Vol.2012-HPC-136 No.13
2012/10/3

動的タスクスケジューリングエンジンStarPUによるKIFMMの実装と性能評価

福田 圭祐[1,a)] 丸山 直也[1,2,3] Miquel Pericàs[1] 松岡 聡[1,3]

概要：Fast Multipole Method （FMM） は，N 体問題のアルゴリズムで，近似計算により $O(N)$ の計算量を実現する．FMM は，計算特性が異なり入力データによって負荷が変動する複数の計算ステップから構成される．本研究では, FMM の入力データ（粒子分布）による負荷変動に対して CPU/GPU 間の負荷分散を適切に行うことを目的とする．そのための手法として，動的タスクスケジューリングエンジンを採用し，そのためのライブラリである StarPU 上に Kernel Independent FMM（KIFMM）アプリケーションを実装し，性能を評価した．この実装を，入力データ毎の総当たりによって最適な静的スケジューリングを決定することができる実装と比較した．均一分散では単純なヒューリスティクスを１つ導入することにより静的スケジューリング実装に対して 137.9%，球表面（不均一）分散においてはヒューリスティクスを用いずに同実装に対して 89.5%の性能を得た．このことから，動的タスクスケジューリングを用いることにより，最適な静的スケジューリング実装に対して競争的なパフォーマンスを発揮しつつ，入力データによる負荷変動に抗して負荷分散を実現することが可能であると言える．

キーワード：FMM, StarPU, 動的タスクスケジューリング

1. はじめに

スーパーコンピューティングのための環境として，CPU/GPU 混在環境が広く受け入れられつつある.GPU を用いたスーパーコンピューターとして，東工大学術国際情報センターの TSUBAME2.0，筑波大学計算科学研究センターの HA-PACS, 米国オークリッジ国立研究所の Titan 等が知られている．

GPU は，多数のプロセッサコアを用いた高い演算性能と，GDDR メモリによる高いメモリバンド幅を提供し，アプリケーションによっては大幅な高速化が達成できることが知られている．しかし一方で，計算による向き／不向きが存在し，ホスト PC とメモリ空間が別であるなどの制約もある．さらに，アプリケーションが複数の計算ステップから構成され，それぞれが異なる計算特性を持つようなアプリケーションを GPU により加速することを考えると，GPU による加速倍率はステップごとに異なり，それぞれのステップを CPU/GPU のどちらで実行すれば良いのかを決定することは容易ではない．個々のステップの実行速度に加え，ステップ間の依存性とデータ転送について考慮しなければいけない．

このようなアプリケーションの例として，FMM が挙げられる．本稿では，FMM の１実装である kifmm3d[1] を扱う.kifmm3d は，評価フェーズとして８つの主要な計算ステップ（U-list, P2M, M2M, V-list, W-list, X-list, L2L, L2P）が存在し，それらが依存関係によりつながれている．さらに，それぞれが違った計算特性を持ち，入力データ（粒子配置）によって各ステップの計算量は変化する．詳細は 2 章で述べるが，例えば U-list（P2P とも呼ばれる）は massively parallel な N 体問題の直接計算であり，一方 V-list は比較的不規則なデータ構造と，FFT 計算を含む複雑なフェーズである．よって，これらの計算ステップの GPU による加速割合も異なる．

我々は既存発表 [2] において，FMM において良好な負荷分散を図るための手法として動的タスクジューリングを用いることを提案し，具体的な方針として kifmm3d を StarPU 上に実装するための基本設計と実装上の障害について報告した．動的タスクスケジューリングでは，計算の各ステップを「タスク」として定義する．それらのタスクに対し，入出力のデータ，依存関係等を設定し，実行時にタスクをプロセッサ間で分担して実行する．本研究では，その提案実装を完了させると同時に，性能評価のため，計算

[1] 東工大
Tokyo Institute of Technology
[2] 理研 計算科学研究機構
[3] JST/CREST
[a)] fukuda@matsulab.is.titech.ac.jp

ⓒ 2012 Information Processing Society of Japan

の最適な割り当てを静的に総当たりによって決定できる実装を用意し，性能の比較対象とした.

