Vol.2019-HPC-170 No.27
2019/7/25

分散配列上のデータ依存関係に基づく分散タスク生成方式の検討

村井 均[1]　佐藤 三久[1]　中尾 昌広[1]　李 珍泌[1]

概要：本報告では，Partitioned Global Address Space (PGAS) 言語 XcalableMP (XMP) の動的タスク並列機能における分散タスク生成方式を提案する．各ノードが分散的にタスクを生成する従来方式では，データの定義参照に基づくタスク間の依存関係を実行時に決定することができないため，ユーザがタスク間の依存関係を明示する必要があった．これに対し，本方式では，特定のノードがタスクグラフを集中的に生成し各ノードへ分配することにより，分散配列に関するデータ依存関係に基づいてタスクグラフを生成することが可能になるため，ユーザの負担は軽減される．XMP コンパイラのリファレンス実装である Omni XMP に本方式を実装した場合のコード変換を想定して，ブロックコレスキー分解のコードを手動で作成した結果，本方式による分散タスク生成が実現可能であることを確認できた．

1. はじめに

近年の HPC 環境では，メニーコアプロセッサが広く使われるようになっている．従来より，ノード内の共有メモリ並列処理のためのプログラミングモデルとしては，OpenMP によるデータ並列処理（ワークシェアリング）が多く用いられているが，通常そのようなデータ並列処理は，全てのスレッドが参加する**大域的同期**を伴う．しかし，コア数の増加とともに，大域的同期は，それ自体のオーバヘッドの大きさに加え，コア間の負荷不均衡を顕在化させることからも，性能低下の重大な要因となり得る．

それに対し，個々の処理（タスク）の並列性に基づく**タスク並列処理**は，大域的同期ではなく，タスク間の細粒度な同期に基づく記述が可能であるため，性能の向上を期待できる．

タスク並列処理においては，正しい（意図する）結果を得るために，タスク間の実行順序（依存関係）を制御する必要がある．このようなタスク間の実行順序は，一般に，タスクをノードとする有向非巡回グラフ（Directed Acyclic Graph: DAG）である「タスクグラフ」として表せる．すなわち，個々のタスク間の依存関係を記述することは，DAG を記述することに帰着し，これはプログラマの大きな負担であるとともに，バグの原因ともなると考えられる．タスクグラフを記述するための効率的な手段が，タスク並列プログラミングにおける課題の一つであると言える．

共有メモリ並列処理のためのプログラミングモデルである OpenMP は，Version 3.0 でタスク並列処理のための `task` 構文をサポートし，さらに Version 4.0 でタスク間の依存関係を記述する機能（`depend` 節）をサポートした [1]．上述の課題に対し，OpenMP のタスク機能は次のようなアプローチをとる．

- プログラマは，`depend` 節を用いて，個々のタスクが定義または参照するデータを指定する．
- 処理系は，プログラム実行時に，`depend` 節の情報に基づいて，タスク間の依存関係（タスクグラフ）を自動的に構築する．

このアプローチの巧妙な点は，逐次実行における個々のタスクのデータの定義および参照の情報のみから，自動的にタスク間の依存関係を構築できる点である．

一方，分散メモリ並列計算機上のプログラミング手段として，指示文ベースの Partitioned Global Address Space (PGAS) 言語である XcalableMP (XMP) が提案されている [2], [3], [4]．XMP は，指示文に基づく抽象度の高い記法によるグローバルビュー並列化の機能をサポートすることで，実用性と利便性を兼ね備えた並列プログラミングモデルとなっている．

XMP における動的マルチタスキングの機能を「タスクレット」と呼ぶ．我々は，文献 [5] において，タスクレット間の依存関係を定義するための記法を提案し，その性能を予備的に評価した．しかし，ノード間の依存をも含むグローバル・タスクグラフ（以下，単にタスクグラフと呼ぶ）における，すべてのノードをまたぐ有向エッジを通信として記述する必要があるこの記法は，制約が多く，プロ

