Vol.2016-HPC-153 No.23
2016/3/2

階層的多体問題向けプログラミングフレームワーク Tapas

福田　圭祐[1,2,a)]　松田　元彦[1]　丸山　直也[1,2]　横田　理央[2]　田浦　健次朗[3]　松岡　聡[2]

概要：
階層的多体問題向けプログラミングフレームワーク Tapas を提案する。Tapas は高生産性・高効率・実用的の 3 つの目的を持って開発されている C++フレームワークであり、プログラマがアプリケーションの本質的部分（計算カーネル）に集中できるように、並列化などの非本質的部分を隠蔽・自動化する。特に，ユーザーの記述した計算ルーチンから C++言語標準の機能を用いて分散メモリ並列時における LET の構築を自動化することが可能である．性能面では，既存の高速な FMM(Fast Multipole Method) 実装である ExaFMM を Tapas 上に移植し TSUBAME2.5 上で評価した結果，現時点での初期実装では ExaFMM と比べて約 30 % ～50 % の性能であった

1. はじめに

多数の粒子の相互作用を計算する多体問題を効率的に計算するために、Barnes-Hut や Fast Multipole Method（FMM）などに代表される階層的な手法が用いられる。これらのアルゴリズムは、計算オーダーを直接計算の $O(N^2)$ から $O(N \log N)$ や $O(N)$ へと低減でき、さらに通信局所性の高さや計算密度の高さから大規模計算環境において有利とされている。さらに、従来の天文学や分子動力学計算に加え，近年では境界要素法（BEM）の前処理などの新しい応用が提案されており、注目が高まっている。

しかし，アプリケーション分野の研究者がこのようなプログラミングを行うにあたっては大きなコストがかかる．なぜなら，これらの階層的アルゴリズムは木構造ベースで実装されることが多く，実装が煩雑となり，共有メモリ・分散メモリ・GPU 利用などの並列化・高速化作業は高度な技術が必要となるからである．ライブラリ化などのコード再利用が望まれるが，アプリケーションごとに細部が異なり，それにより特に分散メモリ並列化の共通化が難しい．現在，多くの実装で MPI と C++，Fortran を用いてスクラッチからの実装がなされている [2], [5], [6], [18], [19], [21]．また，データ構造が複雑であることからコードの各所の書き換え・アルゴリズムの入れ替えに大きなコストがかかり，部分ごとの様々な手法の比較評価が難しいという問題もある．

1　理化学研究所計算科学研究機構
2　東京工業大学
3　東京大学
a)　fukuda@matsulab.is.titech.ac.jp

そこで我々は，階層的多体問題向けフレームワーク Tapas（Parallel Framework for Tree-based Adaptively PArtitioned Space）を提案する．Tapas は，高生産性・高性能・実用性の 3 つを目標として開発されている C++フレームワークであり，並列化などの非本質的な要素を隠蔽・自動化し，プログラマがアプリケーションの本質的な処理の記述に注力することができるようにする．隠蔽された下位レイヤの実装は透過的に切り替え可能で，様々な実装手法の比較検討が用意になるように設計されている．

本稿では，Tapas の設計と実装について述べ，FMM の高速な既存実装である ExaFMM の計算カーネルを Tapas 上に移植し性能を評価する．Tapas を用いて実装された FMM は，ExaFMM と比較して 20%～50%程度の性能を示した．

2. Tapas の設計

本節では，Tapas の設計方針とプログラミングモデルについて述べる．まず設計目標である「高性能・高生産性・実用性」について説明する．次に，ユーザーが Tapas 上でプログラムを作成する際のプログラミングモデルを述べる．

2.1 設計目標

Tapas は，高生産性，高性能，実用性を目標に開発されている．これらの 3 つの目標のそれぞれについて詳細に述べる．

2.1.1 高生産性

プログラム開発の生産性の定義は種々あるが，我々は「関心事の分離」を生産性の指標とする．プログラマが関

心を持つのはプログラムの本質的な処理の部分であって，それ以外の部分については気にせず書くことができることをもって高生産性と呼ぶ．

プログラムの本質とは，アプリケーションの計算カーネル，およびそれらの計算カーネルを木構造に適用する条件判定部分である．一方で，非本質的部分とは，メモリ上でのデータ構造の維持管理と，高性能を達成するための並列化である．ユーザーは，次節で述べるプログラミングモデルに従って一定の制約の元でプログラムを書く．これにより，ユーザーは木構造のメモリ上での具体的な構造，それらの作成／管理，共有メモリ並列化，分散メモリ並列化，GPU 化という非本質的部分について気にかけることなくプログラムを書くことができる．

2.1.2 高性能

Tapas が対象とするアプリケーションは，当然高性能が求められる．Tapas の性能評価にあたっては，4 で述べるように，ExaFMM から計算カーネルおよび最低限のトラバースのコードを抜き出し，それを Tapas 上に移植したもの（TapasFMM と呼ぶ）を用いて評価する．ExaFMM は既存の FMM 実装のうち最も高速なもののうちの 1 つである [21]．ExaFMM に対し，最終的に 8 割程度の速度を達成することを目指す．

