Vol.2016-HPC-153 No.20
2016/3/2

軽量スレッドライブラリ Argobots を用いた OpenMP 実装の性能分析と改善

李 珍泌[1,a)] 杉山 大輔[2,b)] 村井 均[1,c)] 佐藤 三久[1,2,d)]

概要：エクサスケール並列計算機の実現に向けてプロセッサのメニーコア化やメモリ構造の複雑化が進む中、スレッド並列プログラミングモデルにおいて粗粒度タスクの動的生成によるタスク並列化や、スレッド affinity の調整による NUMA アーキテクチャの最適化などが注目を集めている。本研究はメニーコアプロセッサの並列プログラミングモデルとして OpenMP を考え、軽量スレッドライブラリ Argobots を用いたランタイムの改良を行った。Argobots はユーザーレベルで実装されたスレッドを用いることで、OS スレッドに比べて少ないオーバーヘッドでスレッドのコンテキストスイッチやマイグレーションを行うことができる。Omni OpenMP コンパイラの Pthreads による実装を Argobots に変更したことで、NPB-MG や姫野ベンチマークのような従来のループ文によるデータ並列アプリケーションの性能が低下しないことを確認するとともに、Pthreads 実装では効率よく処理することができなかった入れ子の parallel region の性能が改善することを確認した。

1. はじめに

モバイル市場の成長にともない、近年のプロセッサ設計のトレンドは省電力化である。省電力化の主な手法としては動作周波数の低下と計算コアの単純化があげられる。それによって失われる性能を補うためにプロセッサ内のコア数を増やすメニーコア化が進んでいる。High Performance Computing（HPC）市場においても Graphics Processing Unit（GPU）や Intel Xeon Phi のようなメニーコアアクセラレータが登場し、シェアを伸ばしている。汎用プロセッサにおいても同様の傾向が見られ、今後、より単純化された計算コアがプロセッサ内に多数搭載されるメニーコア化のトレンドが続くと予想される。

メニーコアプロセッサで複数の計算コアを用いて計算を行うためにはスレッド並列プログラミングモデルによるアプリケーションの並列化を行う。HPC の分野で広く使われている OpenMP は parallel や for 指示文によるループ文のワークシェアリングを記述する。ワークロードが不規則に発生するアプリケーションを並列化する事例が増えてきたことや、メニーコア化によるコア数の増加によってタスク並列化に対応できるハードウェアが揃ってきたため、OpenMP 3.0 からは task 指示文による動的タスク生成と同期をサポートするようになった。

HPC の分野に限らず、メモリバンド幅がアプリケーション性能に支配的な影響をおよぼす例は多い。メニーコア化が進むにつれてメモリを共有する計算コアの数が増加し、一つの計算コアあたりのメモリバンド幅が相対的に減少する。このような問題を解決するためにプロセッサの中で複数の階層を持つキャッシュメモリを搭載することや、Non-Uniform Memory Access (NUMA) アーキテクチャのようなメモリ参照のコストが不均一なメモリ構造を採用する計算機システムが増えている。

従来の OpenMP の仕様はメモリ最適化に関する機能を持っておらず、コンパイラの独自機能として実装されていた（Intel OpenMP の KMP_AFFINITY など）。不均一なメモリ構造が一般的になり、データ参照の局所化が性能最適化のキーポイントの一つになったため、OpenMP 4.0 からスレッド affinity を指定する環境変数や指示文説が導入された。NUMA アーキテクチャの性能を引き出すためにはこのようなプログラミングインターフェイスを利用するか、ランタイムによるサポートが必要である [1][2]。

現在、HPC の分野ではペタフロップス性能の計算機システムが使われており、次はエクサフロップスのシステムを構築することを目標としている。このような計算機シス

[1] 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science
[2] 筑波大学
University of Tsukuba
[a)] jinpil.lee@riken.jp
[b)] sugiyama@hpcs.cs.tsukuba.ac.jp
[c)] h-murai@riken.jp
[d)] msato@riken.jp

ⓒ 2016 Information Processing Society of Japan

テムを構築するためには現在のプロセッサのトレンドを追及したメニーコア化、メモリ階層の複雑化がさらに進むと考えられる。

本研究の目的はエクサスケールコンピューティングのスレッド並列プログラミングモデルとして OpenMP を考え、メニーコアプロセッサをターゲットにした動的タスク生成によるタスク並列化や、スレッド affinity の調整によるメモリ最適化を行う OpenMP コンパイラを開発することである。本稿ではその準備段階として米国 Argonne National Laboratory（ANL）で開発が行われている軽量スレッドライブラリ Argobots[3] を用いて著者らが開発を行っている Omni OpenMP コンパイラ [4]（以後、Omni コンパイラ）のランタイムの実装を行い、性能評価の報告を行う。著者らは研究報告 [5] で Argobots による Omni コンパイラの初期実装に関する報告を行った。本研究はその続きであり、実装方式の改善や性能分析を行ったものである。

メモリ階層を意識したスレッド並列化を OpenMP の指示文で記述する場合、各階層に対して parallel region を用いる入れ子構造になると考えられる。動的なタスクの生成は処理の再帰処理によって記述される場合が多く、そのような場合でも並列タスクチームが入れ子構造で生成される。本研究では既存のワークシェアリングの手法で並列化されたベンチマークの評価を行うことでスレッドライブラリの変更によるオーバーヘッドを調べるとともに、入れ子並列化のオーバーヘッドについても実験的な評価を行った。

