Vol.2016-HPC-155 No.29
2016/8/9

軽量スレッドライブラリ Argobots による PGAS 言語 XcalableMP の動的タスク並列機能の設計

津金 佳祐[1,a)]　中尾 昌広[2]　李 珍泌[2]　村井 均[2]　佐藤 三久[1,2]

概要：近年，高性能計算分野においてチップ内に多くのコアを搭載するメニーコアプロセッサを用いた大規模並列システムが登場している．そのようなシステムにおける並列化手法の一つとしてタスク並列が注目されており，本稿では，分散メモリ環境上での動的なタスク並列処理をより簡易な記述で実装可能とすべく PGAS 言語 XcalableMP（XMP）の拡張を行う．記述として tasklet 指示文を提案し，ノード内/外におけるタスク間の依存関係の記述による細粒度な同期や，通信と計算のオーバラップによる性能向上を目指す．タスク生成や制御は Argonne National Laboratory（ANL）により開発が進められている軽量スレッドライブラリである Argobots を用いる．現在は，実装対象である Omni XMP Compiler のランタイムのみの実装であるため，コード変換は手動で行う．提案手法の予備評価としてブロックコレスキー分解のコードを対象とし，性能・生産性の評価を行った．比較対象は，MPI+OpenMP による実装と INRIA によって開発が進められている StarPU による実装である．StarPU 実装との比較では一部優位な点が見られたが，MPI+OpenMP 実装との比較では最大で約 15%の性能低下が見られた．生産性の比較では，指示文による記述を採用したことによりベースコードから少ない変更で実装可能なことから，tasklet 指示文による記述の生産性の高さを示した．

1. はじめに

近年，高性能計算分野において消費電力性能比が良いことからチップ内に多くのコアを搭載したメニーコアプロセッサが登場している．2016 年 6 月に公開されたスーパーコンピュータの性能ランキングである Top500[1] において 1 位を得た Sunway TaihuLight の Sunway プロセッサや，Intel が開発している Xeon Phi プロセッサ，消費電力性能比ランキングである Green500[2] において 1 位を得た Shoubu の PEZY-SC プロセッサなど，メニーコアプロセッサが広く注目を集めていることがわかる．そのため，今後メニーコアプロセッサを持つ大規模な並列システムはより増加すると考えられる．そのようなシステムにおけるプログラミングモデルとして性能や生産性の観点から，ノード内の並列化手法としてタスク並列，ノード間のプログラミングモデルとして Partitioned Global Address Space（PGAS）モデルが注目されている．

タスク並列は再帰的構造や while ループなど演算量が動的に決定する場合の並列処理を容易に記述可能とするとともに，処理系による動的な負荷分散を行う点を特徴としている．タスク並列をサポートするプログラミングモデルとして Threading Building Blocks（TBB）[3]，Cilk Plus[4]，OpenMP[5]，OmpSs[6] や Chapel[7] などがあり，タスク制御のための様々な機能の提案・実装が行われている．例えば OpenMP では，仕様 4.0 より登場した `task` 指示文の `depend` 節により，タスクの依存関係を記述可能となった．従来のタスク全体の同期ではなくタスク間の細粒度な同期を動的に処理することが可能となり，大量のコアを持つメニーコアな環境において性能向上が期待される．

一方で，分散メモリ環境上でのプログラミングモデルとしては Message Passing Interface（MPI）が広く普及している．しかし，MPI はプロセス毎のデータの分散配置や複雑な通信の記述など，並列化を行う上の様々な処理手順を明示的に示す必要があるため，プログラミングの学習コストが高くソースコードが煩雑になりやすいといった生産性の低下が問題となっている．そこで，分散メモリ環境上での並列プログラミングをより容易にするために開発されたのが，PGAS 言語 XcalableMP（XMP）[8][9][10] である．XMP は，既存言語（C，Fortran）の拡張であり，OpenMP に似た指示文を用いてループの並列化やノード間通信を記述可能である．そのため，既存の逐次プログラムを大きく

1 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
2 国立研究開発法人理化学研究所計算科学研究機構
RIKEN Advanced Institute for Computational Science
a) tsugane@hpcs.cs.tsukuba.ac.jp

ⓒ 2016 Information Processing Society of Japan

変更することなく容易に並列化を行うことができる．XMP とタスク並列を記述可能なプログラミングモデルを組み合わせることで，分散メモリ環境上のタスク並列処理を記述することは可能である．しかし，ノード内/外で異なるタスク依存の記述が必要となり，プログラムが複雑になりやすい．

以上の背景を踏まえ本稿では，分散メモリ環境上でのタスク並列処理をより簡易な記述で実装可能とすべく XMP の拡張を行う．記述として tasklet 指示文を提案し，ノード内/外におけるタスク間の依存関係を統一的に記述することによる細粒度な同期や，通信と計算のオーバラップによる性能向上を目指す．実装対象を XMP のリファレンスコンパイラである Omni XMP Compiler[11] とし，ランタイムの設計・実装を行う．ブロックコレスキー分解のコードを tasklet 指示文，MPI+OpenMP やタスク並列を記述可能な関連研究により実装し，tasklet 指示文による実装の性能・生産性を評価する．XMP における動的タスク並列機能は，XMP の次期仕様として PC クラスタコンソーシアムの並列プログラミング言語 XcalableMP 規格部会にて検討中であり，本提案はその記述の一つである．