2.1.3 実用性

実用性にも重点を置いて開発されている．本稿において実用性とは，実際のアプリケーション開発に採用されやすいことを指す．この点を定量的に述べることは難しく，またフレームワークの設計のみが採用されやすさを決めるわけではないが，Tapas では以下の点に留意して設計・開発が行われている．

- オープンソースとすること
- サードパーティのツールやライブラリへの依存を避けること
- 幅広い環境で動作すること
- ソフトウェアスタックが単純でメンテナンスがしやすいこと

これらの要件を満たすため，Tapas は依存するソフトウェアを言語標準のコンパイラのみとした．サードパーティのツールや，メンテナンスのコストを大幅に増大させるソースコード変換器などは用いず，プログラミング言語の標準規格の範囲内での実装を目指した．

2.2 プログラミングモデル

Tapas においては，ユーザーは，以下ように仮定し（あるいは仮定せず）プログラミングを行う．

- 空間は再帰的に分割され木構造をなしており，木のノードは Cell として抽象化されている
- 木の葉には，当該の部分空間内の粒子が所属する．
- 木構造がメモリ上でどのように実装されているのかは関知しない
- 計算は，木を構成する Cell のうち，親子の関係にある 2 つの Cell の間で行う
- 木を上方向・下方向にトラバースする必要がある場合は，子に対して再帰的関数呼び出しを行うことにより実現する．
- Cell および粒子へのランダムアクセスはできない．木の幅，深さなどのサイズを陽に取得することもできない
- 並列性は，後述する Map 関数，Accumulate 関数を用いることで暗黙的に記述される

ユーザーは，上記のような条件のもとで，Tapas が提供する Map 関数／ Accumulate 関数を用いてプログラムを記述する．

Map 関数は，ユーザーが指定するコールバック関数，関数適用の対象となる Cell などの要素の集合，その他の追加の引数を受け取る関数であり，暗黙に要素の集合間での並列性を仮定する．この関数は，プロセス間のデータ通信やスレッド並列の処理などを隠蔽してユーザーに提供している．Map 関数には「1 要素 Map」と「2 要素 Map」があり，Tapas によって生成された木構造のルートを表す Cell に対して適用することにより処理を開始できる．Accumulate 関数は，Map 関数内で呼び出される補助的関数であり，集約の操作を抽象化している．

2.3 データ型の定義

ユーザーは，3 つのデータ型を Tapas に与える．第一に，粒子の座標などの値を含む Body 型である．主に座標の値を含むことが想定され，Tapas が関与する範疇では読み取り専用であることが期待される．第二に，粒子に対する計算結果を格納する BodyAttr 型である．この型の値は粒子 1 つにつき 1 つ付加され，加速度やポテンシャルなどの計算によって更新される値を格納されることが期待されている．最後に CellAttr である．これは Cell クラスに付属する値であり，計算によって更新されることが期待される．本稿で例に上げる ExaFMM の移植実装においては，粒子の情報が集約された M ベクトルや L ベクトルがこれにあたる．

2.3.1 1 要素 Map

1 要素 Map は，ある Cell の子 Cell の集合に対して呼び出される Map 関数であり，コールバック関数は 1 つの Cell 要素を受け取る．再帰的に 1 要素 Map を用いることにより，行きがけ順（Pre-order）トラバーサル，帰りがけ順（Post-order）トラバーサルが実現できる．図 1，図 2 は，それぞれ Pre-order トラバーサル，Post-order トラバーサルの擬似コードである．なお，ある Cell の子は Cell の集合としてのみ扱うことができ，ループやインデックスアクセスを用いた個別処理は許可されず，いわゆる行きがけ順

図 1　Pre-order トラバーサルの記述例

```
function UserFuncPreOrder(Cell C) {
    doSomething(C)
    Map(UserFuncPreOrder, children(C))
}
```

図 2　Post-order トラバーサルの記述例

```
1 function UserFuncPostOrder(Cell C) {
2   Map(UserFuncPostOrder, children(C))
3   doSomething(C)
4 }
```

図 3　2 要素 Map の記述例

```
1 function UserFunc2(Cell C1, Cell C2) {
2   doSomething(C1, C2)
3   if (...) {
4       Map(UserFunc2, Product(children(C1), C2))
5   }
6   else if (...) {
7       Map(UserFunc2, Product(C1, children(C2))
8   }
9 }
```

(in-order) トラバーサルはサポートされない.

2.3.2　2 要素 Map

多体問題のアルゴリズムにおいては，親子でない要素間の相互作用を計算する必要がある．例えば，Barnes-Hut では，粒子と遠方の集約された擬似粒子との間で相互作用が行われるし，FMM においては M2L や P2P と呼ばれるようなフェーズで遠方・近距離の Cell 同士の演算が行われる．このような処理を抽象化したものが 2 要素 Map である．2 要素 Map の呼び出し引数は 1 要素 Map と同じであるが，要素の部分には Product という手続きを用いる．これにより，2 つの要素集合の積に対してコールバック関数を適用することができる．これは，いくつかの FMM 実装においては Dual Tree Traversal として知られている手法である [21]．図 3 は，2 要素 Map の利用の例を示した擬似コードである．

2.3.3　Accumulate

Accumulate 関数は，Map 関数内での集約操作を抽象化した関数である．木構造を用いたアプリケーションにおいては，葉から根に向かって情報を集約していく操作は頻出である．並列化の観点から見ると，集約操作は排他制御が必要であるし，フレームワークの API 設計の観点から見ても，要素へのインデックスアクセスおよびループ処理は抽象化と処理の隠蔽を破壊するおそれがあるので禁止する必要がある．よって，Tapas では集約操作を抽象化した Accumulate 関数を提供する（図 4）.