[1] 理化学研究所 計算科学研究センター

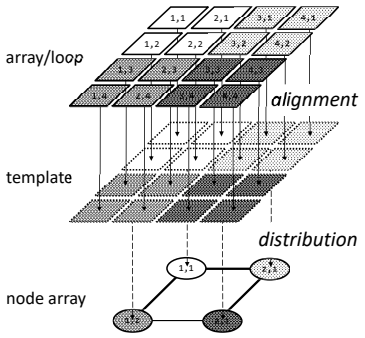

図 1 XcalableMP のデータマッピング

グラマにとって負担が大きい．また，共有メモリ環境を対象とする OpenMP の方式は，分散メモリ環境を対象とする XMP にはそのままでは適用できない．本報告では，分散配列上の依存関係に基づく，分散メモリ環境を対象とする効率的な分散タスク生成方式の検討を行う．

以下，2 章では，研究の背景として，XMP 言語と Omni XMP コンパイラをそれぞれ概観した後，OpenMP のタスク機能との比較により XMP のタスク機能の課題を抽出する．3 章で本方式の基本設計を説明し，4 章で XMP における言語仕様を提案する．続いて，5 章ではその実装方式について，6 章では予備評価のためのプロトタイプ実装について述べ，7 章で今後の課題を検討する．最後に，8 章で関連研究について述べ，9 章で本報告を総括する．

2. 研究の背景

2.1 PGAS 言語 XcalableMP

XcalableMP は，PC クラスタコンソーシアム並列プログラミング言語 XMP 規格部会において検討されている PGAS 言語であり，ベース言語 C および Fortran に対する指示文ベースの拡張として定義される [2].

XMP プログラムにおける実行主体を**ノード**[*1]と呼ぶ．各ノードでは複数のスレッドが動作していてもよい．XMP のグローバルビュー並列化では，指示文を用いて，複数のノード上に配列を分散配置することができる（**図 1**）．

2.2 Omni XcalableMP コンパイラ

Omni XcalableMP コンパイラは，Omni コンパイラ基盤 [6] に基づく XMP 処理系のリファレンス実装であり，理研と筑波大によって開発が続けられている．

Omni XMP コンパイラは，入力された XMP ソースプログラムを，等価な MPI ソースプログラムへ変換する．出力された MPI ソースプログラムは，ターゲット環境に応じたベース言語コンパイラによりコンパイルされ，各種のランタイムやライブラリ（e.g. MPI ライブラリ）とリンクされて，実行形式を得る（**図 2**）．

[*1] Omni XMP のように，MPI を用いる実装では，MPI プロセスに相当する．

2.3 OpenMP のタスク機能

OpenMP の `task` 構文を**図 3** に示す．`task` 構文は，実行時に次にように処理される．

(1) ある特定のスレッド（図 3 ではマスタスレッド）が，一連の `task` 構文を「実行」し，タスクを生成するとともに，`depend` 節の情報に基づいてそれらの間の依存関係を設定する．
(2) カレントチームに属するスレッドが，(1) で生成されたタスクを分担して実行する．このとき，あるタスクの実行は，(1) で設定された依存関係が満たされるまで開始されない．

マスタスレッドは逐次実行の場合と同じ順序で各 `task` 構文に遭遇し，タスクを生成するが，この時点ではタスクのコードは実行されない（かもしれない）ことに注意されたい．図 3 の例では，7 行目の `taskwait` 構文の時点で全てのタスクの実行が完了することが保証される．

タスク間の依存関係は，逐次実行時のデータの定義と参照の順序を保証するように構築される．すなわち，データ依存関係に基づきタスク間の依存関係を構築するためには，タスクが逐次的に生成されることが重要である．

2.4 XMP における課題

一方，分散メモリ環境上のタスク並列処理において，OpenMP と同様の逐次実行によるタスク生成を各ノードが独立に（分散的に）行った場合，ノードをまたがるタスク間の実行順序は一般に不定である．したがって，データ依存関係に基づき，そのようなタスク間の依存関係を構築することはできない．実際，XcalableMP Language Specification Version 1.4 に付録として収録されているタスクレット機能のドラフト仕様 [2] と，我々の以前の研究 [5] では，データ依存関係に依らない方式が用いられている（前者ではタスク間の依存関係を明示的に記述する方式，後者ではタスク間の通信を明示的に記述する方式）．

しかしながら，データ依存関係に基づく方式に比べた場合，そのような明示的な指定による方式は，実行時のタスク間の全てのインタラクションを書き下すことに相当し，プログラマの大きな負担であるとともに，バグの原因ともなると考えられる．