本稿は，以下のように構成される. まず，2 節で FMM，3 節で動的スケジューリングエンジンである StarPU について述べる. その後，4 節において具体的な実装について述べ，5 節で評価を行い，6 節で関連研究について述べた後，最後に 7 節でまとめと今後の課題を述べる.

2. FMM

多数の粒子系に対して粒子間の全相互作用を計算するような問題を N 体問題と呼ぶ.N 体問題の最も初歩的な解法は相互作用をすべて直接計算する方法であるが，これは計算量が粒子数 N に対して $O(N^2)$ となるため，粒子数の増加に対して計算量が急激に増加する. そこで，ある程度の誤差を許容し近似的に計算を行うことによって計算量を下げる手法が研究されており，Greengard ら [1] によって提案された FMM はそのうちの 1 つである. これは，空間を再帰的に分割し木構造を構築したあと，近傍粒子の計算と遠方粒子の計算を分けて考え，遠方の粒子同士の演算を「まとめて」行うことによって $O(N)$ での計算を可能にする. さらに，通信の大部分が局所的でもあることから，今後の大規模計算において高いスケーラビリティを持つ重要なアルゴリズムとなることが期待されている. 大規模な実行例として，横田らによる乱流計算 [3] やら Rahimian らによる血流シミュレーション [4] などが知られている.

本稿では，Ying ら [5] によって FMM の派生アルゴリズムである Kernel Independent FMM（KIFMM）を扱い，その実装である kifmm3d[6] を用いる. 次項ではその概要について述べるにとどめ，詳細については [1], [5] を参照されたい.

kifmm3d は，KIFMM の参照実装として作成された. 粒子分布データを入力として受け取り，空間を分割して木構造を作成し，粒子の相互作用計算を 1 ステップ行って終了する. ここでは，前者を「木構築フェーズ」，後者を「評価フェーズ」と呼ぶ. 本稿では主に評価フェーズに焦点を当て，木構築フェーズについては必要な程度で概要を述べる.

2.1 木構築フェーズ

木構築フェーズでは，空間を再帰的に分割して木構造を作成する. 空間の各次元について 2 等分に分けるので，3 次元では 8 等分に分けることになり，8 分木（octree）が構成される. 分割は，各部分空間について再帰的に行われ，末端の葉が高々 q 個の粒子しかもたないようになるまで分割を継続する. ここで，粒子が均一分散で無い場合，粒子が密集しているところは木の深さが深く，粒子が疎なところは木が浅くなる. このとき，このような木構造と計算の性質を以下 adaptive であると呼ぶ. kifmm3d は adaptive な処理を行う FMM 実装である.

図 1 Flow and dependencies of KIFMM's evaluation phases

2.2 評価フェーズ

次に，評価フェーズには，U-list（P2P）, P2M, M2M, V-list（M2L）, W-list, X-list, L2L, L2P の 8 つのステップが存在する[*1]. ステップは，近傍直接計算（U-list）と遠方近似計算（それ以外）に大別される. U-list は P2P とも呼ばれ，N 体問題の直接計算と同等の計算であり，また massively parallel な演算密度の高い処理であるため GPU に適していると考えられる. U-list を除く 7 つの処理が遠方粒子を近似計算する部分である. まず P2M によって octree の葉に属する粒子を「まとめ」，それをさらに L2P ステップによって木の根に向かって集約していく. 次に V-list ステップにおいて「まとめた」粒子同士の遠方相互作用を近似的に計算する. 次に，計算された遠方相互作用を L2L ステップによって木の下方へ伝搬していき，最終的に L2P ステップによって葉から粒子へ伝達される.W-list と X-list は，木の構造が adaptive である場合に葉の処理の一部をおこなうステップである. これら一連の遠方粒子同士の近似計算は木構造を扱うため，U-list と比較するとデータ構造がやや不規則であり，計算密度が低い傾向にある. **図 1** は，処理の流れと各ステップ間の全体像を図示したものである.