本稿の構成は以下のようである。第 2 章において軽量スレッドライブラリ Argobots の概要について述べる。第 3 章において POSIX Threads ライブラリ（以後、Pthreads）による既存の Omni コンパイラの実装や性能改善の手法について述べ、Argobots を用いた OpenMP ランタイムの実装について説明する。第 4 章においてはマイクロベンチマークなどを用いたランタイムの性能評価を行い、GNU コンパイラコレクションの C 言語版 OpenMP 実装（以後、GNU コンパイラ）や従来の Omni コンパイラ実装と比較を行う。第 5 章では本研究で得られた結果について考察を行い、今後の課題について述べる。

2. 軽量スレッドライブラリ Argobots

この章では軽量スレッドライブラリ Argobots の概要について述べ、従来の Pthreads のようなスレッドライブラリと比較して Argobots を用いた場合にどのような改善が見込めるのかについて説明を行う。

Argobots は ANL のエクサスケールコンピューティングに向けた研究プロジェクト Argo[6] の一部として開発が進められているユーザレベルスレッドライブラリである。図 1 に Argobots の実行モデルを示す。4 個の計算コアを持つプロセッサでスレッドを生成するときの実行モデルをあらわす。Argobots のスレッドは Working Unit（WU）と呼ばれ、User Level Thread (ULT) と Tasklet の 2 種類がある。ULT は OS スレッド相当の機能をユーザーレベルで実装したもので、条件変数や mutex などの同期プリミティブが提供される。Tasklet は実行コストが ULT より軽量な WU であるが、ブロッキング関数の呼び出しや他の WU への yield ができないなどの制約がある。

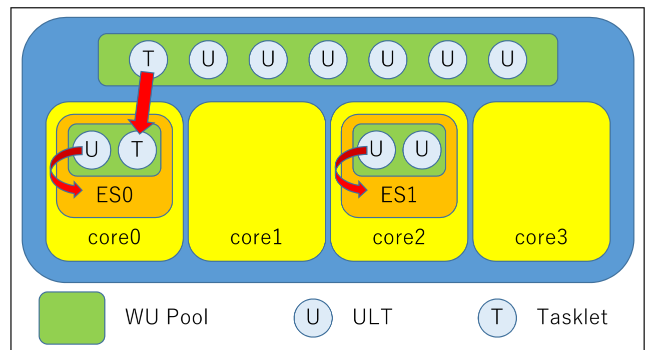

図 1　Argobots の実行モデル

WU は計算コアで直接実行されるのではなく、Execution Stream（ES）の実行プールにスケジュールされる。ES は OS スレッドを抽象化したもので、実行プールから ULT や Tasklet を取り出して実行する。Argobots の実行モデルは一つ計算コア、またはハードウェアスレッドに一つの ES を割り当てることを仮定している。ES の内部の WU は逐次に実行されるが、各 ES が計算コア間で並列に動作することで並列実行を実現する。

各 ES は専用のプールを持つが、複数の ES でプールを共有することも可能であるため、それを利用した work stealing を実装することができる。プールから WU を取り出すスケジューリングのアルゴリズムは Argobots API を用いてユーザが明示的に実装する（デフォルトでは FIFO)。Argobots のスケジューラは ULT の一種であり、内部で書かれたプール操作のアルゴリズムによって ES が次に実行すべき WU を選択する。

Argobots のスレッド生成はユーザーレベルで実現された ULT や Tasklet を ES プールにスケジュールすることで行う。そのため、OS スレッドを生成する Pthreads に比べてスレッドの生成やコンテキストスイッチのオーバーヘッドを抑えることができる。Argobots を用いることで従来の OS スレッドを用いる場合はオーバーヘッドが大きい粗粒度タスクの動的生成や入れ子 parallel region の処理コストを軽減させることができると考えられる [7][8][9]。

3. OpenMP ランタイムの実装

Omni コンパイラは筑波大学 HPCS 研究室および理化学研究所計算科学研究機構で開発が進められている Omni コンパイラプロジェクトの一部である。本章では Pthreads を用いた既存のランタイム（以後、Omni-Pthreads）の性能改善のためにどのように実装を変更したかを説明する。次に

ⓒ 2016 Information Processing Society of Japan

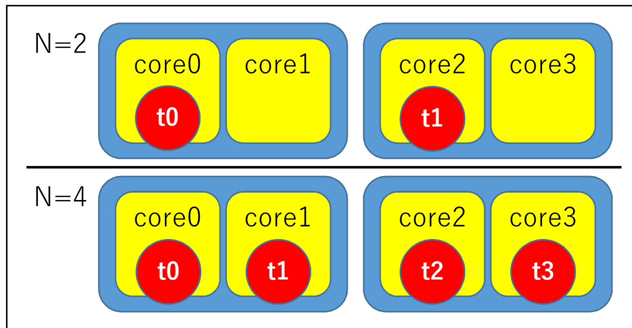

図 2 spread オプションによるスレッドの割り当て

Argobots によるランタイムの実装 (以後、Omni-Argobots) について述べる。

3.1 既存の実装の改善

既存の Omni-Pthreads の実装の概要は同著者らによる論文 [5] で説明されているのでここでは省略する。ここでは Omni-Pthreads が入れ子の parallel region をどう処理しているのかについて説明する。本稿では用語の混用を避けるために Pthreads のスレッドや Argobots の ES のような OS スレッドをプロセスと呼び、OpenMP 仕様上のスレッドをスレッドと呼ぶことにする。

