用 ("mutual" interaction) で計算をする, つまり粒子 $a \leftrightarrow b$ 間の相互作用を一度だけ計算して双方に適用するコード, mutual = 0 は $b \to a$ の作用と, その逆向き $a \to b$ の作用とを, 別途計算するコードである.

C++, stack, mutual=1 が, オリジナルの ExaFMM そのものである.

総じて, 逐次性能に影響をしているのは mutual パラメータのみで, 再帰呼び出しへの書き換え, タスク生成そのものは, ほとんどオーバーヘッドを加えていない. mutual = 0 とすると計算すべき作用の数が約 2 倍に増えるので, 計算時間が増えるのは当然であり, ここでもやはり C++, MassiveThreads の間に殆ど差は見られない.

以降は相互作用による計算 (mutual = 1) で実行した場合の結果のみを示す.

5.3 各フェーズの実行時間

図 4(左) は, マシン B, 10^6 粒子数で, 各フェーズの実行時間を示したものである.

sort bodies: 木生成に先立つ粒子の, Morton 順序での整列

build nodes: Morton 順序での整列した粒子から木の生成

upward pass: 各セルに対する多重極展開の計算

traverse: セル間の相互作用の計算

downward pass: セル間の相互作用を子セル, 粒子へ伝搬, 適用

明らかに相互作用の計算時間が支配的である. しかしそれと同時に, 32 コア程度の並列化であっても, 残りのフェーズを並列化しないことの影響は大きいことが見て取れる. それを分かりやすく表示したのが図 4(右) で, 各コア数での合計実行時間を 1 に正規化した実行時間, つまり各フェーズが全実行時間のどれだけを占めているかを表示している. 32 コアでは, traverse の時間は半分以下で, 木構造の生成およびそれに先立つ粒子の整列が 60%の時間を占めている. 次節で見るように, traverse の台数効果に比べて, 他のフェーズの台数効果は出ていないことが影響している.

また, 木構造の生成そのものよりも, それに先立つ整列の方にむしろ時間を費やしていることもわかる. もっともここでは, 完全にランダムな粒子を Morton 順序に並べ替える処理が行われており, 実際の応用においては毎ステップごとにここまで激しい粒子の入れ替えが起きるわけではない. 整列はクイックソートの再帰呼び出し部分を並列化したもので, もとよりスケーラブルな台数を期待できるアルゴリズムではない (逐次計算量 $O(n \log n)$ に対しクリティカルパスが $O(n)$) が, 予想以上に出ていない.

以上から, より現実的なシミュレーションの設定 (各ステップごとに行われる粒子の入れ替えは少ない) で, よりスケーラビリティの高いアルゴリズムを用いて今後実験を継続する予定である.

5.4 各フェーズの台数効果

図 5 は, マシン N 上, 10^6 粒子で, 各フェーズ単体での台数効果を取り出したものである. traverse フェーズの台数効果は 32 コアで 25 倍出ており, そこそこ良好であるが, その他の処理の台数効果は満足な結果は出ていない. 特に, traverse の次に大きな計算時間を占める整列操作の台数効果が低い.