著者らは研究報告 [12] にて XMP における動的なタスク並列機能の提案は既に行っている．しかし，タスク内で演算に用いられるデータの送受信をノードを跨ぐタスク間の依存関係とし，指示文内に通信を記述することから，コードの見通しが悪く性能チューニングがしづらい記述となっている．そこで本稿では，タスク依存とデータ通信を切り分け，タスク間の依存関係は専用の変数であるイベントにより記述し，データ通信は tasklet 指示文のブロック内にてユーザが自由に記述可能とする．また，スレッドライブラリとして OpenMP の task 指示文の depend 節を用いて tasklet 指示文の予備実装をしたが，タスク生成やコンテキストスイッチが高速な軽量スレッドライブラリにより，さらなる高速化が可能であると考える [13][14]．そこで本稿では，Argonne National Laboratory（ANL）にて開発が進められている軽量スレッドライブラリである Argobots[15] を用いて実装する．

本稿の構成を以下に示す．2 章では関連研究を紹介する．3 章では Argobots，4 章では XMP の概要の説明を行う．5 章は提案する指示文である tasklet 指示文の概要とその実装を解説し，6 章でブロックコレスキー分解のコードを用いて性能・生産性の評価を行う．最後に 7 章でまとめと今後の課題を述べる．

2. 関連研究

分散メモリ環境においてタスク並列を記述可能なプログラミングモデルとして StarPU[16] や QUARK-D[17] がある．StarPU は，INRIA により開発されている CPU や演算加速機構を対象にタスク並列や負荷分散を記述可能なランタイムライブラリである．記述方法としてタスク制御のために codelet と呼ばれる構造体を使用し，実行関数，実行リソースや依存関係などをタスク単位で予め記述し，タスク生成時に用いる方法をとる．QUARK-D は，テネシー大学により開発が進められている分散メモリ環境を対象としたタスク並列処理を記述可能なランタイムライブラリである．QUARK-D が提供する API により StarPU 同様に実行関数，実行リソースや依存関係などを登録しタスク並列を行う．

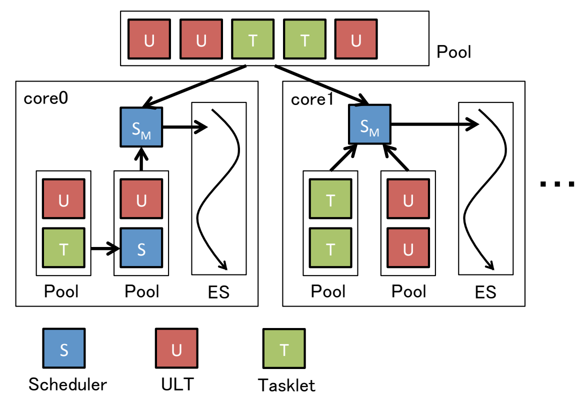

図 1 Argobots の実行モデル

タスク並列を記述可能なプログラミングモデルを用いて様々なプログラムの開発が進められている．例えば，INRIA が提供する KASTORS benchmark suite[18] では，OpenMP の task 指示文や depend 節を用いてポアッソン方程式をヤコビの反復法で解くソルバー，シュトラッセンのアルゴリズムを用いた行列積及び LU 分解を行うプログラムなどを提供している．また Barcelona Supercomputing Center（BSC）では，OpenMP に似たプログラミングモデルである OmpSs が開発されており，パッケージ内にてコレスキー分解や stream のなどのコードを公開している．

3. 軽量スレッドライブラリ Argobots

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[19] の一部として研究開発が進められている，ユーザレベルの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，スレッド制御機構やタスクのスケジューリングのための API を提供している．**図 1** に Argobots の実行モデルを示す．Argobots の実行単位である Working Unit（WU）は，User Level Thread（ULT）と Tasklet の 2 種類がある．ULT は既存のスレッドライブラリと同等の機能をユーザレベルで提供しており，mutex や条件変数などの排他制御や同期機構に加え，高速なタスク生成・コンテキストスイッチを特徴としている．Tasklet は関数を抽象化した軽量実行ユニットであり，高速に実行可能だが ULT が持つ排他制御や同期機構を持たず，コンテキストスイッチもできないなど制約も強い．

```
int A[N];
#pragma xmp nodes P(4)
#pragma xmp template T(0:N-1)
```

0 N−1
T : template T

```
#pragma xmp distribute T(block) onto P
```

0 N/4−1 N/2−1 3*N/4−1 N−1
T : node1 node2 node3 node4

```
#pragma xmp align a[i] with T(i)
```

0 N/4−1 N/2−1 3*N/4−1 N−1
T : node1 node2 node3 node4
A[N] : node1 node2 node3 node4

```
#pragma xmp loop(i) on T(i)
for (i = 0; i < N; i++) { A[i] = func(i); }
```

