Vol.2013-HPC-140 No.8
2013/7/31

並列プログラミング言語 XcalableMP におけるステンシル通信の効率的な実装

村井 均[1]　佐藤 三久[2]

概要：規模と複雑さを増す並列計算機をプログラムする手段として、Partitioned Global Address Space (PGAS) に基づく並列プログラミング言語が近年注目されている。このような PGAS 言語においても、規則的なステンシルコードは依然として重要な ターゲットの一つである。我々は、PGAS 並列プログラミング言語 XcalableMP のコンパイラに、次の 3 つの方法でステンシル通信を実装した。1) 派生データ型メッセージに基づく実装、2) 特にマルチコア環境において有効な、パック/アンパックに基づく実装、3) スーパーコンピュータ「京」の RDMA 機能による実験的な実装。スーパーコンピュータ「京」においてこれらを評価した結果、第一と第二の実装はそれぞれ異なる条件において有効であるため、実行時にそれらを使い分けることが実用的であること、第三の実装は有望であるが同期の手順に課題が残ることがわかった。

1. はじめに

より高い性能を求めて計算機システムの規模と複雑さ (e.g. メモリ階層、ネットワークトポロジ) が増す中、高性能と高生産性の両方をユーザにもたらすプログラミング手段が強く求められている。XcalableMP (XMP) [1], Fortran 2008 の coarray 機能 [2], Unified Parallel C (UPC) [3], Chapel [4], X10 [5] のような、Partitioned Global Address Space (PGAS) に基づく並列プログラミング言語は、その要求を満たすものと考えられており、現在活発に研究されている。

多くの PGAS 言語は、祖先である High Performance Fortran (HPF) [6] ではうまく扱えなかった、タスク並列プログラムに代表されるより広いアプリケーションをサポートすることを大きな目標として掲げるが、規則的なステンシルコード（[7], [8], [9]）が最重要のターゲットの一つであることは変わらず、ステンシル通信を効率的に扱うことは依然として重要である。

我々は、開発中の Omni XcalableMP コンパイラに、3 つの方法でステンシル通信を実装した。本報告では、それらの詳細を説明する。第一は、Message Passing Interface (MPI) [10] の派生データ型に基づく実装である。第二は、バッファのパック/アンパックに基づく実装であり、並列に実行できることからマルチコア環境において有効であると考えられる。第三は、スーパーコンピュータ「京」（以下、京コンピュータ）[11] の拡張 RDMA インタフェース [12] に基づく実験的な実装である。本研究は、PGAS 言語におけるステンシル通信の最適な実装方法を探ることを目的としている。

以下、2 章と 3 章でそれぞれ XMP 言語仕様と Omni XMP を概観した後、4 章と 5 章で提案するステンシル通信の実装を説明する。続いて、6 章ではそれらの評価について述べる。最後に、7 章で関連研究に触れた後、8 章で本報告を総括する。

2. XcalableMP

PGAS 並列プログラミング言語 XcalableMP (XMP) は、PC クラスタコンソーシアム XcalableMP 規格部会が提案している Fortran および C に対する指示文ベースの言語拡張である。XMP のグローバルビューモデルは、データ/タスク並列処理に基づく典型的な並列化をサポートしており、元の逐次コードにわずかな変更を加えるだけで並列化を実現できる。また、XMP のローカルビューモデルでは、Fortran 2008 から導入した coarray 機能を利用し、より柔軟な並列処理を記述できる。さらに、マルチコア環境における OpenMP 指示文と XMP の共存についても、次版の仕様で取り込まれる予定である。本章では、XMP 仕様の概要を説明する。

典型的な XMP プログラムの例が、**図 7** に示されている。

[1] 理研
RIKEN
[2] 筑波大学
University of Tsukuba

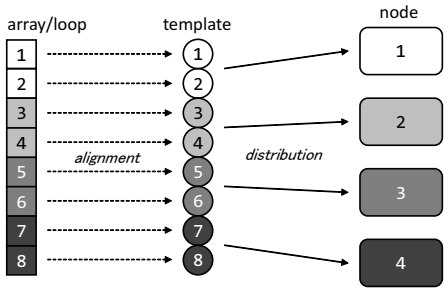

図 1 XMP におけるデータマッピングとワークマッピング

2.1 実行モデルとメモリモデル

2.1.1 実行モデル

XMP の実行モデルは、SPMD (Single Program Multiple Data) に従う。すなわち、XMP における処理の主体である XMP ノード (通常、MPI プロセスに対応する) は、同一のメインルーチンから実行を開始し、同一のコードを独立に (非同期的に) 実行する。一方、各 XMP 指示文は「グローバル」であり、全ノードによって集団的に実行される。

2.1.2 メモリモデル

各ノードが直接にアクセスできるのは、自身のローカルなメモリに存在するデータのみである。リモードノード上のデータにアクセスする必要がある場合、次章で述べる `reflect` のような XMP 指示文または coarray を用いて明示的にノード間通信を記述する必要がある。