総じて, 現時点では殆どこの部分の性能の解析が出来ていないということであるが, 一般論として, それらの処理は

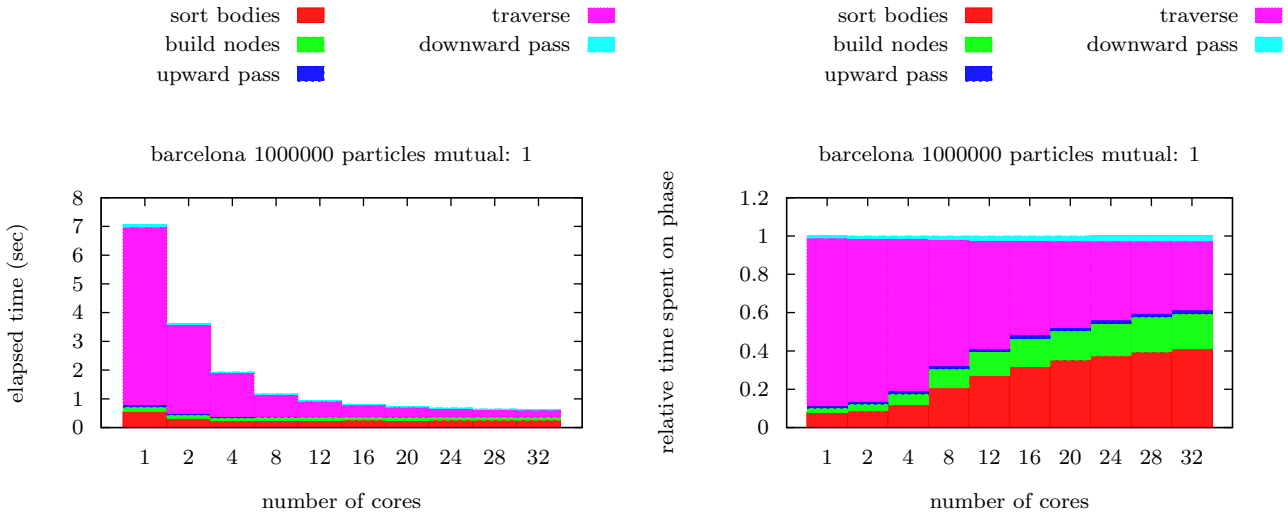

図 4 各フェーズの実行時間 (左), 相対実行時間 (右)

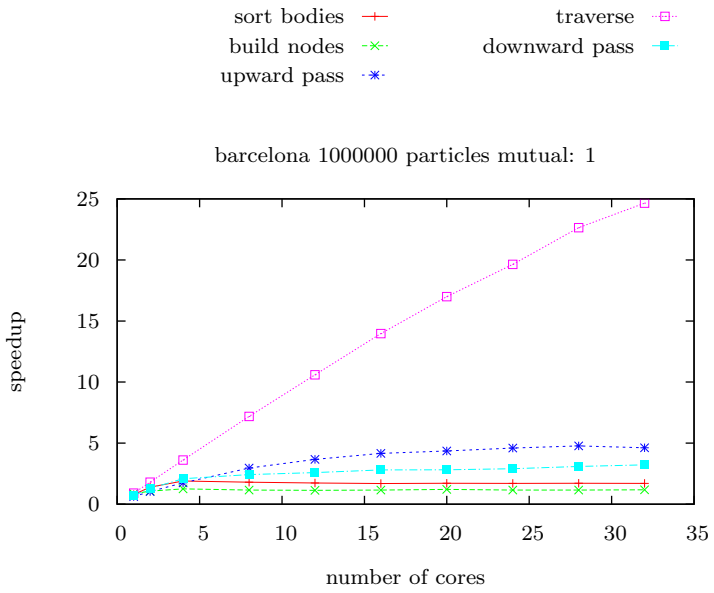

図 5 各フェーズの台数効果

データ参照あたりの計算量が少ないフェーズであり, ワークスティーリングによって, タスクがキャッシュとの親和性や, NUMA を無視したコアに割り当てられることで必要以上に台数効果が劣化するということが考えられる. 今後は, この台数効果を定量的に解析し, ローカリティを考慮したタスクスケジューラの設計へとフィードバックする予定である.

5.5 traverse フェーズおよびの各粒子数による台数効果

最後に traverse フェーズ, および全フェーズを合計した台数効果を, 様々な粒子数, マシン N, B それぞれに対し表示したものが図 6 および図 7 である. traverse フェーズは, 粒子数が大きくなるに連れて台数効果が良好になって行く. マシン N では, 24 コアで, 500K 以上の粒子数で 90% 以上の効率を達成している.

6. まとめと今後の課題

タスク並列処理系により, ExaFMM の並列化を見通し良く行うことができた. 特に,

- 同じ粒子間の相互作用を, 一度だけ計算し両者に逆向きに適用するという, いわゆる作用反作用の法則を精密に保つ計算法を並列化することができた.
- 適応的な木構造の生成を, 並列化することができた.

traverse フェーズの台数効果は良好で, Nehalem 6 コア ×4 ソケットのマシンで, 500K 以上の粒子に対しては 90%以上の台数効果を得た. しかし, 残りのフェーズの台数効果は悪く, その性能解析もできていない.