図 4　Accumulate の使用例

```
1 function UserFuncMap2(Cell Parent, Cell Child) {
2     ...
3     // attr(Cell)はユーザー定義の型の値を返す
          CellAttr
4     Accumulate(attr(Parent), attr(Child))
5 }
```

図 5　ユーザーによる Tapas テンプレートの具体化

```
1 struct FMM_Params : public tapas::HOT<3, real_t,
      Body, kBodyCoordOffset, kvec4, CellAttr> {
2 using Threading = FMM_Threading;
3 };
4
5 using TapasFMM = tapas::Tapas2<FMM_Params>;
```

3.　実装

本セクションでは，Tapas フレームワークの実装について述べる．まず，実装に用いる言語および全体像について述べ，続いて重要な点（共有メモリ並列化，分散メモリ並列化）について個別に説明する．特に，分散メモリ並列化は実装において最も困難な部分である．

3.1　実装の全体像

実装には，C++言語（C++11）を用いる．理由は

- 言語標準が存在し優れた処理系が多く存在すること
- HPC 環境で広くサポートされていること
- 強力なメタプログラミング機能を持つこと

である．これらの性質によって，追加ソフトウェアやプログラム変換等を用いずにフレームワークを実装することが可能となり，高い実用性を達成することができると考える．

ユーザーによって定義されう 3 つの型は，Tapas によってデータコピーが行われるため，ネストされたデータ構造，std::vector を始めとするポインタを含んだデータ構造，抽象基底クラスであってはならない．ユーザーは，これらの型を Tapas が提供するクラステンプレートの型引数として定義する．

3.2　プロセス内並列化

プロセス内の並列化は，1 要素 Map および 2 要素 Map の呼び出しを，複数のスレッドにより並列処理することにより実現する．木のトラバーサルの際には必ず Cell の子の集合に対して Map が呼び出され，子 Cell への関数適用の間には暗黙の並列性を仮定する．

ただし，この並列化は非常に細粒度の並列化となるため，ここでは ExaFMM において実績のある [15] タスク並列モデルを用いる．詳細は参考文献 [15] に譲るが，1 要素 Map

については単純に子 Cell への呼び出しごとにタスクを生成し，2 要素 Map においては Accumulate 処理に排他制御が不要になるように，2 つの集合間の組み合わせ表において，「田」の形に再帰的に領域分割を行いながら，要素に重複がない排他的な組同士を並列に実行する方式を用いて効率の向上を図っている．

3.3 LET を用いた分散メモリ並列化

本稿で述べる Tapas の実装において最も困難であるのが分散メモリ並列化である．この節ではまず，一般的に木構造を分散メモリ上に実装する際の課題について述べ，一般的に用いられている手法（LET）を説明する．次に，それを Tapas において自動化するにあたっての課題と，その解決手法を述べる．

3.3.1 一般的な LET の実装

一般的に階層的粒子法の分散実装においては，

(1) 木は平衡木とは限らないため，他プロセスにある木がどのような構造をしているかは通信をしないとわからない．各プロセスが持つ木の全てを互いに転送するのは現実的に不可能である．
(2) 他プロセスから転送する必要があるデータが実行時パラメーターに依存する

という特徴がある．交換必要のあるデータ，およびその手続きを総称して LET（Locally Essential Tree）と呼び，これには次のようなステップを踏む．

(1) 他のプロセスにある木を，推測もしくは近似する．場合によっては少量の通信を行う
(2) 近似した他プロセスの木と自分が保持する木の間で最低限のトラバーサルの処理を行い，転送の必要なデータをリストアップする．場合によっては近似となるが，保守的に余分にリストを作る．
(3) 決定したデータを互いに送受信する

この時，2 つ目のステップで行うトラバース処理では実際の計算処理はおこなわず，木をトラバースするために必要最低限の処理のみを行う．一般的に，これは計算機科学において Inspector/Executor パターンもしくは Inspector/Executor モデル（I/E モデル）と呼ばれる．Inspector と Executor は，トラバースの条件は同じであるが計算の有無に違いがあるため別々のコードとして記述される．

I/E モデルは，Tapas が対象とする階層的粒子法にかぎらず，不規則なデータアクセスもしくは間接参照が行われるアプリケーションにおいて，並列性の向上や局所性の改善を目的として用いられるテクニックである．Executor である計算コードが所与であるとして，Inspector によってデータのコピー等を事前に行うことにより性能の向上を図るのが目的である．この時，既存研究においては，手動で Inspector コードを作成するか，コンパイラによって Inspector を自動生成することが行われている．