2.3 計算量とパラメーター

各ステップの計算量を表**図 1** に示した. ここで，q は空間分割によって生成される octree の深さを決定し，アルゴリズムの各ステップの計算量に大きな影響を及ぼす重要なパラメーターである. 前述のように，q が大きいとき空間分割と octree は浅くなる. これは，近似計算の程度をへらし，直接計算の割合を増やすことになる.octree の深さが 1，つまり一回も分割を行わない場合がすなわち直接計算に他ならない. 逆に，q が小さい場合，octree の深さが深くなり，より近似計算の割合が多くなる. これにより，粒子分布だけでなく q の選び方によっても計算ステップの計算時間の割合が変動することがわかる.

3. 動的タスクスケジューリングエンジン

FMM において，

- プログラムが依存関係で結ばれた複数の計算ステップからなる
- 各ステップは異なる計算特性を持ち，GPU による加

[*1] なお，kifmm3d では P2M と M2M，L2L と L2P はそれぞれ Upward, Downward と呼ばれ同じステップとされている. しかし，計算内容が異なる上，ExaFMM[7] 等他の FMM 実装においても別処理として扱われていることから別処理とした.

ⓒ 2012 Information Processing Society of Japan

速率が異なる
- 各種プロセッサでの実行に伴い，データ移動を適切に管理する必要があり，その処理が煩雑である
- 各ステップの実行時間は入力データに依存する

という課題が存在することは既に述べた. 本稿では，これらの課題に対する解決手法として動的タスクスケジューリングエンジン上にアプリケーションを構築することを提案する. また，実現のための動的タスクスケジューリングエンジンとして，API ベースのライブラリである StarPU[8] を採用する.

他の手法として，事前に自動チューニングを行うという手法も考えられる. この場合は各ステップを実行するプロセッサを決定するという点においては解決が期待できるが，データ転送処理とその最適化が煩雑になる点，実行時に入力データに依存する点が解決できない.

動的タスクスケジューリングエンジンは，処理をタスクという単位で記述し，実行時に動的に実行する場所を選んで実行するランタイムシステムである. StarPU において，プログラムは `starpu task` という実行単位で管理される. タスクは，`starpu codelet` と呼ばれる関数の実体，入出力データへの参照，他タスクとの依存関係等の情報を保持する．1 つのタスクは，CPU 用と CUDA 用の codelet を持つことが出来，実行時に StarPU ランタイムが自動的に判断してどちらかのプロセッサに割り当てられる. StarPU は，ミドルウェアやライブラリに用いられることが想定されているライブラリであり, プログラマが明示的にタスクを定義する必要があるという点では煩雑である．本実装で用いた task 定義の例を下に掲載した.

```
bzero(&l2l_cl, sizeof(l2l_cl));
l2l_cl.where = STARPU_CUDA | STARPU_CPU;
l2l_cl.cpu_funcs[0] = &call_eval_l2l_cpu;
l2l_cl.cuda_funcs[0] = &call_eval_l2l_cuda;
l2l_cl.nbuffers = 2;
l2l_cl.model = &perf_model_l2l;
l2l_cl.modes[0] = STARPU_R;
l2l_cl.modes[1] = STARPU_RW;
```

