Vol.2015-HPC-150 No.4
2015/8/4

軽量スレッドライブラリ Argobots を用いた OpenMP の設計

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

概要：Argobots は、エクサスケールの大規模な並列システムに向けてシステムソフトウェアとランタイムの研究開発を行う Argo プロジェクトの一部として開発が進められている、ユーザーレベルの軽量スレッドライブラリである。Argobots を利用しメニーコアを効率的に利用するためのプログラミングモデルの第一段階として OpenMP によるプログラミングモデルを提案し、Argobots による OpenMP の基本的な設計の検討と初期評価を行った。Omni OpenMP コンパイラを用い、従来 Pthreads で実装されていた部分を Argbots を用いて実装した。オペレーティングシステムで提供されている Pthreads よりも高速のコンテキストスイッチが可能であることにより、メニーコアプロセッサである Xeon Phi において同期の高速化を確認できた。しかしループのネスト並列化によって並列性を増やした場合の性能に関しては問題があり、現在原因を調査中である。

1. はじめに

現在、最先端の計算科学に用いられる高性能計算システムの性能はペタフロップス (1 秒間に 1,000 兆回の演算能力) に達し、ポストペタスケールシステムとして、エクサスケールのシステムに向かおうとしている。ポストペタスケールシステムとその先のエクサスケールコンピュータは、チップ内にコアを多く内蔵するメニーコアプロセッサをノードとした、大規模な並列システムが一つの形態として予想されている。

メニーコアプロセッサを用いた並列システムを効率的にプログラミングするためには、ノード内での多数の並列性の制御、負荷均衡化だけでなくメモリ参照の局所性の維持、さらにノード間の効率的な通信を可能とする計算と通信を統合するプログラミングモデルなど、様々な問題を解決しなくてはならない。そのための基盤的なソフトウエアを目指して、米国 Argonne National Laboratory (ANL) において、エクサスケールの大規模な並列システムに向けて、システムソフトウエアとランタイムの研究開発を行う Argo プロジェクト [1] が進められている。

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

Argobots[2] は、Argo の一部として開発が進められている、ユーザーレベルの軽量スレッドライブラリである。我々は、Argobots を利用しメニーコアを効率的に利用するためのプログラミングモデルの第一段階として OpenMP によるプログラミングモデルを提案する。OpenMP は、共有メモリシステム向けのプログラミングモデルであり、最新の OpenMP 4.0 においてはタスク並列などの柔軟で不規則な並列性の記述ができるように拡張されている。

Argobots は、ユーザーレベルの軽量スレッドライブラリであり、従来のオペレーティングシステムで提供されている POSIX Threads Library (Pthreads) よりも高速のコンテキストスイッチが可能である。メニーコアプロセッサにおいては、多数のコアを利用するために多くの並列性をあらわにすることが必要であるが、多くの並列性を効率的に実行するためにはコアへの動的な割り当てなどの制御が必要である。また、スレッドの制御だけでなくデータ参照の局所性を高める工夫も必要である。Argobots は、メニーコアプロセッサを抽象化した Argobots Abstract Machine を定義しており、コアと共有メモリへの操作を Abstract Machine での API として提供している。システム設計者は、この API を用いて、実行時システムを設計する。

本稿では、Argobots を活用するための OpenMP の基本的な設計の検討と初期評価について報告する。OpenMP システムとしては、Omni OpenMP コンパイラを用い、従来 Pthreads で実装されていた部分を Argobots を用いて実装した。さらに、ネストされた並列ループやタスク構文での

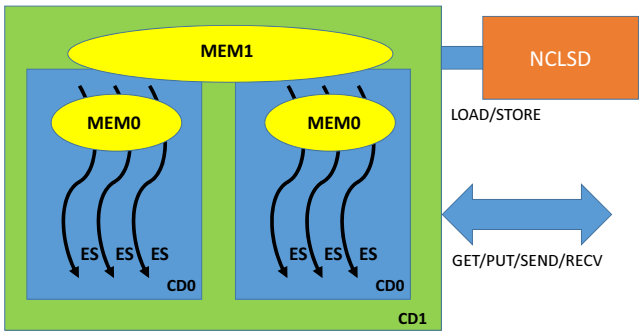

図 1　Argobots Virtual Machine

動的なタスク生成・制御などを Argobots のユーザーレベルスレッドを用いて、実行することも検討した。Pthreads では、基本的にハードウエアでサポートされるスレッドと一致させる必要があるのに対し、Argobots ではユーザーレベルの軽量スレッドであるために並列性ごとに軽量スレッドを割り当て、実行時ルーチンでの制御で効率的にスケジューリングし、あらわになった多くの並列性を利用して効率的に実行できることが期待される。