Omni-Pthreads は環境変数 OMPC_NUM_PROCS で指定された数のプロセスをプログラムの実行開始時に生成する。Omni コンパイラは OpenMP 2.0 の仕様に対応しているため、OpenMP 4.0 で定義されているプロセス affinity オプションが実装されておらず、NUMA アーキテクチャに向けた最適化を行うことができない。本研究で評価を行うにあたって GNU コンパイラと比較を行うために OpenMP 4.0 の環境変数 OMP_BIND_PROC の spread オプションと同等の機能を実装した（実装には hwloc[10] を利用)。

図 2 に OMP_BIND_PROC に spread を指定したときのプロセスの割り当て方を示す（2 ソケット、2 コアの構成)。spread オプションが指定された場合、プロセスはなるべく間隔を開けて計算コアに割り当てられる。二つのプロセスが生成される場合は一つ間隔を開けて割り当てるので、その結果として異なるソケットに割り当てられる。四つのプロセスが生成される場合はプロセスとコアの数が同じなので各コアに連続して割り当てられる。spread オプションを使うことで少ないプロセスを用いる場合でも NUMA ノードのようなメモリ資源を分散して使うことが期待される。

Omni-Pthreads はプロセスプールの動的な拡張を行わないため、プログラムの中でプロセスより多いスレッドが生成することはできない。スレッドの割り当ては mutex 同期で制御される共有カウンターを使った round-robin 方式で行われる。このような手法を用いる場合、入れ子の parallel region の内側のスレッドチームでカウンターの競合が生じる。その結果、各スレッドは早いもの勝ちでプロセスを取得するため、どのプロセスにスレッドが割り当てられるかは実行時のタイミングに依存する。このような実装ではスレッドがどのプロセスを利用するか明示的に制御することができない。

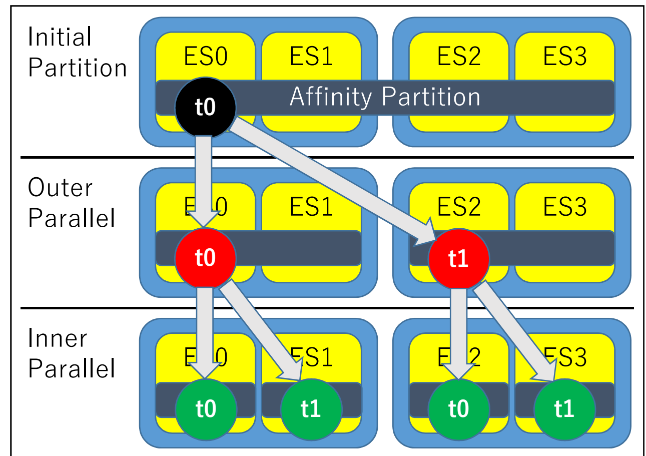

図 3 Omni-Argobots のスレッド割り当て

Omni-Pthreads のもう一つの問題点はプロセスプールサイズが固定であるため、プログラム実行時に必要なプロセスの最大数をプログラム実行前に予測しなければならないことである。そのような予測は一般的に不可能であるため、本稿の評価では環境変数の値としてプログラム実行前に指定している。

3.2 Argobots による実装

Omni-Argobots の初期実装の概要やそれに用いた Argobots の API は研究報告 [5] で報告済みである。ここでは初期実装からの変更とそれによってどのようなところが改善されたかについて述べる。

Omni-Argobots でも Omni-Pthreads と同様のプロセス affinity オプションを実装する。Omni-Argobots はプロセスとして ES を生成するが、なるべく離れた計算コアに ES を割り当てるように実装を変更した（Omni-Pthreads と同様、hwloc を利用)。これによって、ES の数と同じか少ない WU を用いるときに OMP_PROC_BIND の spread オプションと同じ効果がえられる。

Omni-Pthreads はプロセスより多い数のスレッドを実行できない問題があった。Omni-Argobots はプロセス（ES）より多いスレッド（WU）が生成されてもスレッドをプロセスのプールに格納することができるため、Omni-Pthreads のような問題は発生しない。Omni-Argobots は最大でも計算コアの数と同じだけの ES を生成し、それより多いスレッドが生成される場合は各 ES のプールに格納されて逐次に実行される。スレッドの数がプロセスの数より少ない場合は計算コアへの ES 割り当てと同様、spread オプションの割り当てを行った。

Omni-Pthreads では入れ子 parallel region の内側のス

ⓒ 2016 Information Processing Society of Japan

レッドチームがタイミングに依存してランダムにプロセスに割り当てられるという問題があった。Omni-Argobots は parallel region が生成される毎にスレッド割り当ての範囲（以後、パーティション）を区切ることでこの問題を改善している。図 3 に外側と内側が各々 2 スレッドで実行される parallel region のスレッド割り当てを示す。実行開始時のスレッド割り当てパーティションはすべての ES を含む。外側の parallel region で 2 スレッドによる並列実行が行われるが、spread オプションによるスレッド割り当てが行われるため、初期パーティションから ES0 と ES2 が選ばれる。割り当てパーティションは等間隔に区切られ、現在のスレッドチームのスレッドは初期パーティションの半分のサイズのパーティションを持つ。次に内側の parallel region が生成されるときはこのパーティションを用いたスレッドの割り当てが行われるため、外側の parallel region の 0 番スレッドから生成されるスレッドチームは ES0 と ES1 に、1 番スレッドから生成されるスレッドチームは ES2 と ES3 に割り当てられる。