表 1 各フェーズの計算量
Table 1 Computational complexity of each phase

P2M	$O(Np)$
M2M	$O(Mp^2)$
U-list	$O(27Nq)$
V-list	$O(Mp^{3/2}\log p + 189Mp^{3/2})$
X-list	very small for uniform distribution $O(Nq)$ for otherwirse
W-list	very small for uniform distribution $O(Nq)$ for otherwirse
L2L	$O(Mp^2)$
L2P	$O(Np)$

```
if (l2l_cl.where & STARPU_CPU) {
  l2l_cl.type = STARPU_FORKJOIN;
  l2l_cl.max_parallelism = INT_MAX;
}

starpu_task *task_l2l = starpu_task_create();
task_l2l->cl = &l2l_cl;
task_l2l->cl_arg = static_cast<void*>(&_thisptr);
task_l2l->cl_arg_size = sizeof(_thisptr);

task_l2l->handles[0] = trgDwnChk_handle;
task_l2l->handles[1] = trgDwnEquDen_handle;

task_l2l->callback_func = NULL;
task_l2l->synchronous = 0;
task_l2l->tag_id = TAG_L2L;
task_l2l->use_tag = 1;

starpu_tag_t deps[] = { TAG_WLIST, TAG_XLIST };
starpu_tag_declare_deps_array(TAG_L2L, 2, deps);

STARPU_TASK_SUBMIT(task_l2l);
```

StarPU の機能上の特徴としては, 事前にパフォーマンスモデルを構築しておくことにより, 入力データの量から実行時間を推定しスケジューリングを行う点，MSI（Modified, shared, invalid）プロトコルによってプロセッサ間のデータ移動を管理する点などが上げられる．その他実装に用いた詳細な機能については 4.2 項で述べる.

4. 実装

本節では，まず逐次の CPU コードであるオリジナルの kifmm3d の各ステップを CUDA および OpenMP を用いて加速する実装について述べ，次にそれらを部品として構築されている StarPU 実装について述べる．最後に，性能評価の比較対象として実装した静的スケジューリングの最適実装について述べる.

4.1 各フェーズの CUDA および OpenMP による加速

オリジナルの kifmm3d の各ステップの加速については, 我々の既存発表 [9] を元にして, さらに最適化を行っている.

OpenMP 実装においては，プログラム中に存在する `for` ループを，`#pragma omp parallel` ディレクティブを用いて並列化した．一部スレッドセーフで無い部分について対策をしたほかは，非常にシンプルな並列化を行っている.

次に CUDA 実装であるが，すべてのステップについ

ⓒ 2012 Information Processing Society of Japan

て CUDA 化をおこなった．変更点としては，まず木構築フェーズに対する変更がある．オリジナルの kifmm3d は，CPU のみを対象として記述されているため，データ構造が GPU に適していない点が多々ある．例えば，木構造のノードごとに個別に C++のオブジェクトが割り当てられ，データが管理されている．そこで，木構築フェーズにおいて，元の CPU 向けデータ構造に対応する GPU 用のデータ構造も同時に生成するように変更した．これにより木構築の時間はわずかに長くなるが，計測した結果高々数百 ms であり，全体に対する影響は 1%以下であり軽微であった. また，CPU 用のコードで FFTW を用いている部分があるので，これは CUFFT で対応した．なお，全ステップについて CUDA 化を行った点，CUDA 実装において OpenMP を併用していない点が，我々の既存発表 [9] との差異である．

4.2 StarPU 実装

以上のような各ステップの OpenMP 実装，CUDA 実装を部品として，StarPU 実装を行った.

まず，おおまかな方針として，各計算ステップを 1 つの StarPU タスクとして定義した．codelet として，OpenMP 実装，CUDA 実装をそれぞれ CPU 用，CUDA 用の関数として設定する．タスク間の依存関係としては，図 1 の通りに設定した．

StarPU は，CPU 用の関数が基本的に逐次実行であることを前提とするが，ここでは我々の OpenMP 実装を利用するために `parallel tasks`（もしくは `combined workers` とも呼ばれる）という機能を用いた．これは，複数のタスクを合体させて並列に実行するとみなし，タスク内部で OpenMP 等の並列化を可能にする機能である．ただし，`parallel task` を可能にする `pheft` スケジューラは, `heft` スケジューラと比べるとサポートが弱く，スケジューリングがうまくいかない場合も散見される．これについては，OpenMP 実装をデータ並列なタスクとして分割することによって対応可能であるが，将来の課題とした.