なお、本研究は、日米科学技術協定に基づくエクサスケールコンピューティングのためのシステムソフトウエアに関する日米共同研究の一部として進めている。Argobots は、MPI をはじめとする通信ライブラリと統合して用いることを前提として設計されており、この共同研究では、Argobots による OpenMP を PGAS モデルを提供する XcalableMP[3] と統合して用いることを計画している。

第 2 章においては、ユーザーレベル軽量スレッドライブラリ Argobots について述べ、第 3 章において、我々の OpenMP 実装である Omni OpenMP コンパイラでの Argobots による実行時システムの設計と実装について述べる。第 4 章においては、初期評価として、マイクロベンチマークによる評価と簡単なアプリケーションによる評価について報告する。第 5 章においては、結論と課題、これからの計画について述べる。

2. Argobots の概要

Argobots は ANL が開発を進めているスレッドライブラリである。大量の細粒度タスクによる並列処理を実現するためのフレームワークとして、スレッドの生成やスケジューリングのための API を提供する。直接利用するより、並列プログラミングモデルや通信ライブラリのランタイム実装で使われることを想定した設計になっているため、利便性より実装の自由度を優先した低レベル API を提供する。また実行プラットフォームとして Argobots が定義するような低レベル API を明示的に実行することが可能なアーキテクチャを仮定している。

図 2 に Argobots の実行プラットフォームである Argobots Virtual Machine のアーキテクチャを示す。ES は後

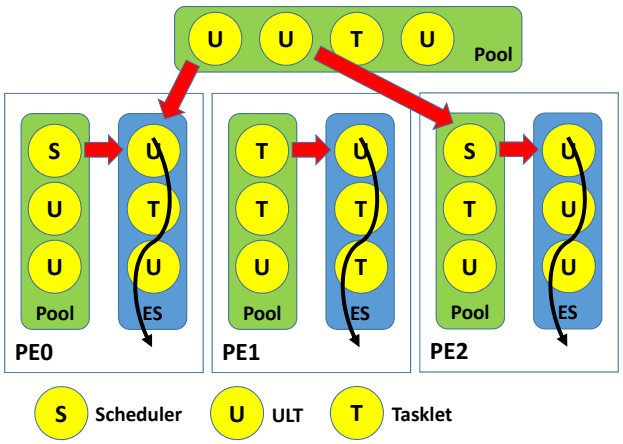

図 2　Argobots の実行モデル

述するが、実行を担当するスレッドのようなものである。Argobots Virtual Machine は 3 種類のメモリモデルの階層によって構成される。まず、ハードウェアによって値の一貫性が保たれる Consistency Domain (CD) がある。CD はデータが共有される範囲によって階層構造を持つことが許される。次に Non-Coherent Load/Store Domains (NCLSD) 領域がある。NCLSD 上のデータはハードウェアによる一貫性の調整は行われず、LOAD/STORE のような明示的な命令によって値の変更が反映される。NCLSD も CD と同様に共有範囲によって階層を持つ。NCLSD より外のメモリ領域は GET/PUT のような通信関数の実行によってデータを転送する。Argobots Virtual Machine ではデータの一貫性や値の転送をこのようなモデルによって明確に定義し、低レベル API による性能チューニングを可能にしている。本稿では Omni OpenMP コンパイラのランタイム実装のために Argobots を利用することを考え、本章では Argobots のアーキテクチャや API の概要について述べる。

2.1 実行モデル

図 2.1 に Argobots のスレッド実行モデルを示す。Processing Element (PE) は CPU コアのように計算を実行するハードウェア資源を表す。図 2.1 の PE 以外の要素はソフトウェアによって実装された概念である。