図 3 のようなスレッドの割り当ては GNU コンパイラでも行われている。GNU コンパイラは Pthreads のスレッドに対してこのような割り当てを行うが、Omni-Argobots は ES の実行プールにスレッドをスケジュールする。Argobots のようなユーザーレベルスレッドは実行する ES をわずかなコストで切り替えられるので、このような特性を利用して実行時にスレッド割り当て方を動的に変更する最適化が考えられる。本研究ではスレッド間でワークロードのバランスが取れた入れ子の parallel region を評価しているため、均等にパーティションを区切る割り当て方を行っている。今後、不規則なメモリ参照に対する動的なスレッド割り当てについて検討を行う予定である。

4. 性能評価

本章では各種ベンチマークを用いた性能評価を行う。GNU コンパイラ、Omni-Pthreads、Omni-Argobots を用いて評価を行い、性能を比較する。

4.1 ベンチマークおよび評価環境

本研究では以下のベンチマークを用いて各実装の性能を評価する。本来、並列プログラムの性能はランタイムの実行性能だけでなく、コンパイラによる並列コードの生成方式にも影響される。しかし、今回評価を行ったベンチマークは OpenMP の parallel や for 指示文によるデータ並列化であるため、並列コードの生成方法は類似しており、ランタイムによる影響が大きい。よって、今回の評価は GNU コンパイラ、Omni-Pthreads、Omni-Argobots のランタイムの比較と考えることができる。

- EPCC OpenMP Micro-Benchmark Suite（Syncbench）
- 入れ子の parallel region を評価するマイクロベンチマーク
- Nas Parallel Benchmarks MG（NPB-MG）
- 姫野ベンチマーク

表 4.1 に評価に用いたハードウェアおよびソフトウェアの一覧を示す。評価環境のプロセッサには HyperThread が有効化されており、論理コアとしては 72 コア存在するが、評価には物理コアと同じ数のスレッドを用いる。

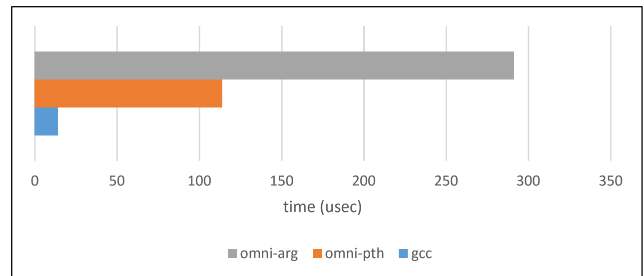

図 4 parallel 指示文の性能評価

4.2 Syncbench の評価

EPCC OpenMP Micro-Benchmark Suite[11] は OpenMP の指示文やユーザレベル API 関数などの構成要素の性能を測定するマイクロベンチマークスイーツである。本研究では OpenMP 2.0 の仕様に相当する Syncbench の評価項目について評価を行った。計算コアと同じ数の 36 スレッドを用いて評価を行った結果を図 4 と図 5 に示す。GNU コンパイラは Omni コンパイラと異なる手法で実装されているため、Pthreads から Argobots へのスレッドライブラリの変更による性能変化を観測することはできないが、Omni コンパイラの今後の最適化のための参考データとして同時に評価を行った。

図 4 は parallel 指示文の評価結果をあらわす。parallel 指示文の結果でわかることは Omni コンパイラの parallel region のオーバーヘッドが GNU コンパイラよる大きいということである。GNU コンパイラ、Omni-Pthreads ともに parallel region の生成で OS スレッドを生成することはないため（GNU コンパイラは入れ子構造の parallel region ではない場合に限る）、性能差はスレッドチーム情報の生成コストの違いによるものと考えられる。Omni-Argobots は Omni-Pthreads と同様の処理でスレッドチームの情報を生成するが、それに ULT 生成のオーバーヘッドが上乗せされるため、より時間がかかっている。

表 1 評価環境

	名称
プロセッサ	Intel (R) Xeon (R) CPU E5-2699 v3 x2 ソケット (18 コア x2, 2.30 GHz)
メモリ	DDR4 128GB
コンパイラ	GNU Compiler gcc version 4.8.5
	Omni Compiler 0.9.2 (modified)