3. 基本設計

本稿では，誤解のない限り「タスクレット」という用語を次の 3 つの意味で用いる．

- タスクグラフのノード
- タスクレットスケジュール情報（後述）
- タスクレットスケジュール情報を元に各ノードにおいて生成され，スレッドによって処理される実行コードとデータ環境のインスタンス（cf. OpenMP タスク）

2.4 節で述べた課題に対し，本報告では，ある特定のノー

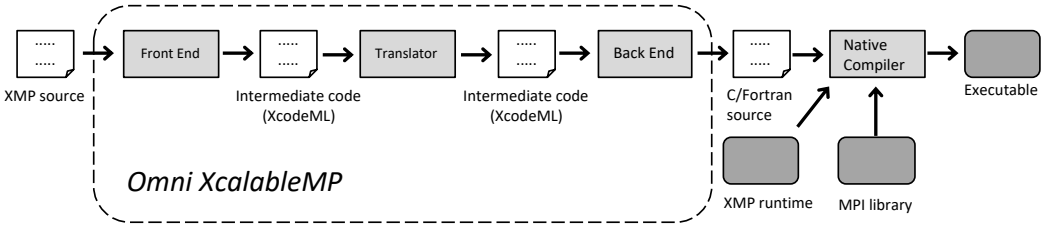

図 2　Omni XcalableMP コンパイラ

```
1 #pragma omp master
2 {
3   #pragma omp task depend(in:...) \
4                    depend(out:...)
5     ...
6 }
7 #pragma omp taskwait
```

図 3　OpenMP の **task** 構文

ド（以下，マスタノードと呼ぶ）が集中的にタスクグラフを構築し，他ノードにタスクレットを配布する，という方式を提案する．本方式により，分散メモリ環境上のタスク並列処理においても，データ依存関係に基づきタスク間の依存関係を記述することが可能になるため，プログラム生産性の向上を期待できる．

本方式の設計にあたり，以下の 4 点について考慮する必要がある．

- **タスク間の依存関係の構築**
  マスタノードは，逐次実行における各タスクレットのデータの定義および参照の情報から，タスクレット間の依存関係（タスクグラフ）を実行時に構築する．
  あるデータを定義するタスクレットからそのデータを参照するタスクレットへのエッジ（フロー依存に相当する）に対し，前者から後者へ当該データを同期的に転送する．このデータ転送により，両タスクレットの実行順序を保証する．後者のタスクレットにおいて，受信したデータは一時領域（バッファ）に保存される．
  同様に，あるデータを定義するタスクレットから同じデータを定義するタスクレットへのエッジ（出力依存に相当する）に対し，両タスクレットの実行順序を保証するための同期を行う．
  なお，上述の通り，受信したデータはバッファに保存され，タスクレット中の処理ではこのバッファを参照するようにコード変換を行う．したがって，逆依存は自然に解消される（発生しない）ため，本方式では考慮しない．
- **タスクレットのスケジュール情報の配布**
  構築したタスクグラフに基づき，各タスクレットの**スケジュール情報**を，対象のノードに配布する．ここで，タスクレットのスケジュール情報は，実行すべきコード，データ環境，受信データおよび送信データに関する情報を含むデータ構造である．
- **タスクレットの生成および実行**
  各ノードは，配布されたタスクレットスケジュール情報に基づき，タスクレットを生成および実行する．生成されるタスクレットは，他のタスクレットからデータを受信する**データ受信部**，指示文で指定された本来の処理を実行する**コード部**，他のタスクレットへデータを送信する**データ送信部**から成る．
- **PGAS への適用**
  タスクレット間の依存関係を決定する際に考慮するデータとして，本報告では，分散配列のみを対象とする．それ以外の入力データについては，タスクレットの「データ環境（パラメータ）」として，タスクレットスケジュール情報の一部としてマスタノードから各ノードへ配布することを基本とするが，指示文の指定により，各ノード上のローカルなデータを参照することも可能である．