Working Unit (WU) は Argobots の実行ユニットを表す。Argobots ではすべての計算は User Level Thread (ULT) か Tasklet の WU で抽象化されて実行される。ULT はユーザレベルで実装されたスレッドである。既存のスレッドライブラリが提供するスレッドと同等の機能を持つが、スレッドの生成とコンテキストスイッチがより高速である。Tasklet は関数を抽象化した軽量実行ユニットである。Tasklet による計算は引数のみに依存し、計算の結果は関数の戻り値として返される。ULT がスレッド固有のスタックを持つ反面、Tasklet はランタイムのスタックを利用し

```
void ULT_func(void *arg) {
  printf("ULT: %d¥n", (int)arg);
}
int main(int argc, char *argv[]) {
  . . .
  ABT_xstream_self(&es[0]);
  for (int i = 1; i < NUM_ES; i++)
    ABT_xstream_create(ABT_SCHED_NULL, &es[i]);
  for (int i = 0; i < NUM_ES; i++)
    ABT_xstream_get_main_pools(es[i], 1, &pool[i]);
  for (int i = 0; i < NUM_ES; i++)
    ABT_thread_create(pool[i], ULT_func, (void *)i,
                      ABT_THREAD_ATTR_NULL, &thread[i]);
  ABT_thread_yield();
  for (int i = 0; i < NUM_ES; i++) {
    ABT_thread_join(thread[i]);
    ABT_thread_free(&thread[i]);
  }
  for (int i = 1; i < NUM_ES; i++) {
    ABT_xstream_join(es[i]);
    ABT_xstream_free(&es[i]);
  }
  . . .
```

図 3　Argobots のコード例

```
int shd = . . .;
Int prv = . . .;
#pragma omp parallel firstprivate(prv)
{
  int x = . . .;
  foo(&shd, prv, x);
}                                    OpenMPコード
```

```
void _parallel_func_0(int &shd, int prv) {
  int shd_temp = *shd;
  int x = . . .;
  foo(&shd_temp, prv, x);
  *shd = shd_temp;
}

int shd = . . .;
int prv = . . .;
ompc_do_parallel_main(_parallel_func_0, &shd, prv);
                                    変換されたコード
```

図 4　Omni OpenMP コンパイラによる OpenMP コード変換

て実行される。

WU を保持するコンテナとしてプールが用意されており、プールの中の WU を Execution Stream (ES) が実行する。Argobots の実行モデルでは WU は ES の中で逐次に実行され、複数の WU が一つの ES 内で同時に実行されることはない。ULT は yield 関数呼び出しによる明示的なコンテキストスイッチによって実行の途中で他の ULT や Tasklet に切り替わることを許すが、Tasklet は実行開始から終了まで ES を占有する。ライブラリの設計上、一つの PE に一つの ES を割り当てることを仮定している。ES の動作は他の ES と独立であるため、異なる ES にスケジュールされた WU は同時に実行される。

Argobots では図 2.1 のように ES 毎にプライベートなプールを持つだけでなく、すべての ES でプールを共有することも可能である。プールの中でどの WU が選択されるのかは ES のスケジューラが決定する。ES の初期化時にデフォルトのスケジューラが生成されるが、必要に応じてユーザがスケジューラを作成することも可能である。スケジューラそのものも ULT や Tasklet と同様、プールにスケジューリング可能なオブジェクトである。このような機能を用いることでユーザがワークスティーリング [4][5] のようなタスクスケジューリングを指定することができる。

2.2　コード例と API 関数

図 2.2 に C 言語による Argobots のコード例を示す。図には省略されているが、ランタイムの初期化と終了に ABT_init() と ABT_finalize() が呼ばれる。ES の生成は ABT_xstream_create() で行う。ランタイムが内部で ES を PE に割り当てる。ES[0] は Primary ES と呼ばれるもので、プログラム開始時に暗黙的に生成される。各 ES のプライベートプールのハンドルは ABT_xstream_get_main_pools() 関数で取得する。ULT の生成は ABT_thread_create() 関数で行う。引数として関数のポインターと引数が渡される。ES のプールに ULT を追加することで ES が動作を開始する。Tasklet の生成も ULT に類似した API が提供される。

ES[0] では main() 関数を実行する Primary ULT が暗黙的に生成される。実行を Primary ULT から ULT_func() 関数を実行する ULT に移すために明示的に ABT_thread_yield() 関数を呼ぶ。各 ES で ULT_func() 関数の実行が終了すると ABT_thread_join() 関数で ULT の同期を行う。すべての WU の実行が終了して ES が待機状態になった時、ABT_xstream_join() 関数を用いて ES の動作を終了させる。

この他にも POSIX Threads Library (Pthreads) で提供されているようなロック変数や条件変数を生成・操作するための API を提供する。