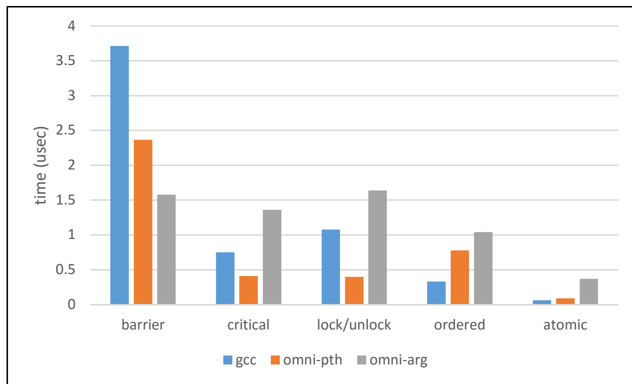

図 5 同期指示文/ランタイム関数の性能評価

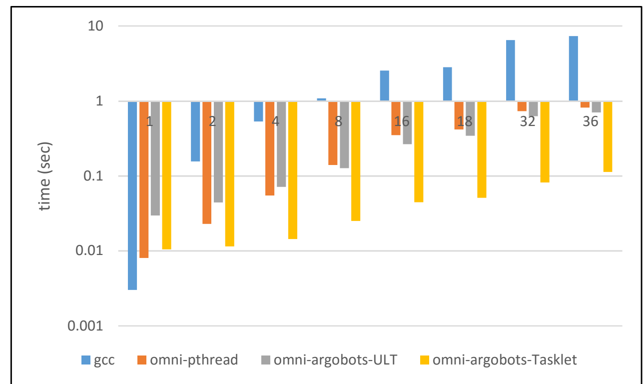

図 6 Nested Parallel Region の性能評価

図 5 は barrier、critical、lock/unlock、ordered、atomic のような OpenMP のスレッド間同期機構の性能評価の結果をあらわす。Omni-Pthreads と Omni-Argobots の同期機構は同じアルゴリズムによって実装されており、スレッドライブラリの API を切り替えるだけの変更を行っている。そのため、図 5 であらわれた性能差は用いたスレッドライブラリの実行コストによる影響と考えられる。Omni コンパイラのバリア同期は spin-lock による Linear アルゴリズムで実装されているが、spin-lock の中で使われる yield 関数のコストによって性能に影響が出ている。そのほかの同期機構に関しては mutex や条件変数のような同期プリミティブの実行コストが Pthreads と Argobots で異なるため、性能に差が出ている。Argobots の同期プリミティブは十分に最適化されておらず、単純に関数を置き換えるだけでは性能が悪化する。

OpenMP の同期機構の最適化のためには Argobots の同期プリミティブの性能改善を行うか、Omni コンパイラ側でアルゴリズムの改善によるランライムの最適化やコード生成方式の改善を行う必要がある。現在 Omni コンパイラのバリア同期は Linear アルゴリズムで実装されているため、コア数に比例して実行時間が増加する。今後、コア数が増えた場合に Tree アルゴリズムを用いた実装に切り替えるなどの最適化を行う予定である [12][13]。

4.3 Nested Parallel マイクロベンチマークの評価

以下に入れ子の parallel region の性能評価のためのマイクロベンチマークのソースコードを示す。float 型の配列に代入を行うもので、二重ループ文による繰り返しが行われる。二つのループ文を OpenMP の parallel for で並列化しているため、入れ子の形になっている。

```
1 #pragma omp parallel for num_threads(36)
2 for (j = 0; j < it; j++) {
3 #pragma omp parallel for
4     for (i = 0; i < it; i++) {
5         v[j * it + i] *= 0.9f;
6     }
```

ループ文の本体の計算が少ないことから、測定された時間は入れ子の形をした parallel region を実行する場合の OpenMP ランタイムのオーバーヘッドを示すことになる。外側の parallel region のスレッド数は 36 に固定されており、内側の parallel region のスレッド数を 1 から 36 に変更させながら評価を行った。入れ子 parallel region のマイクロベンチマークと後述する姫野ベンチマークのために実験的な実装として内側の parallel region のみを Tasklet によって実行する実験を行った。Tasklet は yield ができない、ブロッキング関数が呼べないなどの制約があるため、用途が限られるが、ULT に比べると生成や管理のコストが低い。OpenMP コンパイラで最内ループが Tasklet で実行できることを解析できたら、ULT と Tasklet を選択的に利用することができる。本研究ではその予備実験として内側のみを Tasklet で実行する評価を行った。

図 6 にマイクロベンチマークの評価結果を示す。GNU コンパイラはスレッド数が計算コアの数と同じか少ない場合はスレッドプールからの割り当てを行うため、もっとも高い性能を達成する。Omni-Pthreads も同等の操作を行うため、それにつぐ性能を示す。Omni-Argobots は ULT や Tasklet を parallel region の生成時に生成するため、そのコストがマイクロベンチマークの性能に現れている。内側のループを Tasklet で実行する場合、Tasklet 生成のコストが ULT と比較して低いことから、より高い性能を達成することがわかる。

スレッド数が増えるにつれてスレッドの総数が計算コアより多くなる。そのような場合、GNU コンパイラは parallel region の開始時に新しい OS スレッドを生成する実装になっており、ULT や Tasklet を生成するよりも高いオーバーヘッドが発生する。Omni-Pthreads はあらかじめ実行前に最大スレッド数を指定し、プログラム開始時に OS スレッドを生成する手法であるため、GNU コンパイラに比べるとオーバーヘッドが低い。しかし、実行前に最大スレッド数を予想することは困難であるため、このような手法を一般的に用いることはできない。また、OS スレッドを常時過剰に生成するため、コンテキストスイッチのオー