4. XMP におけるタスク並列処理

XMP におけるタスク並列処理に関する言語仕様は，主に以下の 2 つの構文に基づく（詳細なシンタックスは次節以降で示す）．

- `tasklets` 構文は，タスク並列処理の対象となるブロックを指定する．
- `tasklet` 構文は，タスクレットとそれらの間の依存関係を定義する．

`tasklets` と `tasklet` の例を図 **4** に示す．

4.1 タスクレットの実行モデル

`tasklets` 構文と `tasklet` 構文は，実行時に次のように処理される．

(1) `tasklets` 構文の `on` 節で指定されるノード（マスタノードと呼ぶ．図 4 では `p[0]`）は，実行時に，`tasklets` ブロックに含まれる各 `tasklet` 構文を解析し，対応するタスクレットを生成するとともに，タスクグラフを構築する．
(2) マスタノードは，(1) で生成したタスクレットを各

```
1  #pragma xmp tasklets on p[0]
2  {
3
4   #pragma xmp tasklet ...
5    func0();
6
7   #pragma xmp tasklet ...
8    func1();
9
10   ...
11
12 }
```

図 4　提案方式によるタスク並列処理

ノードへ配布する．

(3) 各ノードは，(2) で配布されたタスクレットを，タスクグラフで表される依存関係（実行順序）を守りつつ実行する．

(4) マスタノードは，tasklets ブロックの出口において，すべてのタスクレットの実行の完了を待つ．

4.2　tasklets 構文

tasklets 構文のシンタックスを**図 5** に示す．tasklets 構文の on 節で指定されるマスタノードは，ブロック内に含まれる各 tasklet 構文の指示に従ってタスクレットを生成し，各ノードに割り当てる．tasklets 構文の直後の位置で，各ノードは，割り当てられたタスクレットを実行する．local 節が指定されている場合，各ノードにおけるローカルな（tasklets ブロック外の）データが，当該 tasklets 構文内のタスクレットで参照される．そうでない場合，マスタノードから配布されたデータが参照される．

4.3　tasklet 構文

tasklet 構文は，指定されたブロックを，**onto** 節で指定されるノードで実行すべきタスクレットとして定義する．そのシンタックスを**図 6** に示す．

params 節には，タスクレットの「パラメータ」を指定する．マスタノードが tasklet 構文に遭遇した時点のパラメータの値が各ノードに渡される．

depende-clause は，当該タスクレットにおけるデータの定義および参照に関する言明でり，タスクレット間の依存関係を構築するために用いられる．in 節および inout 節は，当該タスクレットにおいて，指定されたグローバルデータが参照されることを意味する．同様に，out 節および inout 節は，当該タスクレットにおいて，指定された分散配列が定義されることを意味する．

本報告では，*depend-clause* に関して，次の 2 つの制約を仮定する．

（制約 1） タスクレットは，out 節または inout 節で指定された分散配列が割り当てられているノードで実行されなければならない（cf. owner-computes rule）．

（制約 2） depend 節に現れる複数の *variable*（部分配列であってもよい）が，ある同一のデータを含む場合，それらは同一の表現でなければならない．

制約 2 は，OpenMP の task 構文における制約と等価なものである（文献 [1] の Section 2.13.9）．

5.　実装

5.1　コンパイル時の処理

ソースコード中の tasklets 構文および tasklet 構文の出現に対し，コンパイラは以下の処理を行う．

- tasklets 構文が現れる手続きの冒頭に，タスクレットを登録するランタイム（xmp_regist_tasklet）の呼び出しを挿入する（**図 7** の 1〜2 行目）．
- 各 tasklet 構文の位置に，タスクグラフのノードを追加するランタイム（xmp_create_tasklet）の呼び出しを挿入する（図 7 の 8，10，13，16 行目）．
- tasklets 構文の末尾に，タスクレットの分配（xmp_distribute_tasklet）とタスクレットの実行（xmp_tasklet_wait_all）を行うランタイム呼び出しを挿入する（図 7 の 22〜24 行目）．
- tasklet 構文で指定されたブロックを，タスクレットの本体である新たな関数として切り出す．

5.2　ランタイムの処理

コンパイラによる変換後のソースコードから呼び出される各ランタイムは，それぞれ以下の処理を行う．

- xmp_regist_tasklet
  指定された関数をタスクレットの本体として登録し，一意な ID を割り当てて返す．
- xmp_create_tasklet
  ID で指定された本体を持つタスクレットの依存関係を構築してタスクグラフに追加する．
- xmp_distribute_tasklet
  構築したタスクグラフを元に，タスクレットスケジュール情報を生成し，各ノードに配布する．各ノードは，配布されたスケジュール情報を元に，タスクレットを生成する．
- xmp_tasklet_wait_all
  マスタノードが，生成された全てのタスクレットの実行が完了するのを待つ．