図 2 グローバルビューモデルによるプログラミング例

各 WU は直接コアに割り当てられて実行されるのではなく，WU を入れるためのキューであるプールに入れられ，プール内の WU を Execution Stream（ES）が実行する．図 1 の例では CPU の各物理コアに対して一つの ES が割り当てられている．ES に紐付けられたマスタースケジューラによりプールから WU を取り出し ES 上で逐次的に実行するが，それぞれの ES は並列に動作するため並列実行が可能である．プールには WU の他にスケジューラを入れることも可能である．図 1 の core0 のマスタースケジューラに割り当てられたプール内にあるスケジューラが実行可能になった場合は，そのスケジューラが持つプール内の WU が実行される．また，core1 のプールのようにマスタースケジュラに複数のプールを割り当てることや，core0・core1 で共有のプールを設定することも可能であり，ES 間でのタスクのスティーリングやマイグレーションもユーザが実装可能である．

4. PGAS 言語 XcalableMP

XMP は，次世代並列プログラミング言語検討委員会及び PC クラスタコンソーシアム並列プログラミング言語 XcalableMP 規格部会により，仕様検討及び策定されている分散メモリ型 SPMD（Single Program Multiple Data）を実行モデルとする並列言語である．リファレンス実装である Omni XcalableMP Compiler は，XMP 指示文により記述されたコードを MPI コードへと source-to-source 変換を行うトランスレータであり，筑波大学と理化学研究所により開発が進められている．XMP の実行単位は”ノード”である．XMP はプログラミングモデルとしてグローバルビューとローカルビューの二種類を提供している．グローバルビューモデルは典型的なデータ分散や通信を指示文で

```
1  int A[100], B[25];
2  #pragma xmp nodes P(4)
3  #pragma xmp template T(0:99)
4  #pragma xmp distribute T(block) onto P
5  #pragma xmp align A[i] with T(i)
6
7  #pragma xmp task on P(4)
8  #pragma xmp gmove in
9        B[0:25] = A[0:25];
10
11 #pragma xmp task on P(1)
12 #pragma xmp gmove out
13       A[75:25] = B[0:25];
```

図 3 gmove in/out の例

```
1  int A[25]:[*], B[25], mype, status;
2  #pragma xmp nodes P(4)
3  mype = xmp_node_num();
4
5  if (mype == 4)
6        B[0:25] = A[0:25]:[1];
7  xmp_sync_all(&status);
8
9  if (mype == 1)
10       A[0:25]:[4] = B[0:25];
11 xmp_sync_all(&status);
```

図 4 ローカルビューモデルによるプログラミング例

提供しており，ローカルビューモデルは coarray 記法による片側通信を提供している．XMP は様々な指示文を提供しているが，本稿では `tasklet` 指示文に関係する機能のみを説明する．

4.1 グローバルビューモデル

グローバルビューモデルは，問題で扱うグローバルな配列を各ノードに分散する指示文を記述することで，並列実行を行うプログラミングモデルである．そのため，基本的には逐次実装に指示文を挿入するのみで並列実装を行うことが可能である．**図 2** にグローバルビューモデルによるプログラミング例を示す．ノード毎にデータの分散を行うには，テンプレートと呼ばれる仮想的なインデックス空間を用いる．まず，`node` 指示文により実行ノード集合（プロセス数）を定義し，`template` 指示文によりテンプレートサイズを指定する．次に，テンプレートに対して `distribute` 指示文により分割方法（ブロック，サイクリック，ブロック・サイクリック及び，不均等ブロック）の指定を行い，`align` 指示文で対象の配列と分割されたテンプレートを対応付けることで，各ノードへとデータの分散を行う．これらの動作は全て指示文によるものであり，ユーザは各ノードへと分散されたデータの配置を意識することなく，分割したいループに対して記述された `loop` 指示文により，並列実行を行うことが可能となる．また，グローバルビュー

モデルによる並列プログラムは，基本的に XMP 指示文を無視することで逐次実装の C，Fortran 言語によるプログラムとして解釈することが可能である．

XMP は定型的な通信を行うための指示文を提供しているが，本節ではその一つである gmove 指示文の説明のみとする．gmove 指示文は分散配列，ノードローカルな配列や変数に対する代入処理を行うことが可能な指示文である．対象が分散配列の場合はノード間での通信が発生し，さらに，in/out 節が指定された場合は片側通信（Get，Put）となる．**図 3** に gmove in/out 指示文の例を示す．2 から 5 行目の指示文によって分割された配列 A はノード毎に 25 要素を持つ．task 指示文は on 節で指定されたノードのみブロック内の処理を実行する指示文であり，7 行目の場合はノード 4 のみの実行となる．9 行目より配列 A の 25 要素を配列 B へと代入するが，8 行目の gmove in 指示文があるため配列 A のインデックス 0 から 25 要素を持つノード 1 が対象となる．したがって，ノード 4 はノード 1 からリモートリード（Get）を行う．同様に gmove out 指示文では，11 から 13 行目よりノード 1 自身が持つ配列 B の要素をノード 4 が持つ配列 A のインデックス 75 から 25 要素に向けてリモートライト（Put）を行う．また，片側通信の完了保証は指示文内にて自動で行われる．