U-list のような高度にデータ並列な処理については，`partitioning` という機能が利用可能である．これは，入力および出力となるデータ構造を分割して別のタスクとして並列に処理可能である場合，StarPU がデータ分割を半自動的に行ってくれる機能である．これにより，プログラマが自らデータの分割を計算する必要がなくなると同時に，あるタスクでは分割したデータを用い，別のタスクでは再び合体させたデータを用いる等の処理も可能となる．また，分割されたタスクは別々のタスクとしてスケジューリングされるので，1 つの計算を CPU と GPU で分担するようなことも可能である．本稿ではデータ構造が単純で分割しやすい点と複数 GPU による効率的な加速が期待される点から，U-list について `partitioning` を採用した．分割数はプログラマが指定する必要がある. 今回は，いくつか予備的な評価を行った結果，分割数 8 を用いた.

4.3 比較のための最適実装

動的タスクスケジューリングを用いた実装を評価するに当たって，性能上の目標としてタスク割り当てを静的に決定した実装を用意した．これを最適実装と考え，評価において比較対象とした.

前述のように，FMM の各ステップが CPU および GPU で実行可能であるので，取り得る組み合わせは多数となる．ここでは実際にそれらの組み合わせを総当たり方式で計算し，最も高速であったものを最適実装と定義することにする．具体的には，8 個の計算ステップについて CPU または GPU を選択し，さらにパラメーター q についても可能な 2 通りの中から選択することにより，2^9 通りの組み合わせ存在する．これらを全て評価し，最も実行時間が短かったものを最適とする．なお，パラメーター q が 2 通りという点については評価の項で述べる.

実装は以下のように行った. まず，プログラムの引数として，各ステップをそれぞれ CPU で実行するか GPU で実行するかの指示を与える．プログラムの開始時に，`pthread` を用いて CPU 用のスレッド，GPU 用のスレッドを立ち上げる．各スレッドは，次に自分が実行すべきステップの条件が整ったら（依存するステップが終了したら)，ただちに実行を開始する．各ステップが終了したかどうかの情報は条件変数を用いて管理され，条件が満たされていない場合は待機する.

また，可能な場合は複数 GPU を用いる．評価環境である TSUBAME2.0 では 1 ノードあたり 3 台の GPU が利用可能である. ただし，ステップによっては処理量とデータコピーのコストの都合上，複数 GPU が必ずしも高速であるとは限らない. 今回は，試行回数を減らすため事前に評価を行い，U-list のみが 3GPU を用い，それ以外は 1GPU を用いることとした.

最後に，この最適実装の制限について述べる. まず，CPU 用のスレッドと GPU 用のスレッドを立ち上げるという実装の都合上，複数のステップが GPU を同時に使って実行されることは無い.U-list は 3GPU を使って実行されるが，それ以外のタスクについては 1GPU のみを使うため，理論上は複数の GPU を用いて同時に処理を行うことで高速化が図れる可能性もあるが，今回は考慮しないこととした.

データのコピーについては，GPU での実行の際に必要な入力データと結果の出力データを全て転送することとした.例えばプログラムを通して read only であるとわかっているデータについてはあらかじめコピーしておく等の最適化も考えられるが，GPU のメモリが十分であるかどうかは自明では無いので，今回は実装しなかった．また，GPU での計算後に，出力データをそのまま次の GPU 計算で使うようなケースも考えられ，この場合はデータ転送を省くこ

ⓒ 2012 Information Processing Society of Japan

表 2 評価環境：TSUBAME2.0 単ノード

	TSUBAME2.0
CPU	Intel Xeon 5678（2.93Ghz,6 cores）× 2
GPU	NVIDIA Tesla M2050 × 3
Memory	54GB （host）, 3GB （GPU）
Softwares	Intel C Compiler 11.1, CUDA 4.1, StarPU 1.0