ⓒ 2016 Information Processing Society of Japan

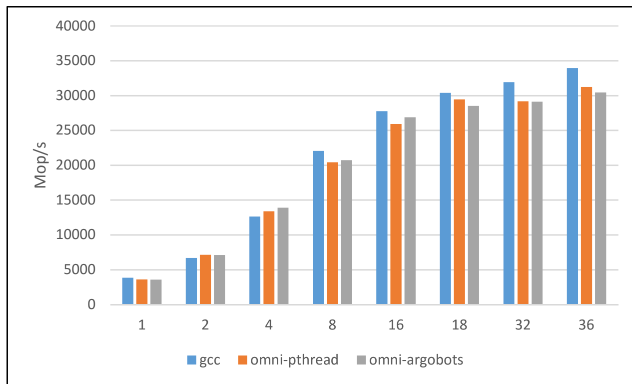

図 7 NPB-MG の性能評価

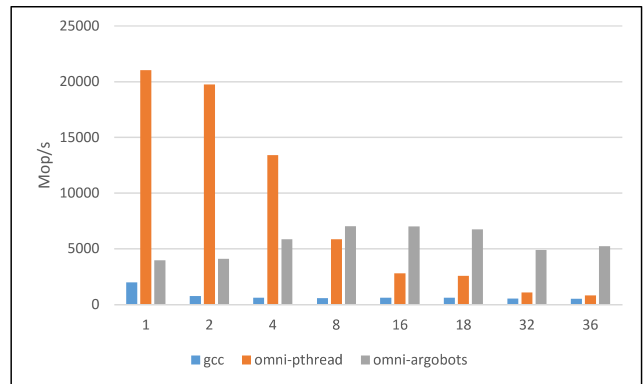

図 8 NPB-MG の性能評価（nested）

バーヘッドが発生し、Argobots を用いた場合に比べると性能が落ちることがわかる。

Omni-Argobots は割り当てパーティションを考慮した spread オプションを実装しているため、外側の parallel region でパーティションのサイズが均等に区切られ、内側のパーティションのサイズは 1 になる。そのため、内側のスレッドチームのワークロードは同じ ES で逐次に実行される。Omni-Argobots の評価結果は同じサイズのワークロードを複数の WU で分割したあとに、それを逐次に実行したオーバーヘッドを示すことになる。内側のスレッド数が 1 の場合に比べることによってワークロードが粗粒度になるにつれて ULT 実装の効率が Tasklet 実装より大きく低下することがわかる。

評価結果により、スレッド数の増加によって計算コアより多い OS スレッドを生成する実装よりユーザレベルで実装されたスレッドを用いた場合の性能がより高かったことを確認し、ULT と Tasklet の実行オーバーヘッドの比較を行うことができた。

4.4 NPB-MG の評価

NPB-MG は Nas Parallel Benchmarks[14] のカーネルベンチマークの一つであり、マルチグリッド法により 3 次元ポワソン方程式の解を求めるものである。元のソースコードは Fortran で書かれているが、評価には NPB 3.3 をベースに C 言語に移植された SNU NPB Suite[15] を用いる。データサイズは Class C である。

図 7 に OpenMP で並列化された NPB-MG の結果を示す。NPB-MG の OpenMP 版は三重ループ文の最外ループが parallel for 指示文によって並列化されている。GNU コンパイラがもっとも高い性能を達成するが Omni コンパイラで並列化されたバージョンも計算コア数に比例して性能が向上していることがわかる。Omni-Pthreads と Omni-Argobots は 32 コアまでで同等の性能を見せる。これは Syncbench で見せた parallel region 生成の性能差はあるものの、ワークロードが十分である場合に Argobots で実装されたランタイムが Pthreads と同等の性能を達成できることを意味する。36 コアのときには ordered 指示文などの同期機構の性能差により Omni-Argobots の性能効率が低下する。

GNU コンパイラと Omni コンパイラの性能差の原因としては Syncbench の結果で確認できたように parallel region の生成コストが高いことが考えられる。GNU コンパイラランタイムのベンチマークを行い、スレッドチームの実装の最適化を行う予定である。

図 8 に入れ子の parallel region で記述された NPB-MG の評価結果を示す。評価のために逐次版 NPB-MG Class C の実行時間の 70%を占める psinv() 関数と redid() 関数に入れ子構造の parallel region を導入した。三重ループ文の最外ループに加えてひとつ下の階層のループも parallel for 指示文によって並列化を行った。内側の parallel region のスレッド数を計算コアと同じ 36 に固定し、外側の parallel region のスレッド数を 1 から 36 まで増加させたときの性能を評価した。外側の並列度が足りないなどの理由で入れ子の parallel region を記述した場合を想定したものである。

GNU コンパイラの場合、マイクロベンチマークで示したように計算コア数より多いスレッドが生成された場合はスレッドプールに対する動的なプロセスの生成が行われるため、スレッド数が増えるにつれて性能か低下する。Omni-Pthreads はプログラムの開始時に最大個数のプロセスを生成するためにそのようなオーバーヘッドは発生しない。しかし、GNU コンパイラと同様にスレッドが増加するにつれてコンテキストスイッチのコストにより性能が低下することが確認できる。Omni-Argobots では ES は計算コア分だけ生成され、スレッドは各 ES のプールに生成される。コンテキストスイッチはプロセスより少ないオーバーヘッドで行われるため、GNU コンパイラや Omni-Pthreads に比べると性能の低下が抑えられている。しかし、ULT の生成コストがオーバーヘッドになり、図 7 のような性能をえることはできない。Tasklet のようなオーバーヘッドの少ない WU を用いることで性能を改善することも考えられるが、現在のコンパイラの実装では NPB-MG の内側のループでバリア同期などによるスレッドの yield が発生す