4.2 ローカルビューモデル

ローカルビューモデルは，各ノードが持つローカルデータに対して通信を行うプログラミングモデルである．XMP では，Fortran2008 から正式採用された Co-array Fortran[20] をベースとした coarray 記法が実装されている．配列代入文形式で記述可能であり，ノード番号，通信要素数を指定することによる片側通信を行うため，MPI のようにローカルデータの振る舞いを詳細に記述できる．**図 4** に coarray を用いた片側通信の例を示す．配列の後ろのコロン以降が通信相手を表す．図 3 と同様の通信パターンを coarray 記法で記述しており，右辺にノード番号が記述された場合はリモートリード，左辺に記述された場合はリモートライトとなる．coarray では，gmove in/out 指示文とは異なり通信完了をユーザが保証する必要がある．図 4 では，xmp_sync_all を用い全ノードの通信完了を保証している．

5. XcalableMP における動的なタスク並列機能を実現する指示文

本章では，提案する記述方法である tasklet 指示文の概要と，MPI と Argobots を用いた Omni XMP Compiler のランタイムの実装を示す．

5.1 tasklet 指示文

分散メモリ環境における動的なタスク並列機能の実現には，ノード内でのタスク生成・依存関係の処理に加え，ノードを跨ぐタスク間の依存関係を処理できる必要がある．ノード内におけるタスクの依存関係は，共有メモリを用いることから依存があるタスクの情報をタスク生成時にランタイム内で判断することができるが，ノードを跨いだ場合は不可能である．そこで本稿では，タスク終了時に依存関係があるタスクを持つノードに対して通信を発生させ，その通信のマッチングにより依存関係とする方法をとる．

図 5 に tasklet 指示文の記述方法を示す．tasklet 指示文はユーザが OpenMP のように in, out を用いた依存関係の記述に加え，ノードを跨いだタスク依存を表すための通信相手を記述する．依存関係には専用の変数であるイベントを用い，in/out 節に記述する．本稿では，xmp_event_t 型で定義されたものをイベントとする．ノード間の依存関係は必ず out 節から in 節とし，out 節のみ依存があるノードを指定するためのノード番号やテンプレートを記述可能とする．また，タスクを実行するノードは on 節で指定されたノード集合またはテンプレートにより決定する．データの通信は gmove in/out 指示文や coarray を用い，tasklet 指示文のブロック内にユーザが記述する．また，通信の完了保証はタスク終了時に自動的に行われる．

図 6 に tasklet 指示文によるプログラミング例とそのタスクフローを示す．配列 A は分散配列であり 2 ノードで実行されるため，50 要素ずつ各ノードに分散される．ノード 1 では taskA と taskB が生成されるがタスク間に依存関係は無いため同時に実行される．また，taskA において tasklet 指示文のブロック内では gmove out 指示文が指定されているため，ノード 1 が持つ配列 B の 50 要素を分散配列 A へ代入する．分散配列 A のインデックス 50 から 50 要素はノード 2 が持つため，ノード 1 からノード 2 へのリモートライトを表す．ノード 2 では，taskC と taskD が生成される．taskC と taskD の間にはイベント EC の依存関係があるため逐次に実行される．また，ノード 1 の taskA の out 節では，イベント EA にてノード 2 が指定されているため，ノード 2 のイベント EA に対して依存関係がある．したがって，in 節にてイベント EA，EC が指定された taskD は，taskC の終了と taskA からの通信完了を待って実行を開始する．

現在，XMP 規格部会において 2 種類の依存関係の記述方法が提案されている．まず一つは，OpenMP の task 指示文の depend 節のように，out 節から in 節の順序で実行されるフロー依存以外に，出力依存（out 節から out 節）や反依存（in 節から out 節）も考慮する手法である．tasklet 指示文が OpenMP による記述に似ているため，既存コードの再利用が可能である．しかし，フロー依存以外に出力依存や反依存のユーザが意図していない依存関係が発生する可能性があるため，挙動が直感的ではなく性能低下の原因となりやすい．もう一方は，全ての依存関係に別々のイベントを割り当て，全ての依存関係は必ず out 節から in

ⓒ 2016 Information Processing Society of Japan

```
#pragma xmp tasklet tasklet-format[, tasklet-format, ...] on {node-ref | template-ref }
  (structured-block)

  where tasklet-format is :
    in (event_variable[, event_variable, ...])
    or
    out (event_variable[: [{ int-expr | template-ref }]][, event_variable[: [...]], ...])
```

図 5　XMP tasklet 指示文

```
int A[100], B[50];
xmp_event_t EA:[*], EB, EC;

#pragma xmp nodes P(2)
#pragma xmp template T(0:99)
#pragma xmp distribute T(block) onto P
#pragma xmp align A[i] with T(i)

#pragma xmp tasklet out(EA:[2]) on P(1)
{
   taskA();
#pragma xmp gmove out
   A[50:50] = B[0:50]; /* Put */
}
#pragma xmp tasklet out(EB) on P(1)
   taskB();
#pragma xmp tasklet out(EC) on P(2)
   taskC();
#pragma xmp tasklet in(EA, EC) on P(2)
   taskD();
```