今後, まずはその解析をすることが重要であるが, 一般論として今後, 動的負荷分散を行う処理系は, キャッシュ上のデータとの親和性や, NUMA を考慮したスケジューリ

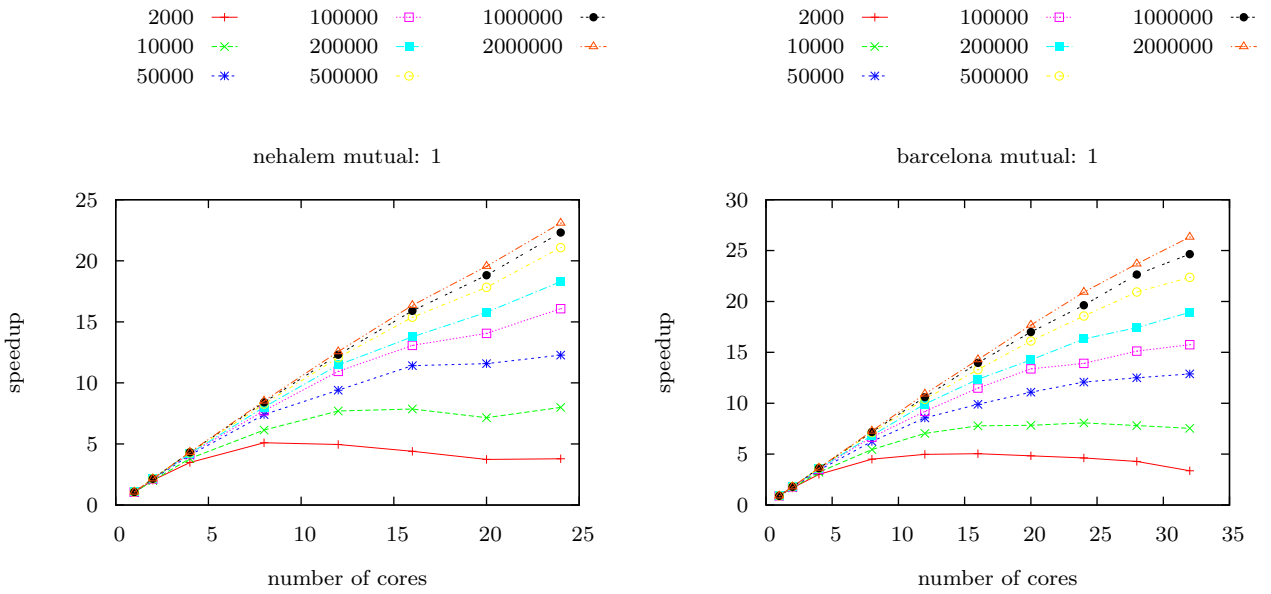

図 6 マシン N (左) および B (右) 上での traverse フェーズの各粒子数での台数効果

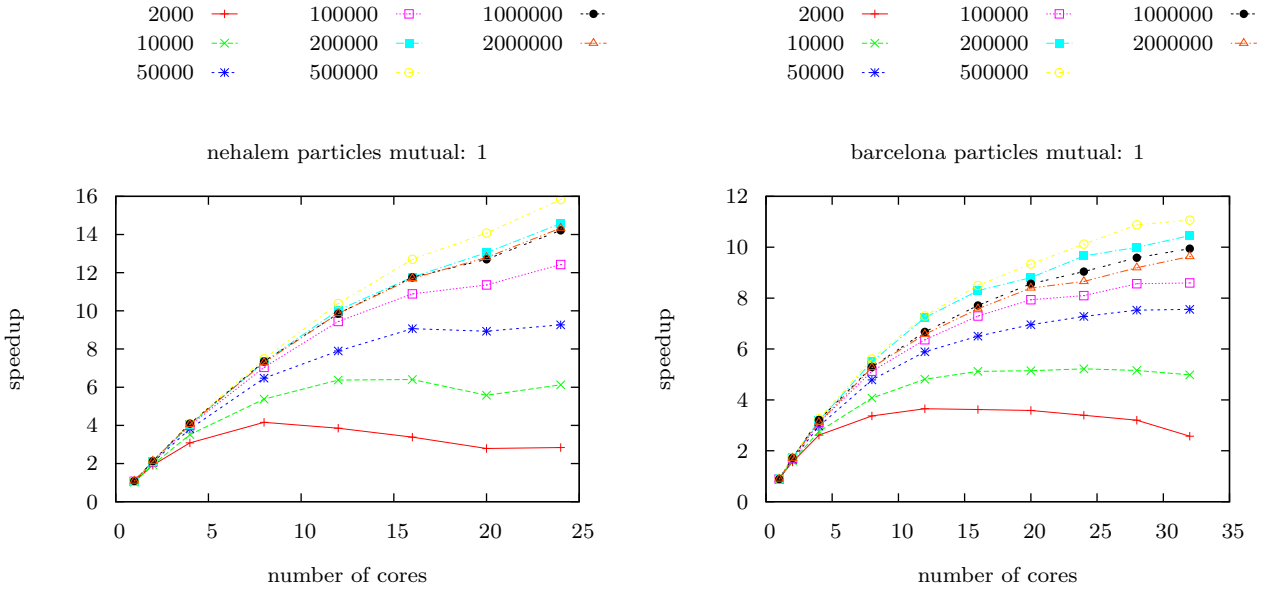

図 7 マシン N (左) および B (右) 上での全フェーズ合計の各粒子数での台数効果

ングを行うことがますます重要であり, 今回の実験でもそのことが確認された, という可能性が大きい. その場合, 現在進めている MassiveThreads のカスタマイズ可能なスケジューリング API などが有用になると期待される.

また, 我々は統一的なタスク並列 (再帰呼び出し) により記述で, 分散メモリ計算機上での実行もサポートすることを目指して, PGAS 処理系および MassiveThreads の分散メモリ拡張の実装を進めている. それらを用いて大規模な分散メモリ環境で FMM を始めとするアルゴリズムを, 高生産・高性能に実行することが中期的な目標である.

謝辞 本研究の一部は科学技術振興機構, CREST 研究課題「高性能・高生産性アプリケーションフレームワークによるポストペタスケール高性能計算の実現」の助成を得て行われた.

参考文献

[1] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[2] Rick Beatson and Leslie Greengard. A short course on fast multipole methods. *methods and elliptic PDEs*, 1997.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[4] Bradford Chamberlain, David Callahan, and Hans Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[5] Aparna Chandramowlishwaran, Samuel Williams, Leonid Oliker, Ilya Lashuk, George Biros, and Richard Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[6] Walter Dehnen. A Very Fast and Momentum-conserving Tree Code. *The Astrophysical Journal*, 536(1):L39–L42, June 2000.

[7] Walter Dehnen. A Hierarchical (N) Force Calculation Algorithm. *Journal of Computational Physics*, 179(1):27–42, June 2002.

[8] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*, pages 212–223, New York, New York, USA, May 1998. ACM Press.

[9] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):280–292, 1987.

[10] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 TFlops hierarchical N -body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, page 1, New York, New York, USA, November 2009. ACM Press.

[11] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. *ACM SIGPLAN Notices*, 44(4):55, February 2009.

[12] Ilya Lashuk, George Biros, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, and Denis Zorin. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, page 1, New York, New York, USA, November 2009. ACM Press.