3.　Omni OpenMP での実装

本章では Argobots を用いた Omni OpenMP コンパイラのランタイム実装について述べる。Omni OpenMP 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研究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 [6][7]。最初に Pthreads を使った既存の実装について説明し、次に Argobots によって実装された新しいランタイムについて述べる。Omni OpenMP コンパイラは OpenMP 3.0 から導入された task 指示文に対応していないため、本章の実装と次章の性能評価ではループ文の並列実行によるデータ並列化をターゲットにする。

3.1 Pthreads による実装

OpenMP のスレッド並列化は parallel 指示文によって記述する。parallel 指示文はスレッドが並列に動作する並列領域 (parallel region) を宣言する。スレッド並列化の実装を説明するために、まず Omni OpenMP コンパイラによる OpenMP コードの変換について述べる。図 3.1 にコードの変換例を示す。parallel 指示文で指定されたブロック文は関数化されるが、ブロック文の中で利用される変数が関数の引数として渡される。最後に parallel 指示文とブロック文は Omni OpenMP コンパイラのランタイム関数 ompc_do_parallel_main() に置き換えられる。

ompc_do_parallel_main() 関数はスレッドの生成とブロック文の並列実行を行う。Pthreads を用いた既存の実装では OpenMP のスレッドと POSIX スレッドが 1 対 1 に対応する。POSIX スレッドの生成は OS カーネルのシステムコード呼び出しを伴うため、実行が parallel 指示文に到達したら POSIX スレッドを生成するアプローチではオーバーヘッドが大きい。Omni OpenMP コンパイラの実装ではプログラム開始時にスレッドプールを生成し、プールの中には omp_get_max_threads() と同じ数の POSIX スレッドが生成される。ompc_do_parallel_main() 関数ではスレッドプールにある待機状態のスレッドにブロック文の実行を割り当てる。

ブロック文の実行が終わったら暗黙のバリア同期が行われる。スレッド間の同期通信の実装には Pthreads のロック変数や条件変数のようなスレッドプリミティブが使われている。barrier や reduction 指示文の実装も同様の機能を用いて実装されている。

3.2 Argobots による実装

まず Argobots の実行モデルと OpenMP の構成要素の関連付けを考える。Argobots ではスレッド間の並列実行を保証するためには CPU コア毎に ES を割り当てることが必須である。そのため、Argobots を用いた OpenMP の実装ではプログラムの開始時に CPU コア数と同じ数の ES を生成する。現在の実装ではまだワークスティーリングを考慮しないので WU プールは ES 毎に生成する。

Pthreads による実装の場合、OpenMP スレッドは POSIX スレッドに 1 対 1 で対応づけることができたが、Argobots の場合も OpenMP スレッドと ULT を 1 対 1 で対応させることができる。Argobots の ULT 操作のインターフェースは Pthreads API に類似しており、ロック変数や条件変数などのスレッド間通信のためのプリミティブも提供されている。したがって、OpenMP ランタイムの Pthreads 関数を Argobots のものに置き換えることで Argobots によるスレッド並列化を実現する。

Pthreads の実装ともっとも異なるところは OpenMP スレッドの生成が ompc_do_parallel_main() 関数の中に移動したことである。Argobots の ULT は POSIX スレッドと比べてスレッド生成とコンテキストスイッチが高速に行えるので並列領域の実行開始時にスレッドを生成するようにしている。このように実装することで CPU コア数より多いスレッドが必要な細粒度の並列実行にも対応することができる。不均等な負荷分散や今後の課題である task 指示文による動的なタスク生成でもスレッドスケジュールを工夫することによってハードウェア計算資源を効率的に利用することができると期待する。

OpenMP ランタイムの実装効率を高めるために Argobots の機能追加を行った。Pthreads 版 OpenMP ランタイムでは自スレッドの情報を参照するためにスレッドハンドルのメモリアドレスをキーとするハッシュテーブルをヒープで管理している。本稿は従来のデータ並列化より多くのスレッドが生成されるアプリケーションや実行プラットフォームを仮定しているため、ヒープ上の共有メモリオブジェクトによるスレッド情報の管理は望ましくない。自スレッドの情報を管理するだけの場合だと Pthreads では Thread Local Storage (TLS) を用いた実装が考えられる。しかし Argobots では TLS 機構を持たないので ULT のスレッドハンドルにメモリポインターを追加することで TLS の実装を行った。これによって parallel 指示文開始時に発生するスレッド間の同期 (ハッシュテーブルの管理) を回避することができた。