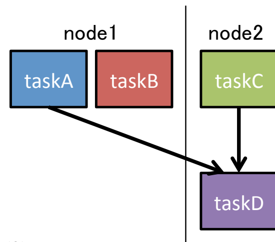

図 6　tasklet 指示文によるプログラミング例とタスクフロー

節の順序で実行する方法である．この手法の場合，依存関係はフロー依存しかなくユーザが意図していない依存関係は発生しづらい．しかし，全ての依存関係に別々のイベントを割り当てる必要があるためユーザによるイベント管理が複雑でコードが煩雑になりやすい．本稿では，前者の OpenMP の `task` 指示文に則った記述方法による実装のみを示す．

5.2　Argobots による実装

Argobots による `tasklet` 指示文の実装を示す．実装方針として，タスク生成や制御には Argobots を用い，ノード間におけるタスク依存は MPI による通信を用いる．本稿では，各物理コアに対して ES とプールをそれぞれ一つ割り当てる．タスクの生成は逐次的に行われ，`tasklet` 指示文のブロック内の処理を一つの ULT とし各プールに順番に挿入する．生成されたタスクは，マスタースケジューラによってプールから取り出され，ES 上で実行される．依存関係を満たしていれば演算を実行し，そうでなければコンテキストスイッチにより実行中のタスクをプールに戻し，次のタスクを実行する．

タスク間の依存関係は，タスク生成時の情報とイベントが持つカウンタにより処理される．まず，タスクの実行条件を示す．`in` 節を持つタスクの場合，依存関係はフロー依存のみであるため，直前の `out` 節を持つタスクの終了により実行が開始される．`out` 節を持つタスクの場合，出力依存と反依存の二つの依存関係があるため，直前の `out` 節を持つタスクとそのタスク以降に生成された `in` 節を持つ全てのタスクの終了により実行が開始される．以上の実行条件を表すため，本稿ではカウンタによる実装を行った．カウンタは `in`/`out` 節毎にイベント内にて保持され，それらの節を持つタスクが終了した場合にそれぞれをインクリメントする．タスク生成は逐次的に行われるため，タスク生成時に `in`/`out` 節のそれぞれを数え生成番号をタスクに付与することが可能である．ES から取り出されたタスクはイベント内のカウンタの値と自らが持つ生成番号とを比較し，同一（依存関係のあるタスクが終了している）ならば実行を開始する．

ノード間の依存関係の場合は MPI による通信を用いる．`out` 節にて指定されたノード番号もしくはテンプレートにより，依存関係のあるノードを判断し通信を開始する．`in` 節を持つタスクでは通信を受け取るまで待機するが，`MPI_Test` によりブロッキングせずに通信完了を確認する．通信が完了していれば処理を開始し，完了していなければ Argobots によるコンテキストスイッチを発生させ，別のタスクを実行する．同一のイベントが指定された `in` 節を持つタスクが複数ある場合，最初に生成されたタスク内にて通信を行い，通信完了を待って他の `in` 節を持つタスクの実行を開始する．

`tasklet` 指示文では，`in` 節を持つタスクにおいて通信の有無を判断することができない場合がある．例えば OpenMP の場合，`depend` 節に `in` を指定されたタスクは，そのタスク以前に生成された `out` または `inout` のタスクとのみ依存関係があるため，それらのタスクが生成されてなければ直ちに実行される．しかし，`tasklet` 指示文の `in` 節の場合は，以前に生成されたタスクがないのか，通信待ちが必要なタスクかの判断ができない．そのため，本稿における `tasklet` 指示文の依存関係の記述は，OpenMP の `task` 指示文と全く同じではなく，ノード内において `out` 節を持つタスクが生成されていない `in` 節を持つタスクの場合は通信待ちを行い，`out` 節を持つタスクが生成されていた場合は，通信待ちを行わずノード内のフロー依存のみとする．

ⓒ 2016 Information Processing Society of Japan

表 1　実験環境（COMA）

CPU	Intel Xeon E5-2670v2 × 2
	CPU（10 cores/CPU）× 2 = 20 cores
Memory	64GB
Interconnect	Infiniband FDR Connect-X3（56 Gbit/s）
OS	Red Hat Enterprise Linux Server 6.4
Compiler	Intel Compiler 16.0.2
MPI	Intel MPI 5.1.3
Intel MKL	11.3.2

```
1  double A[nt][nt][ts*ts];
2  #pragma omp parallel
3  #pragma omp single
4  {
5    for (int k = 0; k < nt; k++) {
6  #pragma omp task depend(out:A[k][k])
7      omp_potrf (A[k][k], ts, ts);
8
9      for (int i = k + 1; i < nt; i++) {
10 #pragma omp task depend(in:A[k][k]) depend(out:A[k][i])
11       omp_trsm (A[k][k], A[k][i], ts, ts);
12     }
13     for (int i = k + 1; i < nt; i++) {
14       for (int j = k + 1; j < i; j++) {
15 #pragma omp task depend(in:A[k][i], A[k][j]) \
16                  depend(out:A[j][i])
17         omp_gemm (A[k][i], A[k][j], A[j][i], ts, ts);
18       }
19 #pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
20       omp_syrk (A[k][i], A[i][i], ts, ts);
21     }
22   }
23 #pragma omp taskwait
24 }
```