3.3.2 C++テンプレートメタプログラミングを用いた I/E モデルの実現

Tapas においては，ランダムアクセスを用いず木構造を再帰的にトラバースするという制約以外は幅広いアルゴリズムを記述できるため，事前に Inspector コードを準備しておくことは不可能である．また，2.1.3 節で述べたように，設計方針として，外部のツールやライブラリに依存せず，ソフトウェアスタックを単純にすることを目指している．このため，ソースコードを変換する Source to Source コンパイラや，LLVM 等のツールは用いない設計を目指しており，言語標準準拠のコンパイラによってコンパイルされるライブラリとして設計する．よって，C++言語によるメタプログラミングによって Executor コードから Inspector コードを生成できるかどうかが，Tapas において最も技術的に困難な点である．

この問題を解決するため，Tapas では C++テンプレートを用いたメタプログラミング技術を利用する．ユーザーが与える計算コードを関数オブジェクトのテンプレートとして作成してもらうこととし，Inspector を実現するためのダミーのクラスを与えて Inspector を静的に生成する．図 6，7 は本稿で実装した ExaFMM の Tapas 上へ移植したプログラムの一部であり，2 要素 Map によって呼び出されるユーザー定義関数の 1 つである．木をトラバースする関数と，2 つの Cell 間の演算である M2L と呼ばれる関数を簡略化したものである．

前述したユーザーが定義する 3 つのデータ型のうち，図 7 中の＊ 1 で示される CellAttr クラスである．ただしユーザーが与えた型そのものではなく，テンプレートのパラメータとして与えられた Cell クラスの中から導出された Cell::CellAttr クラスとして利用している．次に，計算を行い CellAttr へと代入する時点において，CellAttr の属性（ 図 6, 7 中では L などである）に代入を行うのではなく，CellAttr を一時変数にコピーし，それに対して計算を行った上で CellAttr ごと代入演算子を用いるように書く．Tapas は Inspector 生成時にはテンプレートパラメーターである_Cell と_CellAttr にユーザー定義の型を模したダミーのクラスを与え，そのクラスに対しては operator= 演算子を再定義し何もしない関数に置き換えてある．これにより，コンパイラによりテンプレートのインスタンス化によって生成された Inspector コードは，代入文が存在しないことによる dead code elimination 最適化に寄ってコンパイラが除去することが期待される．これにより，一方で，木をトラバースするための再帰的な条件判定のための計算コードは除去されず，通常通りに動作する．これによって，ユーザーが提供した Executor から，トラバースのみを行う Inspector を生成することができる．

上記の説明では 2 要素 Map の LET 構築について述べたが，1 要素 Map においても，同様の Inspector 生成による

図 6 ユーザーコードの記述例（トラバース条件部）

```
1  // を上に移植した ExaFMMTapas
2  // コードの一部を簡略化したもの
3  struct FMM_DTT {
4    template<class Cell>
5    inline void operator()(Cell &Ci, Cell &Cj, int
         mutual,
6                           int nspawn, real_t theta)
                                {
7      vec3 dX;
8      asn(dX, Ci.center() − Cj.center());
9      real_t R2 = norm(dX);
10     vec3 Xperiodic = 0; // dummy; periodic not
           ported
11
12     real_t Ri = 0;
13     real_t Rj = 0;
14
15     for (int d = 0; d < 3; d++) {
16       Ri = std::max(Ri, Ci.width(d));
17       Rj = std::max(Rj, Cj.width(d));
18     }
19     Ri = (Ri / 2 ∗ 1.00001) / theta;
20     Rj = (Rj / 2 ∗ 1.00001) / theta;
21
22     if (R2 > (Ri + Rj) ∗ (Ri + Rj)) {
23       numM2L++;
24       M2L(Ci, Cj, Xperiodic, mutual);
25     } else if (Ci.IsLeaf() && Cj.IsLeaf()) {
26       tapas::Map(P2P(), tapas::Product(Ci.bodies(),
             Cj.bodies()),
27                  Xperiodic, mutual);
28
29       numP2P++;
30     } else {
31       tapas_splitCell(Ci, Cj, Ri, Rj, mutual, nspawn,
             theta);
32     }
33   }
34 };
```

図 7 ユーザーコードの記述（演算部）

```
1  template<class Cell>
2  void M2L(Cell &Ci, Cell &Cj, vec3 Xperiodic, bool
       mutual) {
3    complex_t Ynmi[P∗P], Ynmj[P∗P];
4    vec3 dX;
5    asn(dX, Ci.center() − Cj.center());
6    dX −= Xperiodic;
7
8    // ∗ 1 operator= オーバーロードのための変数
         コピー
9    typename Cell::CellAttr attr_i = Ci.attr();
10   typename Cell::CellAttr attr_j = Cj.attr();
11
12   for (int j=0; j<P; j++) {
13     real_t Cnm = ODDEVEN(j);
14
15     for (int k=0; k<=j; k++) {
16     int jks = j ∗ (j + 1) / 2 + k;
17     complex_t Li = 0, Lj = 0;
18     for (int n=0; n<P−j; n++) {
19       for (int m=−n; m<0; m++) {
20       // 計算コード...
21       Li += std::conj(Cj.attr().M[nms]) ∗ Cnm ∗
             Ynmi[jnkm];
22       // ...
23     }
24     for (int m=0; m<=n; m++) {
25       // ...
26       Li += Cj.attr().M[nms] ∗ Cnm2 ∗ Ynmi[jnkm];
27       // ...
28     }
29
30     attr_i.L[jks] += Li;
31     }
32   }
33   Ci.attr() = attr_i; // データ型ごと代入演算子を用い
         て代入 CellAttr
34 }
```