とも可能であり，これは実装可能であるが，かなり煩雑となるため将来の課題とした．いずれにしても，`cudaMalloc` にかかる時間は全体の数%以下であるので，改善の余地はあるものの大きな影響はないと考える．

5. 評価

5.1 パラメーター q について

まず，FMM を実行する上で重要となるパラメーター q について述べる. 多くの場合，FMM においては利用者が適切な q を選んで与えることが多い. 本研究では，q をどのように選択するかという問題については扱わないものとし，複数の q の候補について全てプログラムを実行して最適なものを選択することとした.StarPU と最適実装の両方についてこのようにしている．

q の候補としては，以下の理由で総当たりの候補を決定した. まず，扱う粒子数を 100 万とする. 一様分布を仮定したとき，実際に q の変化に従って木の高さが変化する点は $q = 31(\approx 10^6/8^5), 245(\approx 10^6/8^4), 1954(\approx 10^6/8^3), 15625(\approx 10^6/8^2)$ である. 今回，$q = 15625$ は，CPU と GPU の両方において U-list 計算が数分〜数十分以上となるので除き，さらに CPU で $q = 1954$ も同様に除いた．よって，31（CPU と GPU），245 （CPU と GPU），1954（GPU のみ）を q を候補とした. 今回の評価では非均一分散についても同様としたが，しかし，不均一分散においてどのような q を選択すべきかというのは自明では無く今後の課題である．

5.2 評価環境

評価には，TSUBAME2.0 の単ノードを使用した. 詳細を**表 5.2** に示す．また，実験データとして，100 万粒子の均一分散および球表面分散を使用し，kifmm3d のカーネルは最も基本的である Single Layer Laplace カーネルを用いた．

5.3 StarPU パフォーマンスモデルの構築

評価に先だって，StarPU のパフォーマンスモデルを構築した. これは，StarPU を `calibration` モードと呼ばれるモードに設定して実行を行うことで各 codelet の性能データが蓄積され，性能モデル式にフィッティングされてパフォーマンスモデルが構築される．

ここでは，StarPU を用いつつ，恣意的に CPU codelet のみ，CUDA codelet のみを用いるようにプログラムを調整し実行した. パフォーマンスモデルが構築されるまでに何回の実行が必要であるかは自明でないが，生成過程のパフォーマンスモデルの変化を観察した結果，各 200 回程度で一定の値に落ち着き，十分であると判断した．

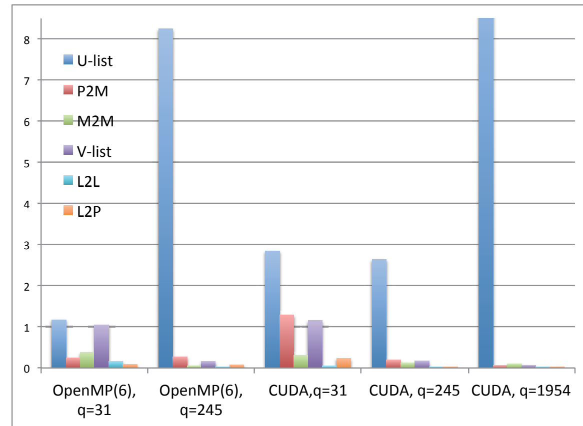

図 2 フェーズごとの OpenMP/CUDA による実行時間
粒子数 100 万の一様分布を用いて，q を変化させた時の CUDA/OpenMP 実装によるステップごとの実行時間