ⓒ 2016 Information Processing Society of Japan

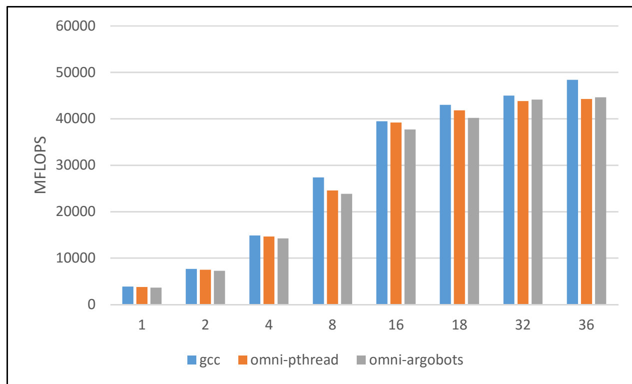

図 9 姫野ベンチマークの性能評価

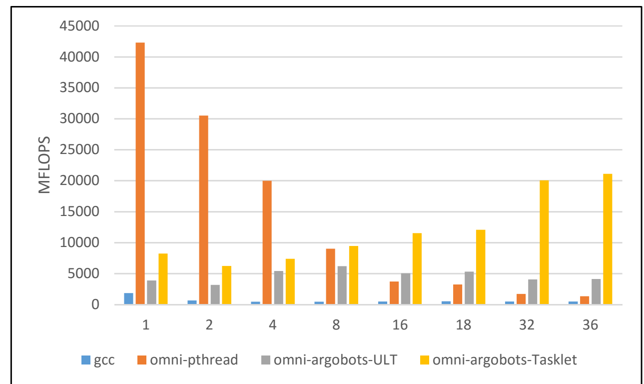

図 10 姫野ベンチマークの性能評価（nested）

るため、Tasklet を用いることはできない。

4.5 姫野ベンチマークの評価

姫野ベンチマーク [16] はヤコビ法による 3 次元ポワソン方程式の解を求めるものである。評価では LARGE データセットを用いる。

図 9 に姫野ベンチマークの評価結果を示す。GNU コンパイラがもっとも高い性能を示し、Omni コンパイラが生成した並列プログラムも計算コア数に比例した性能向上を見せる。すべての評価で同等のスレッド affinity オプションを実装しているため、メモリ性能が性能に支配的な影響をおよぼす姫野ベンチマークではスレッドの増加による性能変化の傾向が互いに類似している。NPB-MG に比べるとスレッド同期の影響は少ないため、Omni-Pthreads と Omni-Argobots でほぼ同等の性能を見せる。

図 10 に入れ子 parallel region で記述された姫野ベンチマークの評価結果を示す。NPB-MG と同様に最外ループだけでなく、ひとつ下の階層のループも parallel for 指示文で並列化されている。内側の parallel region で 36 スレッドを生成し、外側の parallel region のスレッド数を 1 から 36 に変更しながら評価を行った。GCC コンパイラ、Omni-Pthreads、Omni-Argobots（ここでは omni-argobots-ULT）は NPB-MG と同様な性能変化を見せる。特に Omni-Argobots の性能は NPB-MG とほぼ同じ数字を示し、ULT 生成や管理のオーバーヘッドを計測していることを示す。

姫野ベンチマークのリダクションは誤差の計算のために必要な処理であり、実行ステップ数には影響しない。リダクション演算を省略することによって内側のループを Tasklet で実行できるようになる。図 10 の omni-argobots-Tasklet は内側の parallel region を Tasklet で実行した結果をあらわす。その結果、ULT に比べて性能が向上し、図 9 で示した最大性能の半分程度の性能を達成した。この結果から入れ子の parallel region の実行において ULT と Tasklet の選択的な利用が Argobots を用いたランタイムの実装に重要であることがわかった。

4.6 評価結果の考察

ULT の生成コストはスレッドプールからアイドルなスレッドを割り当てる他のランタイムの実装にくらべるとオーバーヘッドが大きいが、NPB-MG や姫野ベンチマークなど実用上の大きさを持つワークロードのベンチマークでは目に見える性能の低下は観測できなかった。しかし、Argobots の条件変数や mutex などの同期機構が Pthreads に比べて高いオーバーヘッドを持つためにスレッドの同期が頻繁に発生するベンチマークでは Pthreads による実装より性能が低下することを確認した。今後、ランタイムの実装アルゴリズムの改善による同期プリミティブ利用頻度の削減などの最適化を行う予定である。

ULT はコンテキストスイッチのコストは低いものの、生成コストがボトルネックになるため、軽いワークロードの処理には向かない。しかし、Tasklet のような利用上の制約がないために基本的には ULT を用いた parallel region の生成を行う必要があり、性能改善のためには軽い処理をまとめるなどの最適化が必要である。Tasklet は実行コストが少ないため、大量の軽量ワークロードを作るのに向いている。しかし、ブロッキング関数を呼び出せないなど利用法に制約があるため、どの OpenMP 並列構造をタスクで実行するかは処理系による解析が必要である。Tasklet の利用範囲を広げるためにはトランザクショナルメモリを用いた投機的な実行によってブロッキングを回避するなどの方法も考えられる [17]。