5.3　スレッド/タスクレットの管理

各ノードにおけるスレッド/タスクレットの管理には，Argonne National Laboratory が開発しているユーザレベルスレッドライブラリ Argobots [7] を用いる．さらに，各ノードにおいてマルチスレッドによりタスクレットを実行

```
[C] #pragma xmp tasklets [ on {nodes-ref | template-ref} ] [ local ( variable [ , variable ]... ) ]
    structured-block
[F] !$xmp tasklets [ on {nodes-ref | template-ref} ] [ local ( variable [ , variable ]... ) ]
    structured-block
    !$xmp end tasklets
```

図 5　tasklets 構文（[C] と [F] は，それぞれベース言語 C と Fortran における仕様を表す）

```
[C] #pragma xmp tasklet [ onto {nodes-ref | template-ref} ] [ params ( variable [ , variable ]... ) ] [ depend-clause ]...
    structured-block
[F] !$xmp tasklet [ onto {nodes-ref | template-ref} ] [ params ( variable [ , variable ]... ) ] [ depend-clause ]...
    structured-block
    !$xmp end tasklet
depend-clause は以下のいずれか:

      in ( variable [ , variable ]... )
      out ( variable [ , variable ]... )
      inout ( variable [ , variable ]... )
```

図 6　tasklet 構文（[C] と [F] は，それぞれベース言語 C と Fortran における仕様を表す）

するために，MPI_THREAD_MULTIPLE に対応した MPI 実装を用いる．

6. 予備評価

ブロックコレスキー分解をターゲットとして，提案方式の予備的な評価を行った．

まず，5.1 節で述べたコンパイラによるソースコード変換を想定し，ターゲットコードを手動で変換した（図 7 および図 8）．本予備評価では，特にタスクレット本体の切り出しとデータ依存の制御について，次に示す方法でソースコード変換を行った．

- タスクレット本体
  ターゲットの処理であるコード部の前後に，データ受信部とデータ送信部を以下のように付加し，タスクレット本体とする．
  - データ受信部（図 8 の 11〜22 行目）
    MPI_Irecv により，他のタスクレットからデータを受信する．未完の受信がある場合，タスクレット・スイッチを行い，当該ノードに割り当てられている他のタスクレットを実行する（同図 17〜17 行目）．
  - コード部（同図 24〜25 行目）
    tasklet 構文で指定された本来の処理を実行する．
  - データ送信部（同図 27〜31 行目）
    MPI_Bsend により[*2]，他のタスクレットにデータを送信する．
- データ依存
  本予備評価におけるタスクレットは，in 節に指定されたデータを他のタスクレットからバッファへ受信し，コード部においてバッファ上のデータを操作し，out 節に指定されたデータを他のタスクレットへ送信するという定型的な動作を行う．すなわち，分散配列に直接的にはアクセスせず，バッファを介して間接的にアクセスする．しかし，tasklets ブロックにおいて，あるデータを最初に参照するタスクレットおよび最後に定義するタスクレットは，それぞれ分散配列の領域を直接的に参照または定義する必要がある．そこで，本予備評価では，ブロックコレスキー分解における各ブロックに関し，元となる分散配列の領域からデータを読み出して他のタスクレットへ供給する「プロローグ・タスクレット」と，他のタスクレットからデータを受け取って分散配列の領域へデータを書き戻す「エピローグ・タスクレット」を，それぞれ tasklets ブロックの先頭と末尾に設けることで対処した．これらのタスクレットを含めてデータ依存関係を構築することにより，分散配列へのアクセスは解決される．

次に，5.2 節で述べた 4 つのランタイムのプロトタイプを実装した．本予備評価では，xmp_tasklet_wait_all において，全てのタスクレットを実行するものとした．

手動で変換したソースコードと，ランタイムのプロトタイプ実装をコンパイル，リンクし，特定の問題サイズと実行環境において，得られたプログラムが正しく動作することを確認した．