[13] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande - JAVA '00*, pages 36–43, New York, New York, USA, June 2000. ACM Press.

[14] Hatem Ltaief and Rio Yokota. Data-Driven Execution of Fast Multipole Methods. Technical report, March 2012.

[15] Eric Mohr, David A. Kranz, and Robert Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[16] Abtin Rahimian, Ilya Lashuk, Shravan Veerapaneni, Aparna Chandramowlishwaran, Dhairya Malhotra, Logan Moon, Rahul Sampath, Aashay Shringarpure, Jeffrey Vetter, Richard Vuduc, Denis Zorin, and George Biros. Petascale Direct Numerical Simulation of Blood Flow on 200K Cores and Heterogeneous Architectures. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, November 2010.

[17] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[18] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, June 1995.

[19] Jaswinder Pal Singh, Chrk Holt, John Hennessy, and Anoop Gupta. A parallel adaptive fast multipole method. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*, pages 54–65, New York, New York, USA, December 1993. ACM Press.

[20] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating Futures into Calling Standards. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '99*, pages 60–71, New York, New York, USA, May 1999. ACM Press.

[21] Michael S. Warren and John K Salmon. Astrophysical N-body simulations using hierarchical tree data structures. pages 570–576, December 1992.

[22] Michael S. Warren and John K Salmon. A parallel hashed Oct-Tree N-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*, pages 12–21, New York, New York, USA, December 1993. ACM Press.

[23] Rio Yokota and Lorena A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *International Journal of High Performance Computing Applications*, pages 1094342011429952–, January 2012.

[24] 松井 健, 平石 拓, 八杉 昌宏, and 馬谷 誠二. 高速版 Barnes-Hut 多体シミュレーションの並列実装. In **先進的計算基盤システムシンポジウム** *SACSIS*, 2012.

[25] 中島潤 and 田浦健次朗. 高効率な I/O と軽量性を両立させるマルチスレッド処理系. **情報処理学会論文誌 プログラミング** *(PRO)*, 4(1):13–26, 3 月 2011.