I/E モデルを用いることができる（ただし，後述のように本稿執筆時点では未実装である）．1 要素 Map は，Pre-order（Downward）トラバーサル，Post-order（Upward）トラバーサルを実現するために用いられる．木構造を分散メモリ並列化するにあたっては，木のメモリ上での具体的な構造にもよるが，根に近い範囲を全プロセスでコピーを保持して共有し（グローバル木），それ以外の部分を書くプロセスで独自に保持する（ローカル木）ということが行われる．この時，特に帰りがけ順においては，各プロセスでローカル木に対してトラバーサルを行い，グローバル木の葉に相当する部分まで計算を行ったらグローバル木の葉の値を交換し，続いて再び各プロセスがグローバル木についてトラバースを行うという処理を行う．一方，Pre-order トラバーサルではこのような処理は不要である．よって，ユーザーが記述した 1 要素 Map が，Pre-order なのか Post-order なのかという区別を行う必要があるが，前述の I/E モデルと同等のテクニックにより，判定が可能である..

3.3.3 Push-based LET と Pull-based LET

I/E モデルにより他プロセスと交換する必要がある Cell の選定の際，送られるデータを保持しておりデータ送信を行う側のプロセス（Sender）と，データを受け取って後のトラバースに使用するプロセス（Receiver）が存在する．なお，すべてのプロセスが，他のすべてのプロセス対して同時に Sender でもあり Receiver でもある．この時，Inspector は，相手プロセスの保持する木を何らかの方法で近似し，その近似された木を用いてトラバースを行う．このトラバースは，Sender が行うことも Receiver が行うことも可能である．本稿では，Sender がトラバースを行う前者を Push-based LET 構築，Receiver がトラバースを行う後者を Pull-based LET 構築と呼ぶ．既存実装の殆どは Push-based　LET 構築を用いているが，Tapas では Pull-based LET 構築を採用した．Pull-based LET では，Executor による実際のトラバースを行う Receiver が計算に必要なデータをリストアップすることにより，トラバース処理のタスクグラフを早期に構築でき，通信と計算のオーバーラップおよび柔軟なタスクスケジューリングが可能となるからである．これらは将来の課題である．

3.3.4 LET 構築の具体的な計算内容

木構造ベース粒子法における LET の構築方法については，先行研究で数多く手法が提案されているが [2], [3], [4], [16], [17]，執筆時点の Tapas では比較的単純な戦略を用いている．前述のように Tapas では将来のスケジューリングへの拡張性を考慮して Pull-based LET を用いており，その内容の内訳は以下のようになっている．なお，各項目の末尾に書かれた名称は評価の内訳の名称として使われている．

(1) 各プロセスで，仮想的な全体木について Inspector を実行し，必要な Cell を列挙する（LET-Trav）
(2) 列挙された Cell の所属プロセスを検索し，当該プロセスへデータ送信のリクエストを送出する（LET-Req）
(3) 各プロセスが受け取ったリクエストを解析し，自分が保有する Cell を照らしあわせ，リクエストに応答してデータを送信する（LET-Res）
(4) 受信したデータをデータ構造に登録する（LET-Reg）

3.4 実装の現状と制約

実装の制約について述べる．フレームワークとしての抽象化に関わる本質的な制約としては，現状の実装では CPU 向けの SIMD 化は対応していない．現時点までの調査により，プログラマが粒子同士の計算カーネルを与える現状の Tapas のインターフェースではコンパイラによる自動 SIMD 化およびコンパイラ依存の pragma 等を用いた半自動 SIMD 化は達成できていない．CPU における SIMD 化をサポートするために，オプショナルな追加インターフェースを用いることでプログラマが SIMD コードを記述できるようにすることを検討している．

1 要素 Map における I/E モデルの実装は不完全である．評価に用いた ExaFMM では 1 要素 Map は 2 通り存在し，P2M と M2M からなる Upward フェーズ，L2L と L2P からなる Downward フェーズである．一般的には，Upward は Post-order トラバーサル，Downward は Pre-order トラバーサルに相当する．Tapas においては，これら 2 種類の 1 要素 Map は異なる処理を行う必要があるため，再帰的な 1 要素 Map を用いて定義されたユーザーのプログラムから，Pre-order であるか Post-order であるかを判別する必要がある．2 要素 Map と同様の手法を用いて I/E モデルにより判定を行う予定であるが，本稿執筆時点では未実装のため，ユーザーが UpwardMap，DownwardMap と明示的に記述することとした．

フレームワークの抽象化もしくはインターフェースに関わらない実装上の制約としては，先行研究によって知られている各種最適化が未実装であることがあげられる．例としては，ExaFMM における R-opt, p4est[3] における空間分割時のヒューリスティクスなどである．これらの最適化は，ユーザーコードに影響をあたえることなく，Tapas 内部で透過的に実装可能である．