3.3 今後の実装の課題

現在の実装では WU プールを ES 毎に持っているため、ワークロードが ES 間で不均等に分散された場合は計算効率が低下する。今後の実装の方針としてすべての ES で共有される WU プールを作成し、ワークスティーリングのようなスレッドスケジューリングを実装することを目指す。また、Intel Xeon Phi のようなハードウェアスレッドを複数持つアーキテクチャにおいて、ES をどのように生成するか検討が必要である。Intel Xeon プロセッサなどでは物理コアと同じ数の ES を生成することでスレッドの並列実行を保証することができる。しかし、Xeon Phi コプロセッサでは物理コアの中でさらに複数のハードウェアスレッドが存在し、それらを効率よく使うことによって性能向上を得ることができる。Argobots の使い方としてクロックサイクルによって実行が切り替わるハードウェアスレッドをどのように活用できるか検討を続ける必要がある。

4. 性能評価

本章では Argobots を用いて実装した Omni OpenMP コンパイラ利用して、OpenMP 指示文のマイクロベンチマークとラプラス方程式のソルバーと姫野ベンチマークの性能評価を行う。実行プラットフォームとしてマルチコアアーキテクチャの Intel Xeon プロセッサとメニーコアアー

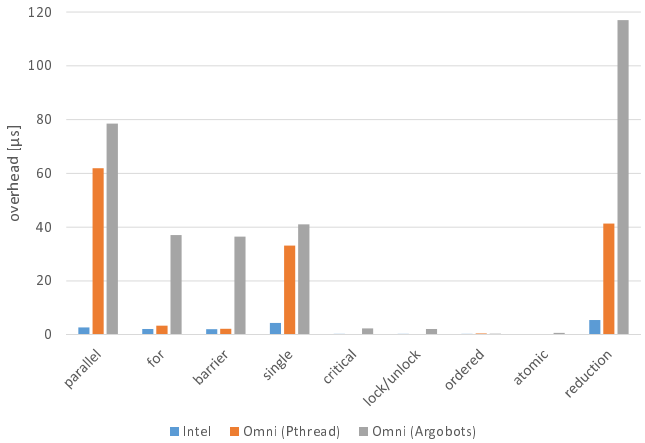

図 5 EPCC Syncbench の評価結果 (Xeon 24 スレッド)

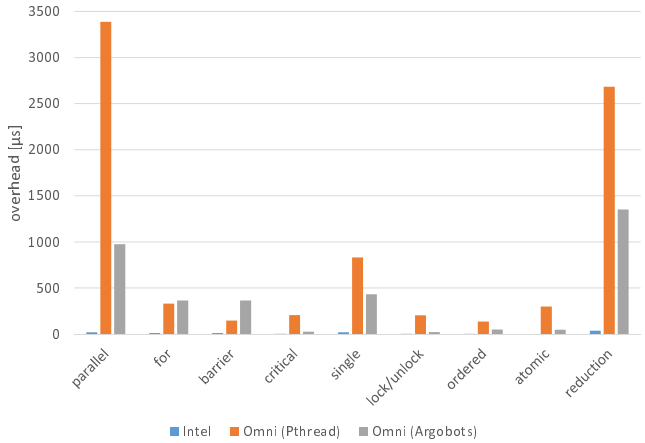

図 6 EPCC Syncbench の評価結果 (Xeon Phi 60 スレッド)

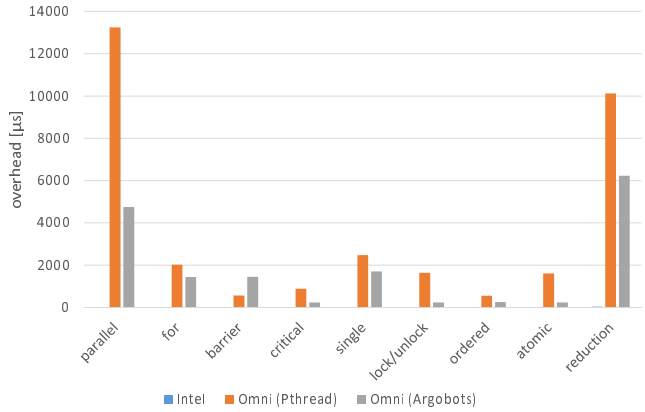

図 7 EPCC Syncbench の評価結果 (Xeon Phi 240 スレッド)