図 7　OpenMP task 指示文によるブロックコレスキー分解のコード

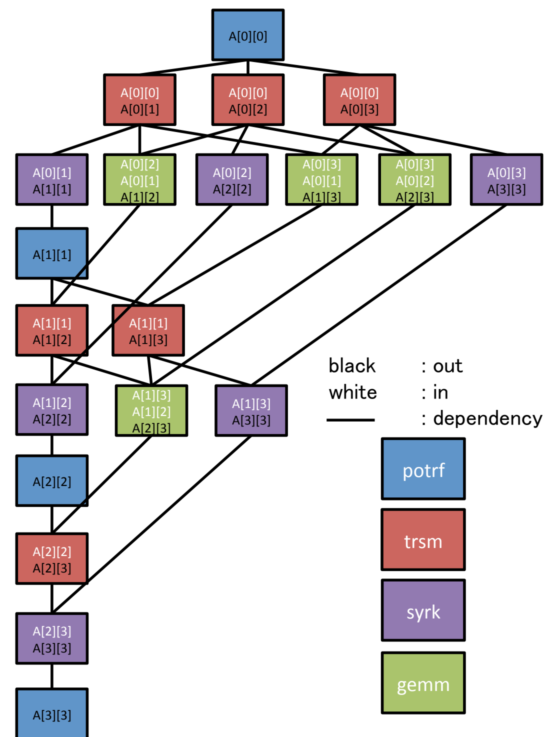

図 8　4 × 4 ブロックを持つブロックコレスキー分解のタスクフロー

6. 性能評価

tasklet 指示文を用いたプログラムの評価には筑波大学計算科学研究センターの COMA[21] を利用する．1 ノードあたりの構成や使用ソフトウェアのバージョンは**表 1** に示す通りである．1 ノードあたり 1MPI プロセスを割り当て，最大 4 ノード 4MPI プロセスで評価する．また，ノード内のスレッド数は 1 から 16 へと変動させる．対象のプログラムをブロックコレスキー分解とし，XMP（tasklet 指示文），MPI+OpenMP 及び StartPU による 3 種類の実装による性能・生産性の評価を行う．

6.1　ブロックコレスキー分解の概要と評価

ブロックコレスキー分解を解くコードは BSC OmpSs のパッケージ内で提供されているコードを用いる．ブロックコレスキー分解は以下の 4 つの処理にわけられる．括弧内は BLAS や LAPACK のルーチン名を表す．

(1) コレスキー分解（potrf）
(2) 三角行列を係数行列とする行列方程式を解く（trsm）
(3) 対称行列のランクを更新（syrk）
(4) 行列積 (gemm)

上記の処理は全てブロック単位で行われる．**図 7** に OpenMP の task 指示文の depend 節を用いて実装したブロックコレスキー分解のコード，**図 8** にブロック数が 4 × 4 の行列を対象としたタスクフローを示す．ブロックコレスキー分解のそれぞれの処理には依存関係が存在し，例えば k = 0 の場合，potrf(A[0][0]) は，trsm(A[0][0], A[0][1])，trsm(A[0][0], A[0][2]) 及び trsm(A[0][0], A[0][3]) と依存関係がある．しかし，trsm 間には依存関係がないため並列実行が可能であり，空いているスレッドがあれば順次スケジューリングされ実行される．

tasklet 指示文により実装されたブロックコレスキー分解のコードの評価を行う．**図 9** に tasklet 指示文を用いて実装したブロックコレスキー分解のコードを示す．本稿では，Omni XMP Compiler のランタイムのみの実装であるため，ランタイム呼び出しへの変換は手動で行う．行列サイズは 4096 × 4096 と 8192 × 8192 の 2 種類を用い，1 ブロックサイズは 128 × 128 とする．分割方法は 1 次元のサイクリック分割とする．また，ブロックコレスキー分解の 4 つの処理には全て Intel Math Kernel Library（MKL）を用いる．

図 10 に行列サイズ 4096 × 4096，**図 11** に行列サイズ 8192 × 8192 の評価を示す．横軸の”XMP”は tasklet 指示文による実装，”MPI”は MPI+OpenMP による実装を表す．結果として，StarPU 実装と比較すると行列サイズ