GPU 対応は実装中である．

4. 評価

4.1 評価の方法と環境

本セクションでは，性能評価の結果を述べる．評価対象として，ExaFMM を性能測定上の参照実装として性能の基準とした．ExaFMM は，既存の FMM 実装の中で，とりわけ低精度での実行において最も高速な実装の 1 つであることが示されている [20]．Tapas については，ExaFMM の計算カーネル（P2P, P2M, M2M, M2L, L2L, L2P）と，

それを実行するのに必要なトラバース用の関数を抜き出し，Tapas に移植した．FMM の各計算フェーズの具体的な意味については参考文献に譲る．移植にあたっては，計算ルーチンとトラバース関数それぞれについて，処理の内容は変えずに，文法上必要な変更を施した．

ExaFMM にはいくつかのアルゴリズム上および実装上の最適化が施されているが，そのうちのいくつかについては未実装であるので評価の対象とせず，OFF にした状態で実行した．具体的には，R-opt と CPU 向け SIMD 化である．なお，R-opt とは，木のノード（Tapas においては Cell）を，ノード内の粒子が実際に存在する範囲まで縮小する手法である．これにより，相対的に Cell 間が遠くなり，精度を落とすことなく近似計算の割合を増やすことができ，全体の高速化に繋がる．R-opt については，Tapas でも実装可能であり，将来の課題でる．SIMD 化については，SIMD 化を望むユーザー向けにオプショナルな追加インターフェースを追加することで可能だと考えており，検討中である

実行には，TSUBAME2.5 の Thin ノードを用い，Open MPI（1.8.2）, Intel C++ Compiler （2013.1.046），MassiveThreads（執筆時点の最新版）を用いた．計算カーネルの種類は Spherical を用い，計測は 5 回行い平均値を用いた．

また，ExaFMM は，Tapas 開発時において Tapas への移植へ用いたバージョンである．ExaFMM はその後も開発が続けられており，本稿で掲載したバージョンよりも最適化が進んでいると考えられることを申し添えておく．

計算時間に影響を与える ExaFMM の実行時パラメーターとして θ がある．これは，木のノード（Tapas における Cell）同士の距離の判定の際，2 つのノード間の距離とそれぞれの直径との相対的な値による「遠近」の決定のしきい値である．θ が大きくなると近似の精度を下げる代わりに実行時間が短くなり，θ が小さくなるとその逆である．本稿で用いた ExaFMM 実装におけるデフォルト値は 0.4 である．評価にあたっては，Tapas に移植した FMM の θ を調整し，ExaFMM と誤差が同等となる $\theta = 0.39$ に調整した．また，もう 1 つの重要な実行時パラーメータとして N_{crit} がある．これは，木の末端の葉の大きさを決めるもので，木を構築するときにすべての葉がたかだか N_{crit} 個の粒子を持つように，再帰的に空間を分割する．N_{crit} が大きいと，近似計算に対して直接計算の割合が大きくなり，逆も同様である．本稿の実験では $N_{crit} = 64$ とした．

4.2 P2P 実行

前項で述べた N_{crit} の値を意図的に粒子数より大きくすると，空間分割が一回も行われず，実質的に N 体問題の直接計算を行うこととなる．これを評価した結果が図 8 である．

ExaFMM においては，P2P の計算は葉 Cell に属する粒子同士の間で二重ループを用いて書かれている．しかし，Tapas においては，ユーザーが単一の粒子と粒子の間の相互作用を計算する関数を記述し，それを Tapas がループによって粒子の対ごとに適用する．このような違いにより，コンパイラによる最適化に違いが現れ，Tapas の実行時間にはオーバーヘッドがある．具体的には，我々が評価に用いた Intel コンパイラにおいては，書き込み先粒子について値を足し込みする部分で，足し込み対象の変数がレジスタに割り当てられない現象が見られた．

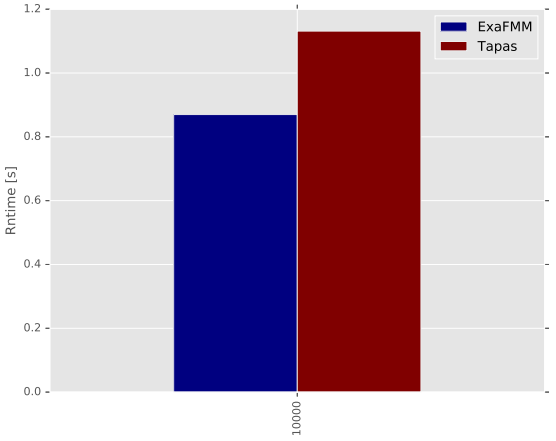

図 8 P2P 計算における ExaFMM と Tapas の比較
N=10,000, SIMD 無し

4.3 全体性能：強スケーリング

複数計算ノードにおける実行で，物理ノード内はタスク並列によるマルチスレッド，ノード間は MPI による分散メモリ並列を用いて強スケーリングを行った結果が図 9 である．ここで用いた粒子数である 100 万粒子はそれほど大きい粒子数ではなく，特に 10 ノード以上においては ExaFMM も十分にスケーリングしておらず，設定・実行方法の見直しも含め，より大きな粒子数での再評価は将来の課題である．Tapas は，1 ノードの時はデータ交換フェーズである LET 構築・通信を行っておらず，2 ノード以上では現状では大きなオーバーヘッドになっている．また，全体として 10 ノード付近では ExaFMM の 2 倍程度の時間がかかっている．