キテクチャを持つ Intel Xeon Phi コプロセッサを用いた。Xeon Phi の評価結果はネイティブモードで実行されたものである。評価環境を表 1 と表 2 に示す。比較対象として Pthreads 版 Omni OpenMP コンパイラと Intel OpenMP コンパイラでも同様の評価を行った。

4.1 EPCC OpenMP マイクロベンチマーク

EPCC OpenMP Microbenchmark Suite[8] は OpenMP のランタイムのオーバーヘッドを測定するベンチマークである。本稿ではスレッドの生成やスレッド間の同期を Argobots によって実装したのでスレッド間の同期を伴う OpenMP 指示文の性能を測定する syncbench を用いて評価を行った。syncbench では parallel、barrier、reduction などの指示文の実行時間を測定する。

図 3.3 に Xeon プロセッサでの評価結果を示す。性能評価には物理コア数と同じになるように 24 スレッドを生成している。全体的に Intel OpenMP コンパイラと比べて Omni OpenMP コンパイラの性能が低いことがわかる。Argobots 版が Pthreads 版と比べて性能が低下している原因は現在調査中である。

次に図 3.3 に Xeon Phi コプロセッサでの評価結果を示す。性能評価には物理コア数と同じになるように 60 スレッドを生成している。Pthreads 版と比べて Argobots で同期プリミティブの性能が高いことがわかる。性能が向上した原因としてスレッドのコンテキストスイッチのコストが考えられる。例えば Pthreads によるバリア同期のランタイムでは一定時間 spin-lock を実行した後に sched_yield() 関数によるコンテキストスイッチが行われる。Argobots による実装でも sched_yield() が ABT_thread_yield() 関数に置き換わるだけで、バリア同期のアルゴリズムは同じである。その場合、OS スレッドを利用する Pthreads より軽量スレッドを利用する Argobots の方がコンテキストスイッチにかかるコストの分、性能面で有利であると考えられる。評価の結果、大量のスレッドを生成する Xeon Phi 環境では軽量スレッドによって同期プリミティブを実装することで OpenMP 指示文の実行性能が向上することが確認できた。ただし、Intel コンパイラの結果と比べると実行時間に大きい差があることがわかる。性能差の原因はまだ判明しておらず現在調査中である。

最後に図 3.3 に Xeon Phi コプロセッサで 240 スレッドを用いた場合の評価結果を示す。データの傾向は図 3.3 とほぼ一緒であるが、全体の実行時間が増加していることがわかる。増加の原因は現在の Argobots 版 OpenMP 実装でハードウェアスレッドの総数と同じ数の ES を生成しているからだと考えられる。ハードウェアスレッドは命令のスループットを増加させることには寄与するが、スレッドの

表 1 Intel Xeon 評価環境

	名称
プロセッサ	Intel Xeon-E5 2670 v3 × 2 ソケット (24 コア, 2.30 GHz)
メモリ	DDR4 64GB
コンパイラ	Intel Compiler 15.0.2 20150121

表 2 Intel Xeon Phi 評価環境

	名称
プロセッサ	Intel Xeon Phi 7120P (61 コア, 1.238 GHz)
メモリ	GDDR5 16GB
コンパイラ	Intel Compiler 14.0.1 20131008

```
#pragma omp parallel
{
#pragma omp for
  for (int x = 1; x < SIZE - 1; x++)
    for (int y = 1; y < SIZE - 1; y++)
      uu[x][y] = u[x][y];
#pragma omp for
  for (int x = 1; x < SIZE - 1; x++)
    for (int y = 1; y < SIZE - 1; y++)
      u[x][y] = (uu[x - 1][y] + uu[x + 1][y]
    + uu[x][y - 1] + uu[x][y + 1]) / 4.0;
}
```