5.4 評価結果

まず，フェーズごとの OpenMP，CUDA の性能評価を**図 2** に示す.q ごとにそれぞれのステップの計算時間は大きく異なることが確認できる. おおむね，CPU の場合は $q = 31$，CUDA の場合は $q = 245$ が最適であることもわかる. また，本実装における U-list の CUDA 実装の最適化が不足していることも示唆している. これは今後の課題であるが，単純なデータ構造である N 体問題の直接計算と比較してデータ構造が複雑になり，レジスタが足りなくなってローカルメモリが使われてしまっていることが主要因だと考えている．

次に，最適実装を評価した結果について述べる. 粒子数 100 万について 2 つの粒子配置（均一分散，球表面分散）について総当たり方式により最適実装を求めた結果，**表 5.4** のようになった．

表 3 総当たりによる最適実装の探索結果

	均一分散	球表面分散
q	245	245
U-list	GPU	CPU
P2M	CPU	CPU
M2M	CPU	CPU
V-list	CPU	CPU
W-list	CPU	GPU
X-list	CPU	GPU
L2L	CPU	CPU
L2P	CPU	CPU
実行時間	2.29 [s]	1.37 [s]

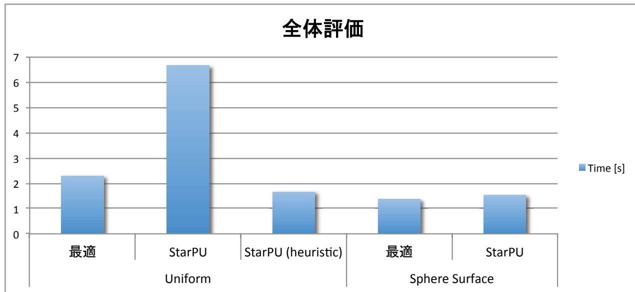

図 3 全体の性能評価
粒子数 100 万の一様分布について最適実装，StarPU 実装，StarPU（ヒューリスティクス有り）の 3 種類の実装，球表面分布について最適実装と StarPU 実装を用いて評価した結果

最後に，StarPU を用いた性能評価を述べる. ここで，単純に StarPU にスケジューリングを任せる場合に加えて，均一分布において U-list を CUDA codelet のみを用いて計算するという簡単なヒューリスティクスを導入したパターンも調査した. これは，U-list が直接計算であるということをふまえて，このステップにいては CUDA のみと指定した方が高速であろうという判断が容易に可能だと考えたためである. 評価結果を**図 3** に示す.

実行時間としては，一様分布について最適実行に対して StarPU（ヒューリスティクス無）が 34%，StarPU（ヒューリスティクス有）が 137%の性能となり，球表面分布については最適実行に対して StarPU が 89%の性能となった. 均一分散におけるヒューリスティクス無しの場合，U-list のタスクの過剰な割合が CPU で実行されてしまい大幅に実行時間が長くなってしまう現象が見られた. これは，StarPU のスケジューラーが，全体の実行時間を計算せず，プロセッサが空いていれば無条件に処理を始めるという判断をおこなってしまうことにあると考えられる. ヒューリスティクス有りではその問題が抑制されている. ただし，最適実行を大幅に超える性能が出た理由については，現在解析中である. また，StarPU のこのような挙動については，スケジューラーをさらに調整することで回避可能である可能性もあり，現在検討中である.

以上，今回は簡単なヒューリスティクスを導入したものの，最適実装も粒子分布ごとに異なるものが選ばれているという点も考慮に入れれば，最適実装に対して十分に競争的な性能を発揮できていると考えられる.

6. 関連研究

6.1 FMM

まず，FMM 全般について近年の関連研究を述べる. 本研究で対象としている kifmm3d を用いた研究としてはまず Lashuk らによる研究 [10] が挙げられる. 本研究の一部と同様，kifmm3d を CUDA を用いて高速化したものである. また，CPU のみでの高速化については Chandramowlishwaran らによりマルチコア CPU 向けに高度な最適化が行われている [11][12]. また，Rahimian らは，これらの実装をベースとして Jaguar スーパーコンピューター上で大規模なシミュレーションを行った [4].