付　　録

A.1 Morton 順序

Morton 順序 (Morton Order) は, 多次元空間の点を, Morton Key 呼ばれる自然数の大小で順序付けたものである. 簡単のため 2 次元で説明する. 図 A·1 は正方形の各辺を 4 等分割してできた各セルの Morton Key, およびその 2 進数表現を表している. 矢印は Morton 順序を表している. その形から, Z 順序と呼ばれることもある. A·2 は 8 等分割の場合である. これらから 16 分割, 一般に 2^L 分割した場合の Morton 順序や, 3 次元の場合を想像するのは容易であろう.

Morton 順序の有用な性質は, 元々の正方領域を再帰的に等分割して得られる正方領域に含まれるセルの Morton Key の集合が, 単純な区間になることである. 例えば図 A·1 で, Morton Key 全体は $[0, 16)$ に含まれる数だが, 左下が, $[0, 4)$, 右下が $[4, 8)$, 左上が $[8, 12)$, 右上が $[12, 16)$ という具合に, **「空間の再帰的な等分割が, 単なる一次元の区間の等分割に対応する」**.

また, もうひとつの重要な性質は, あるセルの座標からそのセルの Morton Key を得るのが簡単だということであ

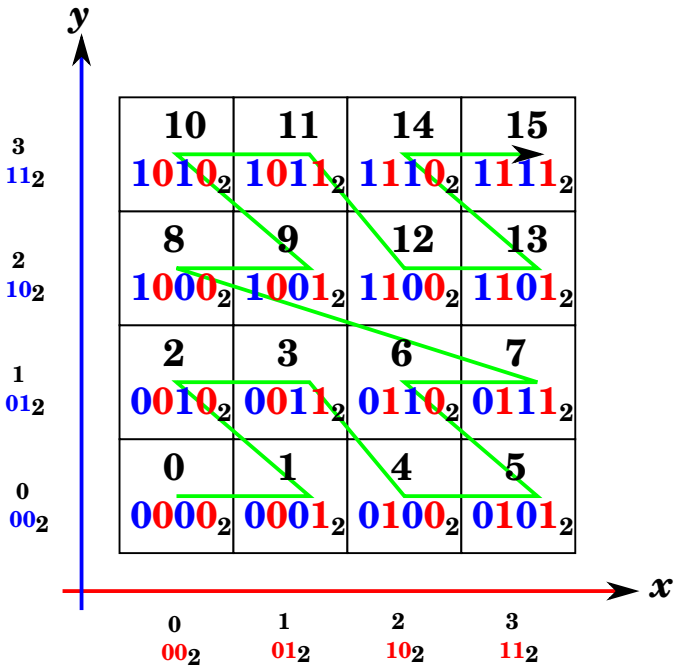

図 A·1 Morton Key と Morton 順序 (4 分割の場合)

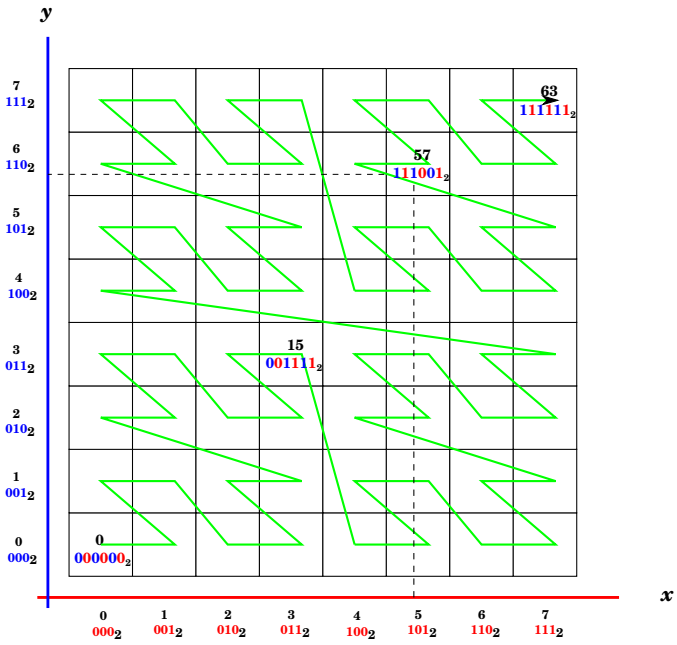

図 A·2 Morton Key と Morton 順序 (8 分割の場合)

る. 具体的には各軸の座標を求め, その bit 列を interleave させればよい.