なお ExaFMM において強スケーリング・弱スケーリング時にそれぞれプロセス数 10,8 の時に大幅な実行時間の局所的な増加が見られているが，原因は調査中である．

強スケーリング時の Tapas における計算時間の内訳を示したものが図 10 である．計算の内訳は，

- Tree : 木構築
- Upward, Traverse, Downward : ExaFMM の Upward, Traverse, Downward と対応する，P2M/M2M,

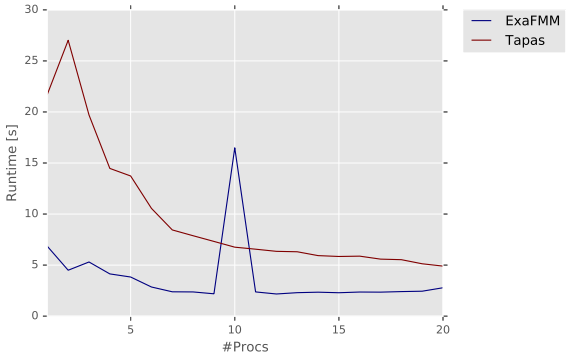

図 9　1,000,000 粒子 強スケーリング

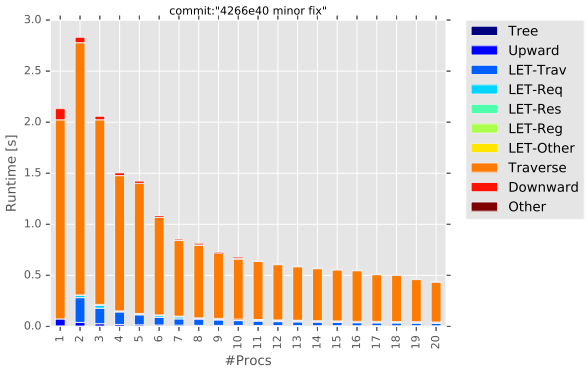

図 10　1,000,000 粒子 強スケーリング時の計算時間内訳

M2L/P2P, L2L/L2P の各フェーズの実行時間

- LET-*：Traverse のフェーズに先立って行われる構築の各フェーズの実行時間．

4.4　全体性能：弱スケーリング

強スケーリングと同様に，弱スケーリングを行った結果が図 11，図 12 である．3 プロセス以降で LET 構築の Traverse 時間（LET-Trav）の実行時間が大きな割合を占め，またプロセス数によって増減が激しいことがわかる．この点は，3.3.1 節で述べたリモートプロセスの木構造の近似・簡略化を行っていないこと，もしくはコンパイラにより不要コードの除去が不十分であることが原因と考えられる．原因と最適化は今後の課題である．リモートプロセスの木構造の簡略化については，先行研究で手法が多数提案されており，それを用いることで解決できると考えられる．

5.　関連研究

木構造ベースのシミュレーションコードの最初期のものとして P.Liu らによるもの [9] がある．これは C++で書かれたテンプレートベースライブラリであるという点で我々の提案と類似しているが，発表が 97 年と古く現在は入手不可能である．C.Liu らは，再帰的なデータ構造を持つ力学アプリケーションに対して，MassiveThreads と類似した処理系 Cilk を用いた最適化を与えた [8]．また，同じく階層的粒子法を対象とした C++フレームワークとして

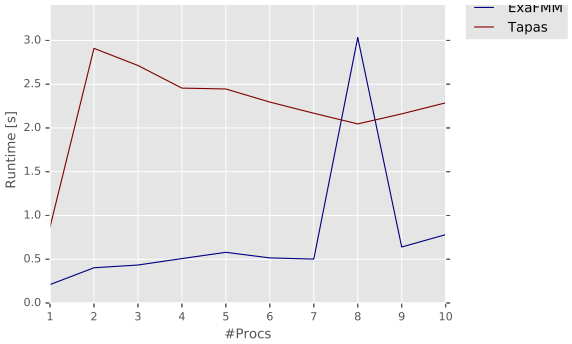

図 11　1,000,000 粒子/rank 弱スケーリング

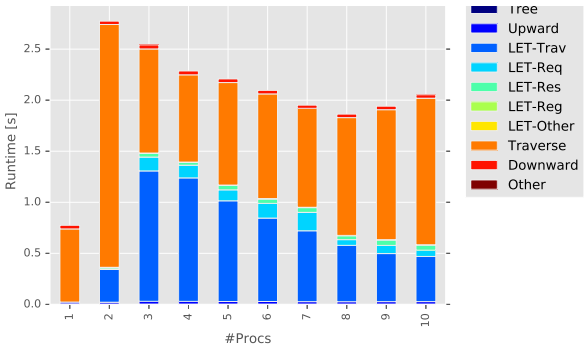

図 12　1,000,000 粒子/rank 弱スケーリング時の計算時間内訳

Sato らは Cosplit を提案している [13]．これは執筆時点では共有メモリ並列化のみの対応である．Iwasawa らによる FDPS[7] は粒子系シミュレーションのためのフレームワークであり，実績のある実装をベースとしており，非常に高速でスケーラブルに動作する一方アルゴリズムの選択の幅は限られている．