FMM の実装としては，横田らによる ExaFMM[7] が挙げられ，これは CPU および GPU 向けに高度に最適化された FMM 処理系である.

6.2 FMM における動的タスクスケジューリング

FMM を, 動的タスクスケジューリングエンジンによって実行する試みは, 近年多く行われている. Lataief ら [13] は, 数値計算ルーチン向けに設計された CPU 用のスケジューリングライブラリ Quark[14] を ExaFMM に適用した. 田浦らは，軽量スレッドライブラリである MassiveThreads[15] を用いて ExaFMM をマルチコア CPU 向けに実装した [16].

また，Agullo らは，別の FMM 実装 black-box fmm に StarPU を適用した [17]. 本研究と同様に CPU と 3GPU よる実行を行い，3800 万粒子という大規模な実行を行っている.

7. まとめと今後の課題

本研究では，N 体問題の近似計算アルゴリズムである FMM を動的タスクスケジューリングエンジンである StarPU 上に実装し，静的にタスクを割り当てる最適実装と性能を比較した. 一部ヒューリスティクスを導入したものの，性能は十分に競争的であり，また複数の異なる粒子分散について性能可搬性を実現できることがわかった.

また，個々の計算カーネルで最適化が不十分である部分があるので，今後も継続的に改善したい. 同時に，NVIDIA の Tesla GPU 以外のプラットフォームでも適切なロードバランシングと性能可搬性が実現できることを目標に，OpenCL による実装も検討する.

StarPU 以外の動的タスクスケジューリングエンジンを用いて性能を評価したいと考えている. 動的タスクスケジューリングエンジンは，それぞれ実装の方針やプログラミングモデルが異なっている. それぞれが性能と生産性にどのような影響を与えるのかどうかを比較したい.

参考文献

[1] L. Greengard *et al.*, “A fast algorithm for particle simulations,” *J. Comput. Phys.*, vol. 73, pp. 325–348, December 1987.
[2] 福田 圭祐, 丸山 直也, and 松岡 聡, “Cpu/gpu を共用したヘテロジニアス環境における fmm の最適化,” vol. 2011-HPC-132, 2011.
[3] R. Yokota *et al.*, “Petascale turbulence simulation using a highly parallel fast multipole method,” *CoRR*, vol. abs/1106.5273, 2011.
[4] A. Rahimian *et al.*, “Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures,” ser. SC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

ⓒ 2012 Information Processing Society of Japan

[5] L. Ying *et al.*, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591 – 626, 2004.

[6] "Parallel kernel-independent fast multipole 3d code," November 2012, `http://www.mrl.nyu.edu/ harper/kifmm3d/documentation/`.

[7] R. Yokota *et al.*, "Fast multipole methods on a cluster of gpus for the meshless simulation of turbulence," *Computer Physics Communications*, vol. 180, no. 11, pp. 2066 – 2078, 2009.

[8] C. Augonnet *et al.*, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[9] 福田 圭祐, 丸山 直也, and 松岡 聡, "Cpu/gpu を共用したヘテロジニアス環境における fmm の最適化," vol. 2011-HPC-129, 2011.

[10] I. Lashuk *et al.*, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 58:1–58:12.

[11] A. Chandramowlishwaran *et al.*, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1 –12.

[12] A. Chandramowlishwarany *et al.*, "Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method," ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12.

[13] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *CoRR*, vol. abs/1203.0889, 2012.

[14] U. of Tennessee, *QUARK Users Guide*, April 2011. [Online]. Available: http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf

[15] 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*）**, vol. 4, no. 1, pp. 13–26, mar 2011.

[16] 田浦 健次郎, 中島 潤, and 横田 理央, "Exafmm のタスク並列処理系 massivethreads による並列化とその評価," vol. 2012-HPC-135, 2012.

[17] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Pipelining the fast multipole method over a runtime system," *CoRR*, vol. abs/1206.0115, 2012.