ⓒ 2016 Information Processing Society of Japan

```
1  double A[nt][nt][ts*ts]; /* distributed array */
2  double B[ts*ts], C[nt][ts*ts]; /* local array */
3  xmp_event_t EA[nt][nt]:[*], EB, EC[nt];
4  #pragma xmp nodes P(*)
5  #pragma xmp template T(0:nt−1)
6  #pragma xmp distribute T(cyclic) onto P
7  #pragma xmp align A[*][i][*] with T(i)
8
9  for (int k = 0; k < nt; k++) {
10 #pragma xmp tasklet out(EA[k][k]:[T(k+1:)]) on T(k)
11   xmp_potrf(A[k][k]);
12
13 #pragma xmp tasklet in(EA[k][k]) out(EB) on T(k+1:)
14 #pragma xmp gmove in
15   B[:] = A[k][k][:];
16
17   for (int i = k + 1; i < nt; i++) {
18 #pragma xmp tasklet in(EB) out(EA[k][i]:[T(i+1:)]) on T(i)
19     xmp_trsm(B, A[k][i]);
20   }
21
22   for (int i = k + 1; i < nt; i++) {
23 #pragma xmp tasklet in(EA[k][i]) out(EC[i]) on T(i+1:)
24 #pragma xmp gmove in
25     C[i][:] = A[k][i][:];
26
27     for (int j = k + 1; j < i; j++) {
28 #pragma xmp tasklet in(EA[k][i], EC[j]) out(EA[j][i]) on T(i)
29       xmp_gemm(EA[k][i], C[j], EA[j][i]);
30     }
31 #pragma xmp tasklet in(EA[k][i]) out(EA[i][i]) on T(i)
32     xmp_syrk(EA[k][i], EA[i][i]);
33   }
34 }
35 #pragma xmp taskletwait
```

図 9 XMP tasklet 指示文によるブロックコレスキー分解のコード

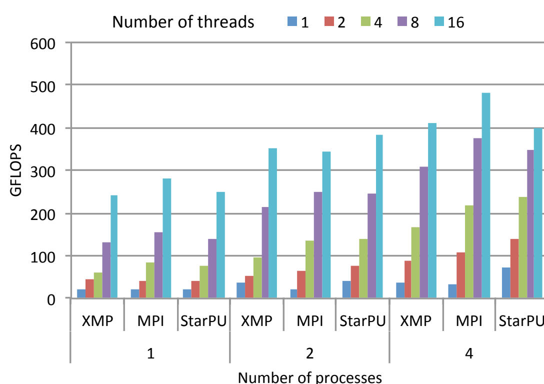

図 10 ブロックコレスキー分解の評価（行列サイズ 4096 × 4096, ブロックサイズ 128 × 128）

8192 × 8192 の 4 ノード 16 スレッド時に最大で約 10%の性能向上がみられる．しかし，多くの場合において XMP による実装の性能はほぼ同等か低い．また，MPI+OpenMP 実装と比較すると，ノード数が少ない場合の性能は，StarPU と同様にほぼ同等か低く，さらに，ノード数が増加した場合に最大で約 15%の性能低下が起きた．現在，詳しい原因を調査中だが，一つの原因として，現在の実装におけるタスクの生成は各 ES のプールに対して順番に挿入するだけであり，依存関係を満たせなかった場合に発生するコンテキストスイッチが頻発していることがあげられる．そのため今後は，終了したタスクが自らと依存関係があるタスクを起動するような実装とすることで，できるだけコンテキストスイッチを減らし性能向上させられるような実装を行うことを考えている．

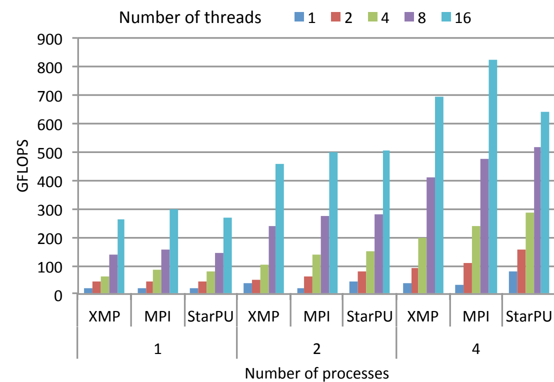

図 11 ブロックコレスキー分解の評価（行列サイズ 8192 × 8192, ブロックサイズ 128 × 128）

表 2 ブロックコレスキー分解のコード行数

	Serial	OMP	MPI+OMP	StarPU	XMP
SLOC	318	330	582	494	338
modified	-	0	13	31	2
added	-	12	264	176	20
deleted	-	0	0	0	0

6.2 生産性

`tasklet` 指示文による実装の生産性を定量的に評価するため，各プログラミングモデルによる実装のコード行数を比較する．比較手法として逐次実装からの差分（修正，追加，削除の行数）で評価を行う delta-SLOC 方式 [22] を用いる．**表 2** にそれぞれのプログラミングモデルによるブロックコレスキー分解の行数と逐次実装からの差分を示す．まず，全体の行数として `tasklet` 指示文による実装は，MPI+OpenMP 実装と比較して 58%，StarPU 実装とは 68%までコード行数を抑えることができた．また，逐次実装からの各差分を比較すると，XMP 実装は MPI+OpenMP，StarPU 実装よりも修正・追加行が少なく，ほぼ OpenMP 実装と同等と言える．これは，MPI+OpenMP 実装では必要なノード間の依存関係を表すための通信記述を指示文に内包し，データの分散配置をグローバルビューによる指示文のみの記述としたためである．一方で，StarPU は暗黙にタスク間で通信を行うため，通信を記述する必要はないが，初期化時に全ノードの分散データ配置の登録やタスク内の処理の関数化，タスク制御のための codelet の記述により修正・追加行が増加している．以上より，`tasklet` 指