図 8　ラプラスソルバーの計算カーネル

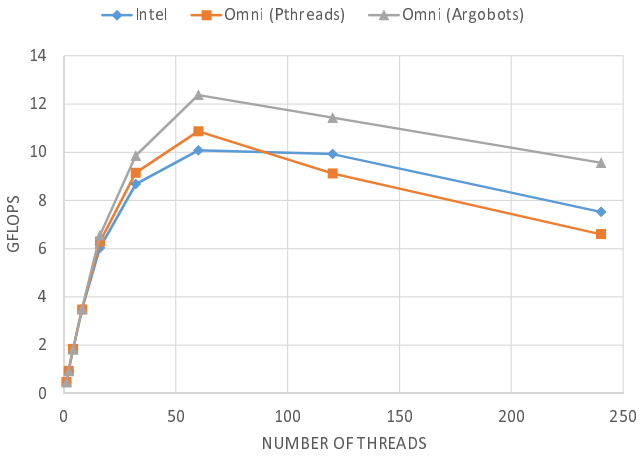

図 9　ラプラスソルバーの評価結果

同時実行を保証するものではない。

4.2　ラプラスソルバー

次に 2 次元ラプラス方程式を差分法で解くソルバーを OpenMP で並列化して実行時間を評価する。図 4.2 にソルバーの計算カーネルコードを示す。計算カーネルは外側のイテレーションループと内側の計算を行う二重ループで構成される。イテレーションループが parallel 指示文の内側にあるため、並列領域の生成は 1 回のみである。

図 4.2 にラプラスソルバーの評価結果を示す。これから述べる性能評価の結果はすべて表 2 の Xeon Phi システムを利用して得られたものである。Argobots 版の性能がもっとも高く、次に Intel OpenMP 版、Pthreads 版の順で性能が低下していくことがわかる。逐次の実行性能がすべてのバージョンで同等であるため、ソルバーの性能差はスレッド並列化の実装に由来するものと考えられる。しかし、具体的な原因は判明できておらず、現在調査中である。

4.3　姫野ベンチマーク

姫野ベンチマーク [9] は 3 次元ポワソン方程式をヤコビ反復法で解くことで非圧縮流体解析コードの性能評価を行うものである。最外ループ文を OpenMP で並列化したものを Xeon Phi で実行し、その結果を図 4.3 に示す。

図 3.3 で示したように reduction のような同期プリミティブは軽量スレッドをあつかう Argobots でより高い性能を達成する。その結果、Pthreads 版より Argobots 版の姫野ベンチマークが高い性能を達成することが図 4.3 の結果からわかる。しかし、Intel OpenMP 版と比べると性能に開きがある。特にスレッド数が物理コア数と同じである 60 までは Intel と同等の性能を見せるがそれを超えたあたりから性能向上率が低下する。

原因として Argobots が物理コアの数を超えた ES の数を生成する場合に非効率的な ES のスケジューリングが行われていることが考えられる。Intel OpenMP ではスレッドの affinity を明示しているが、Argobots 版では物理コアと ES のバインドは Argobots ランタイムと OS に依存している。複数のハードウェアスレッドを利用して命令のスループットを上げることは重要であるが、生成された ES とハードウェアスレッド、物理コアのスケジューリングが非効率的に行われる場合は図 3.3 で見られるような性能低下が発生すると考えられる。

姫野ベンチマークは 3 次元データを扱うため、3 つのループ文が並列化の対象になる。Xeon Phi のような大量のスレッドを生成するプラットフォームでは 1 つのループを並列化するだけでは計算資源を使いきれない可能性がある。そのような問題点を解決するために複数の並列領域を階層構造で生成するネスト並列化が考えられる。図 4.3 に

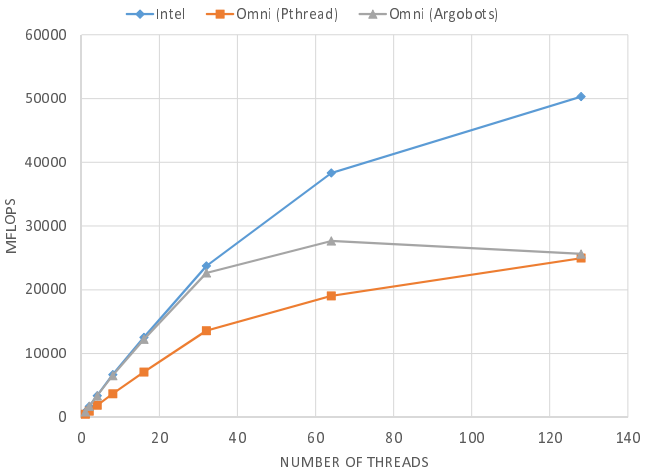

図 10　姫野ベンチマークの評価結果

```
#pragma omp parallel private(i, j, k, n, s0, ss)
{
  for(n=0; n<nn; ++n) {
    gosa = 0.0;
#pragma omp for reduction(+:gosa)
    for(i=1; i<imax-1; i++) {
#pragma omp parallel
{
#pragma omp for reduction(+:gosa)
      for(j=1; j<jmax-1; j++) {
        for(k=1; k<kmax-1; k++) { . . .
```