5. 結論と今後の課題

エクサスケールコンピューティングに向けたスレッド並列プログラミングモデルには計算コアを余すことなく利用できるタスク並列化やデータ参照を局所化するスレッド割り当てが必要になる。本研究では軽量スレッドライブラリ Argobots を用いて Omni OpenMP コンパイラのランタイムを実装し、マイクロベンチマークや NPB-MG、姫野ベンチマークによる性能評価を行った。Argobots を用いることで柔軟なスレッド割り当てができるようになり、入れ子の parallel region で書かれたプログラムを Pthreads に比

ⓒ 2016 Information Processing Society of Japan

べて少ないオーバーヘッドで実行できることがわかった。また、従来の parallel や for 指示文で記述されたデータ並列化に対しても Pthreads と同等の性能を達成することを確認した。

今後の課題としては以下のようなものがあげられる。

- 与えられた parallel region が Tasklet によって実行できるかを検出し、コード生成を行う。
- メモリ参照が不規則なアプリケーションに対して動的なスレッド割り当てを実装し、評価を行う。
- OpenMP 3.0 の task 指示文を実装し、そのランタイムの試作と評価を行う。
- GNU コンパイラや Clang/LLVM の OpenMP 実装とランタイムインターフェイスで互換性を持たせることでコード生成に依存しないランタイムのみ比較を行う。

謝辞 本研究の一部は、理化学研究所計算科学研究機構と筑波大計算科学研究センターの共同研究「ポスト京の並列プログラミング環境およびネットワークに関する研究」によるものである。

参考文献

[1] Durand, M., Broquedis, F., Gautier, T. and Raffin, B.: *OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings*, chapter An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines, pp. 141–155 (online), DOI: 10.1007/978-3-642-40698-0_11, Springer Berlin Heidelberg (2013).

[2] Muddukrishna, A., Jonsson, P. A., Vlassov, V. and Brorsson, M.: *OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings*, chapter Locality-Aware Task Scheduling and Data Distribution on NUMA Systems, pp. 156–170 (online), DOI: 10.1007/978-3-642-40698-0_12, Springer Berlin Heidelberg (2013).

[3] Argobots Home: https://collab.cels.anl.gov/display/ARGOBOTS/Argobots+Home.

[4] Omni Compiler Project: http://omni-compiler.org/.

[5] 大輔杉山，珍泌　李，均　村井，三久佐藤：軽量スレッドライブラリ Argobots を用いた OpenMP の設計，技術報告 4，筑波大学, 理化学研究所計算科学研究機構, 理化学研究所計算科学研究機構, 筑波大学／理化学研究所計算科学研究機構 (2015).

[6] Argo OS: http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[7] Tanaka, Y., Taura, K., Sato, M. and Yonezawa, A.: *Languages, Compilers, and Run-Time Systems for Scalable Computers: 5th International Workshop, LCR 2000 Rochester, NY, USA, May 25–27, 2000 Selected Papers*, chapter Performance Evaluation of OpenMP Applications with Nested Parallelism, pp. 100–112 (online), DOI: 10.1007/3-540-40889-4_8, Springer Berlin Heidelberg (2000).

[8] アドナン，三久佐藤：Flexible Fine Grain Thread : Management By StackThreads/MP Library for OpenMP Task (ハイパフォーマンスコンピューティング (HPC) Vol.2010-HPC-125)，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2010, No. 7, pp. 1–8（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110007995486/⟩ (2010).

[9] Olivier, S. L., Porterfield, A. K., Wheeler, K. B., Spiegel, M. and Prins, J. F.: OpenMP Task Scheduling Strategies for Multicore NUMA Systems, *Int. J. High Perform. Comput. Appl.*, Vol. 26, No. 2, pp. 110–124 (online), DOI: 10.1177/1094342011434065 (2012).

[10] Portable Hardware Locality: https://www.open-mpi.org/projects/hwloc/.

[11] EPCC OpenMP micro-benchmark suite: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite.

[12] Nanjegowda, R., Hernandez, O., Chapman, B. and Jin, H. H.: *Evolving OpenMP in an Age of Extreme Parallelism: 5th International Workshop on OpenMP, IWOMP 2009 Dresden, Germany, June 3-5, 2009 Proceedings*, chapter Scalability Evaluation of Barrier Algorithms for OpenMP, pp. 42–52 (online), DOI: 10.1007/978-3-642-02303-3_4, Springer Berlin Heidelberg (2009).

[13] Rodchenko, A., Nisbet, A., Pop, A. and Luján, M.: *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, chapter Effective Barrier Synchronization on Intel Xeon Phi Coprocessor, pp. 588–600 (online), DOI: 10.1007/978-3-662-48096-0_45, Springer Berlin Heidelberg (2015).

[14] NAS Parallel Benchmarks: https://www.nas.nasa.gov/publications/npb.html.

[15] SNU NPB Suite: http://aces.snu.ac.kr/software/snu-npb/.

[16] Himeno Benchmark: http://accc.riken.jp/supercom/himenobmt/.

[17] Bonnichsen, L. and Podobas, A.: *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings*, chapter Using Transactional Memory to Avoid Blocking in OpenMP Synchronization Directives, pp. 149–161 (online), DOI: 10.1007/978-3-319-24595-9_11, Springer International Publishing (2015).

ⓒ 2016 Information Processing Society of Japan