[*2] ここで MPI_Bsend を用いるのは，送信処理を発行後，その完了を待つことなく当該タスクレットの実行を完了させるためである．

```
1  #pragma xmp tasklets on t[0]
2
3  for (int k = 0; k < nt; k++){
4   #pragma xmp tasklet onto t[k] \
5                        inout(A[k][k][:])
6   potrf(...);
7   for (int i = k + 1; i < nt; i++){
8    #pragma xmp tasklet onto t[i] \
9            in(A[k][k][:]) inout(A[k][i][:])
10   trsm(...);
11   for (int i = k + 1; i < nt; i++){
12    for (int j = k + 1; j < i; j++){
13     #pragma xmp tasklet onto t[i] \
14             in(A[k][i][:], A[k][j][:]) \
15             inout(A[j][i][:])
16     gemm(...);
17    }
18   }
19   #pragma xmp tasklet onto t[i] \
20           in(A[k][i][:]) inout(A[i][i][:])
21   syrk(...);
22  }
23
24 }
```

(a) 変換前

```
1  int tid0 = xmp_regist_tasklet(
2                  tasklet_potrf, "potrf");
3  ...
4
5  if (me == 0){
6
7   for (int k = 0; k < nt; k++){
8    xmp_create_tasklet(tid0, ...);
9    for (int i = k + 1; i < nt; i++){
10    xmp_create_tasklet(tid1, ...);
11    for (int i = k + 1; i < nt; i++){
12     for (int j = k + 1; j < i; j++){
13      xmp_create_tasklet(tid2, ...))
14     }
15    }
16    xmp_create_tasklet(tid3, ...))
17   }
18  }
19
20 }
21
22 xmp_distribute_tasklet();
23 xmp_tasklet_wait_all();
24 xmp_barrier();
```

(b) 変換後

図 7 提案方式によるブロックコレスキー分解の実装

7. 今後の課題

- タスクレット・スケジューラの改良
  本報告では，タスクレット間の依存関係は，タスクレット自体が発行する通信によって実現されている．その結果として，必要な全てのデータの受信が完了するまで，オーバヘッドが大きいタスクスイッチが繰り返されることになる．
  スレッド/タスクレット管理に用いた Argobots では，スレッド（タスク）スケジューラをユーザが定義できる．受信処理の完了をタスクレットのスケジュール条件と見なし，依存関係の制御（データの送受信）を，タスクレットではなくスケジューラの仕事とすることにより，性能向上を見込める．
- タスク間通信の高速化
  タスク間通信の実装には MPI の一対一通信（i.e. `MPI_Isend` および `MPI_Bsend`）の機能を用いたが，片側通信の機能または他の低レベル通信ライブラリ [8], [9], [10], [11] を用いることで性能向上を見込める．

8. 関連研究

OmpSs およびその後継である OmpSs-2 は，ノード内の共有メモリ並列処理をターゲットとする指示文ベースのプログラミングモデルである [12], [13], [14] ．分散メモリ環境に OmpSs を適用する場合に，MPI の一対一通信のみを含むタスクを記述することにより，ノードをまたぐデータ依存を指定する方法が提案されている．ノードをまたぐ依存をデータ通信により実現する点で本方式と近いが，分散配列上のデータ依存関係を指示文を用いて直接に記述できる点が本方式の利点である．

その他に，共有メモリ向けのタスク並列プログラミングモデルとしては，Thread Building Blocks (TBB)[15]，Cilk Plus [16]，Chapel [17] などが知られている．

9. おわりに

本報告では，分散配列上のデータ依存関係に基づく分散タスク生成方式の検討を行った．各ノードが分散的にタスクを生成する従来方式に対し，本方式では，特定のノード上のスレッドがタスクグラフを集中的に生成し，各ノードへ分配することにより，分散配列に関するデータ依存関係に基づいてタスクグラフを生成することが可能になる．本方式に基づくソースコード変換を想定して，ブロックコレスキー分解のコードを手動で変換するとともに，タスクレット処理のためのコンパイラ・ランタイムのプロトタイプを実装した．その結果，本方式による分散タスク生成が実現可能であることを確認できた．ただし，現時点では，条件により実行時エラーが発生する場合があるため，性能評価の結果については研究会の場で報告する予定である．

```
1  void tasklet_XXX(void **args){
2
3   // から各種情報を抽出args
4   int *params = (int *)args[0];
5   int *in_rank = (int *)args[1];
6   int *in_tag = (int *)args[2];
7   ...
8
9   double *T = (double *)malloc(...);
10
11  // データ受信部
12  if (in_tag[0] >= 0){
13   MPI_Irecv(T, ..., in_rank[0],
14             in_tag[0], ...);
15  }
16
17  // タスクレット・スイッチをしつつ受信の完了を待つ
18  while (nreqs){
19   MPI_Testall(nreqs, req, &flag, ...);
20   if (flag) break;
21   tasklet_yield();
22  }
23
24  // コード部
25  ...
26
27  // データ送信部
28  for (int i = 0; i < num_out; i++){
29   MPI_Bsend(T, ..., out_rank[i],
30             out_tag[i], ...);
31  }
32
33  free(T);
34
35 }
```