図 11　姫野ベンチマークのネスト並列化

ネスト並列化が行われた姫野ベンチマークのループ構造を示す。外側の二重ループ文が parallel 指示文によって並列化されている。これによって 1 つのループ文を並列化したバージョンと比べて並列度が向上し、より多くのスレッドでも性能が向上することが期待される [10]。

しかし、性能評価の結果ではネストされた並列化が行われる場合は内側の同期のオーバーヘッドが蓄積されるので性能がさらに低下することがわかった。また、2 つの parallel 指示文で大量のスレッドを生成し、Argobots のランタイムでそれらのスケジューリングが効率的に行われていないことが原因として考えられる。

Xeon Phi 上でマイクロベンチマークや姫野ベンチマークを用いた性能評価の結果、Pthreads を用いた実装と比べると同期プリミティブの性能向上が見られるものの、Intel OpenMP コンパイラと比べて実行性能とスケーラビリティで課題があることがわかった。

5. 結論と今後の課題

エクサスケールコンピュータの実現に向けてチップ内に多くのコアを内蔵したメニーコアプロセッサへの関心が高まっている。そしてメニーコアプロセッサの性能を引き出すために大量の細粒度タスクの並列実行をサポートする軽量スレッドライブラリ Argobots が提案されている。本稿では Argobots の並列プログラミングモデルとして OpenMP を用いることを提案し、Omni OpenMP コンパイラのデータ並列化のためのランタイムを Argobots で実装した。Intel Xeon Phi コプロセッサを用いた性能評価の結果、従来の POSIX スレッドより性能が向上するものの、Intel OpenMP コンパイラで得られた性能に比べると最適化が不十分だということがわかった。

今回の調査でわかったことを踏まえて今後の課題として以下のようなことを考える。

- parallel 指示文をネストした場合に Argobots 版 OpenMP のランタイム性能が低下する原因を調査し解消する。
- 1 つの WU プールを複数の ES で共有する、あるいはワークスティーリングを行うスケジューラを用いるなど、より計算資源の利用効率が良くなるような仕組みを導入する。
- Xeon Phi のような複数のハードウェアスレッドを持つシステムで計算資源を効率よく利用できるような OpenMP ランタイムの実装を考える。
- Omni コンパイラに OpenMP 3.0 の task 構文を導入し、Argobots によるランタイムの実装を行う。
- task 構文を用いて Unbalanced Tree Search のような、動的なタスクの割り当てが必要となる問題を使って性能評価を行う。

最終的にはタスク並列化を記述できる構文の導入とランタイムの最適化によってメニーコアプロセッサの性能を引き出せるような並列プログラミングモデルの実現を目標とする。

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。

参考文献

[1] Argo OS: http://www.mcs.anl.gov/project/argo-exascale-operating-system.
[2] Argobots Home: https://collab.cels.anl.gov/display/ARGOBOTS/Argobots+Home.
[3] XcalableMP: http://xcalablemp.org/.
[4] Olivier, S. L., Porterfield, A. K., Wheeler, K. B. and Prins, J. F.: Scheduling Task Parallelism on Multi-socket Multicore Systems, *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, New York, NY, USA, ACM, pp. 49–56 (online), DOI: 10.1145/1988796.1988804 (2011).
[5] Liaskovitis, V., Chen, S., Gibbons, P. B., Ailamaki, A., Blelloch, G. E., Falsafi, B., Fix, L., Hardavellas, N., Kozuch, M., Mowry, T. C. and Wilkerson, C.: Parallel Depth First vs. Work Stealing Schedulers on CMP Architectures, *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, New York, NY, USA, ACM, pp. 330–330 (online), DOI: 10.1145/1148109.1148167 (2006).
[6] Omni Compiler Project: http://omni-compiler.org/.
[7] Sato, M., Shigehisa, M. S., Kusano, K. and Tanaka, Y.: Design of OpenMP Compiler for an SMP Cluster, In EWOMP '99, pp. 32–39 (1999).
[8] EPCC OpenMP micro-benchmark suite: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite.
[9] Himeno Benchmark: http://accc.riken.jp/2145.htm.
[10] 義純田中，健次朗田浦，明憲米澤：OpenMP におけるネストした並列性の実装と評価，情報処理学会論文誌プログラミング（PRO），Vol. 41, No. 2, pp. 54–64（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110002725348/⟩ (2000).