不規則なアプリケーション・アルゴリズムに対して，主にデータ局所性の向上による最適化を目的として Inspector/Executor モデルを適用した例としては，不規則な間接参照を含む配列上のループ最適化に用いられる例がある [1], [11], [14]．分散メモリ上の配列上のループにコンパイラベースの技法を適用した例としては [10], [12], [20] がある．

6.　まとめと今後の課題

本稿では，C++上に構築された階層的粒子法向けプログラミングフレームワーク Tapas を提案し，初期実装を示し，性能を評価した．機能面においては，ユーザーの記述した計算ルーチンから，C++言語標準の機能を用いて分散メモリ並列時における LET の構築を自動化できることを示した．性能面では，既存の高速な FMM 実装である ExaFMM の計算カーネルを Tapas 上に移植したものを評価し，ExaFMM と比べて約 30 % ～50 % の性能であり，さらなる最適化が必要であることがわかった．この最適化はフレームワークとしての抽象化部分とは別に評価可能であると考えられ，抽象化のオーバーヘッドを含めて 80%の

性能を目標として最適化を行っていく．また，3.4 節で述べた未実装機能の実装を行っていく．

また，ユーザーが与えるプログラムに変更を加えることなく，透過的に GPU 上での実行を可能にすることも将来の課題である．Barnes-Hut や FMM をはじめとする階層的粒子法では GPU などのアクセラレーターが有効であることが知られており，それらをサポートすることは現在主流を占める GPU スーパーコンピューターにおいて必須であると考えられる．

謝辞

本研究は、JST、CREST の支援を受けたものである。

参考文献

[1] Manuel Arenaz, Juan Tourio, and Ramn Doallo. An inspector-executor algorithm for irregular assignment parallelization. In *In Proc. of the 2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA*, pages 4–15. Springer, 2004.

[2] Jeroen Bédorf, Evghenii Gaburov, Michiko S. Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 54–65, Piscataway, NJ, USA, 2014. IEEE Press.

[3] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. p4est : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, January 2011.

[4] Felipe A. Cruz, Matthew G. Knepley, and L. A. Barba. PetFMM-A dynamically load-balancing parallel fast multipole library. *International Journal for Numerical Methods in Engineering*, 85(4):403–428, January 2011.

[5] William Fong and Eric Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228(23):8712–8725, December 2009.

[6] Tomoaki Ishiyama, Keigo Nitadori, and Junichiro Makino. 4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 5:1–5:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono, Keigo Nitadori, Takayuki Muranushi, and Junichiro Makino. Fdps: A novel framework for developing high-performance particle simulation codes for distributed-memory systems. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, pages 1:1–1:10, New York, NY, USA, 2015. ACM.

[8] Chenyang Liu, Muhammad Hasan Jamal, Milind Kulkarni, Arun Prakash, and Vijay Pai. Exploiting domain knowledge to optimize parallel computational mechanics codes. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 25–36, New York, NY, USA, 2013. ACM.

[9] Pangfeng Liu and Jan-Jan Wu. A framework for parallel tree-based scientific simulations. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, pages 137–144, Aug 1997.

[10] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Las Vegas, Nevada, USA, June 18-21, 1997*, pages 48–56, 1997.

[11] David Ozog, Sameer Shende, Allen Malony, Jeff R. Hammond, James Dinan, and Pavan Balaji. Inspector/executor load balancing algorithms for block-sparse tensor contractions. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 483–484, New York, NY, USA, 2013. ACM.

[12] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Distributed memory code generation for mixed irregular/regular computations. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 65–75, New York, NY, USA, 2015. ACM.

[13] Shigeyuki Sato and Kenjiro Taura. Abstraction of space partitioning for spatial computation. In *2015-2-(9): Manuscript for presentation at IPSJ-SIGPRO*, 2015.

[14] Jimmy Su and Katherine Yelick. Array prefetching for irregular array accesses in titanium. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 158. IEEE, 2004.

[15] Kenjiro Taura, Jun Nakashima, Rio Yokota, and Naoya Maruyama. A Task Parallelism Meets Fast Multipole Methods. In *Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, 2012.

[16] Shang-Hua Teng. Provably Good Partitioning and Load Balancing Algorithms for Parallel Adaptive N-Body Simulation. *SIAM Journal on Scientific Computing*, 19(2):635–656, March 1998.

[17] M. S. Warren and J. 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.

[18] Michael S. Warren. 2hot: An improved parallel hashed oct-tree n-body algorithm for cosmological simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 72:1–72:12, New York, NY, USA, 2013. ACM.

[19] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, May 2004.

[20] Daisuke Yokota, Shigeru Chiba, and Kozo Itano. A new optimization technique for the inspector-executor method. In *In Proc. of the International Conference on Parallel and Distributed Computing Systems (PDCS*, pages 706–711. ACTA Press, 2002.

[21] Rio Yokota. An FMM Based on Dual Tree Traversal for Many-core Architectures. *Journal of Algorithms and Computational Technology*, 7, September 2012.