図 8 提案方式によるタスクレット本体のコード

謝辞 本報告の結果（の一部）は，理化学研究所のスーパーコンピュータ「京」（課題番号:ra000002）および筑波大学計算科学研究センターの学際共同利用プロジェクト（Cygnus）を利用して得られたものです.

参考文献

[1] OpenMP Architecture Review Board: OpenMP Application Program Interface Version 5.0, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (2018).

[2] XcalableMP Specification Working Group: XcalableMP Specification Version 1.4, http://xcalablemp.org/download/spec/xmp-spec-1.4.pdf (2018).

[3] Nakao, M., Lee, J., Boku, T. and Sato, M.: XcalableMP Implementation and Performance of NAS Parallel Benchmarks, *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS10)*, New York (2010).

[4] 李 珍泌，朴 泰祐，佐藤三久：分散メモリ向け並列言語 XcalableMP コンパイラの実装と性能評価，情報処理学会論文誌:コンピューティングシステム，Vol. 3, No. 3, pp. 153–165 (2010).

[5] Tsugane, K., Lee, J., Murai, H. and Sato, M.: Multi-tasking Execution in PGAS Language XcalableMP and Communication Optimization on Many-core Clusters, *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, New York, NY, USA, ACM, pp. 75–85 (online), DOI: 10.1145/3149457.3154482 (2018).

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

[7] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., Iwasaki, S., Jindal, P., Kalé, L. V., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K. and Beckman, P.: Argobots: A Lightweight Low-Level Threading and Tasking Framework, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 29, No. 3, pp. 512–526 (online), DOI: 10.1109/TPDS.2017.2766062 (2018).

[8] Bonachea, D. and Hargrove, P. H.: GASNet-EX: A High-Performance, Portable Communication Library for Exascale, (online), DOI: 10.25344/S4QP4W.

[9] Daily, J., Vishnu, A., Palmer, B., van Dam, H. and Kerbyson, D.: On the suitability of MPI as a PGAS runtime, *2014 21st International Conference on High Performance Computing (HiPC)*, pp. 1–10 (online), DOI: 10.1109/HiPC.2014.7116712 (2014).

[10] Breitbart, J., Schmidtobreick, M. and Heuveline, V.: Evaluation of the Global Address Space Programming Interface (GASPI), *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pp. 717–726 (online), DOI: 10.1109/IPDPSW.2014.83 (2014).

[11] 小田嶋哲哉，森江善之，李 珍泌，佐藤三久：マルチタスク PGAS モデルをサポートするノード間軽量通信レイヤの検討，研究報告ハイパフォーマンス・コンピューティング（HPC），Vol. 2019-HPC-168, No. 1, pp. 1–7 (2019).

[12] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X. and Planas, J.: Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures., *Parallel Processing Letters*, Vol. 21, pp. 173–193 (online), DOI: 10.1142/S0129626411000151 (2011).

[13] Fernández, A., Beltran, V., Martorell, X., Badia, R. M., Ayguadé, E. and Labarta, J.: Task-Based Programming with OmpSs and Its Application, *Euro-Par 2014: Parallel Processing Workshops*, Cham, Springer International Publishing, pp. 601–612 (2014).

[14] BSC Programming Models: OmpSs-2 Specification, https://pm.bsc.es/ftp/ompss-2/doc/spec/ (2018).

[15] Reinders, J.: *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*, O'Reilly (2007).

[16] Intel Corporation: Intel®Cilk™Plus Language Extension Specification Version 1.2, https://www.cilkplus.org (2013).

[17] Cray Inc: Chapel Language Specification 0.93, http://chapel.cray.com/spec/spec-0.93.pdf (2013).