示文による実装の生産性は高いと言える．

7. まとめ

本稿では，分散メモリ環境上での動的なタスク並列処理をより簡易な記述で実装可能とすべく PGAS 言語 XcalableMP において `tasklet` 指示文の提案・予備実装を行った．ノード内/外におけるタスク間の依存関係を記述可能とすることで，細粒度な同期や通信と計算のオーバラップを可能とした．タスクの生成・制御には軽量スレッドライブラリである Argobots を用い，ノードを跨ぐタスク間の依存関係には MPI の通信を用いた．`tasklet` 指示文の初期評価として，ブロックコレスキー分解のコードを対象とし，MPI+OpenMP と StarPU による実装と性能・生産性の比較を行った．結果として，StarPU 実装と比較すると一部優位な点も見られたが，MPI+OpenMP 実装と比較した場合，最大で約 15%の性能低下が起きた．また，生産性の比較では，指示文による記述方法を採用したことにより，ベースコードからの少ない変更点で実装可能なことから，`tasklet` 指示文による記述の生産性の高さを示した．

今後の課題として，6.1 節で述べたタスク生成方法の工夫や，性能低下の詳しい原因調査を行い性能を向上させることがあげられる．また，Omni XMP Compiler の `tasklet` 指示文のコードトランスレータ部分を実装し性能評価を行うことや，メニーコアアーキテクチャである Intel Xeon Phi を用いた評価などがあげられる．

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学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ッワークに関する研究」による．また本研究の評価は，筑波大学計算科学研究センターの平成 28 年度学際共同利用プロジェクト「アクセラレータおよびメニーコアを搭載したクラスタシステムのための高生産並列言語の開発と評価」（代表者：中尾昌広）を利用して得られたものである．

参考文献

[1] Top500 Supercomputer Sites, https://www.top500.org/
[2] The Green500, http://www.green500.org/
[3] Intel Threading Building Blocks, https://www.threadingbuildingblocks.org/
[4] Intel CilkPlus, https://www.cilkplus.org/
[5] OpenMP Application Programming Interface Version 4.5, http://openmp.org/wp/, 2015
[6] Duran A, Ayguade E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J, Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures, Parallel Process Lett 21(2):173-193.
[7] Chapel language specification version 0.98, http://chapel.cray.com/, 2015
[8] XcalableMP Language Specification Version 1.2.1, http://www.xcalablemp.org/, 2014
[9] J. Lee, M. Sato, Implementation and Performance Evaluation of XcalableMP: a Parallel Programming Language for Distributed Memory Systems, Proc. 39th Int. Conf. Parallel Process. Workshops (ICPPW), San Diego, (2010) pp. 413–420.
[10] M. Nakao, J. Lee, T. Boku, M. Sato, Productivity and Performance of Global-view Programming with XcalableMP PGAS Language, Proc. 12th IEEE/ACM Int. Symposium Cluster, Cloud Grid Comput (CCGrid), Ottawa, (2012) pp. 402–409.
[11] Omni Compiler Project, http://omni-compiler.org/
[12] 津金 佳祐, 中尾 昌広, 李 珍泌, 村井 均, 佐藤 三久, PGAS 言語 XcalableMP における動的タスク機能の提案, 情報処理学会研究報告, Vol.2015-HPC-151, No.5, pp. 1–7, 沖縄, 2015 年 10 月
[13] 杉山 大輔, 李 珍泌, 村井 均, 佐藤 三久, 軽量スレッドライブラリ Argobots を用いた OpenMP の設計, Vol.2015-HPC-150, No.4, pp. 1–7, 大分, 2015 年 8 月
[14] 李 珍泌, 杉山 大輔, 村井 均, 佐藤 三久, 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, Vol.2016-HPC-153, No.20, pp. 1–8, 愛媛, 2016 年 3 月
[15] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, A. Castell, D. Genet, T. Herault, P. Jindal, L. V. Kal, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, P. Beckman, Argobots: A Lightweight, Low-Level Threading and Tasking Framework, ANL/MCS-P5515-0116, (2016) pp. 1-12
[16] C. Augonnet and R. Namyst, A unified runtime system for heterogeneous multicore architectures. In Proceedings of the International Euro-Par Workshops 2008, HPPC ’08, volume 5415 of LNCS, (2008)
[17] YarKhan A, Dynamic Task Execution on Shared and Distributed Memory Architectures, PhD Dissertation, Major Advisor: Jack Dongarra, University of Tennessee, (2012) pp. 1–120
[18] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, T. Gautier, Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite, 10th International Workshop on OpenMP (IWOMP), Brazil, (2014)
[19] Argo: An exascale operating system, http://www.mcs.anl.gov/project/argo-exascale-operating-system
[20] ISO/IEC 1539-1:2010,Information technology - Programming languages - Fortran -, (2010).
[21] 筑波大学計算科学研究センタースーパーコンピュータ COMA(PACS-IX) について, http://www.ccs.tsukuba.ac.jp/pr/media/press_20140414
[22] A. Stone, J. Dennis, M. Strout, Evaluating coarray fortran with the cgpop miniapp, Proc. 5th Int. Conf. PGAS Program. Models, Texas, (2011) pp. 1–10.

ⓒ 2016 Information Processing Society of Japan