2.2 データマッピングとワークマッピング

2.2.1 データマッピング

まず、`align` 指示文により、配列は、仮想的な配列である**テンプレート**に**整列**する。次に、テンプレートは、`distribute` 指示文により、指定した形式 (ブロック、サイクリック、ブロック・サイクリック、不均等ブロック) でノード集合に**分散**される。その結果、配列の各要素は、分散されたテンプレートを介して、一つ以上のノードに割り当てられる (**図 1**)。分散された配列のローカルな部分 (論理的には一つの矩形領域を構成する) は、各ノードのローカルメモリ上に割付けられる。

2.2.2 ワークマッピング

配列と同様に、ループネストの繰り返し空間はテンプレートに「整列」される。整列したループネストは、実行ノードによって並列に実行される。

2.3 ステンシル通信のための指示文

2.3.1 `shadow` 指示文

ブロック分散または不均等ブロック分散を指定された配列は、**シャドウ**と呼ばれる付加的な領域を持つことができる。シャドウは、隣接するノード間で分散境界上のデータを交換する通信（ステンシル通信）のためのバッファとして用いられる。

図 2 (a) に `shadow` 指示文の文法を示す。`shadow` 指示文は、配列の各次元に付加するシャドウの幅を宣言する [*1]。ここで、配列のある次元において、上側シャドウと下側シャドウの幅は異なっていてもよい。

2.3.2 `reflect` 指示文

図 2 (b) に `reflect` 指示文の文法を示す。`reflect` 指示文は、配列のシャドウ領域を、対応する反映元の値で更新することを指示する。

`width` 節を付加すれば、シャドウ領域の一部だけを更新することもできる。さらに、`width` 節の中で`/periodic/` 修飾子を指定することにより、シャドウ領域を「周期的に」更新できる（グローバルな下限（上限）側のシャドウを、グローバルな上限（下限）の要素の値で更新する)。

また、`reflect` 指示文が `async` 節を伴う場合、関連する通信は**非同期的**に実行される。そのような非同期的な通信は、MPI 標準のノンブロッキング通信と同じく、後続する計算とオーバラップできる可能性がある。

2.3.3 `wait_async` 指示文

`wait_async` 指示文 (図 2 (c)) は、*async-id* と関連付けられた全ての非同期通信が完了するまでブロックし、後続する文の実行を抑止する。なお、`reflect` 以外の通信も非同期通信として指定することが可能であり、`wait_async` はそれらも扱うことに注意されたい。

3. Omni XcalableMP

Omni XcalableMP は、筑波大学 HPCS 研究室と理研 AICS プログラミング環境研究チームにおいてオープンソースプロジェクトとして開発されている、XMP コンパイラのリファレンス実装である [13]。

Omni XMP は、トランスレータとランタイムライブラリの 2 つに大きく分かれる。トランスレータは、XMP ソースプログラムを、ランタイムルーチン呼び出しを含むベース言語のプログラムへ変換する。特に、ソースプログラム中に現れる `reflect` や `wait_async` のような**実行指示文**は、一連のランタイムルーチン呼び出しへ置換される。ランタイムライブラリは、実行時の各種処理 (実行制御、通信および同期、メモリ管理など) を行う。

現在の実装では、移植性の観点から、ランタイムライブラリは MPI に基づいているが、京コンピュータにおける拡張 RDMA インタフェース (5 章) や GASNet [14] のような他の通信ライブラリに基づく実装も開発中または検討中である。

Omni XMP は、Linux クラスタ、Cray マシン, 京コンピュータの他、MPI が動作する任意のプラットフォームをサポートしている。

[*1] *shadow-width* として “*” が指定されている場合、配列の全領域がシャドウと見なされる。この機能は「フルシャドウ」と呼ばれるが、本稿では扱わない。

```
[F]  !$xmp shadow array-name ( shadow-width [, shadow-width]... )
[C]  #pragma xmp shadow array-name [shadow-width][shadow-width]...
```

shadow-width は、以下のいずれかでなければならない。

int-expr
int-expr : *int-expr*
*

(a) shadow 指示文

```
[F]  !$xmp reflect ( array-name [, array-name]... ) [width ( reflect-width [, reflect-width]... )] [async ( async-id )]
[C]  #pragma xmp reflect ( array-name [, array-name]... ) [width ( reflect-width [, reflect-width]... )] [async ( async-id )]
```

reflect-width は、以下のいずれかでなければならない。

[/periodic/] int-expr
[/periodic/] int-expr : *int-expr*

(b) reflect 指示文

```
[F]  !$xmp wait_async ( async-id [, async-id ]... ) [on nodes-ref | template-ref]
[C]  #pragma xmp wait_async ( async-id [, async-id ]... ) [on nodes-ref | template-ref]
```

(c) wait_async 指示文

図 2 ステンシル通信のための XMP 指示文の文法（行頭の [F] は XMP/Fortran の、[C] は XMP/C の仕様であることを示す）

4. 実装

我々は、一般の (MPI をサポートする) プラットフォームのために、reflect 通信を 2 つの方法で実装した。すなわち、MPI の派生データ型に基づく実装と、バッファのパック/アンパックに基づく実装である。

Omni XMP ランタイムは、その 2 つのいずれを用いるかを自動的に判断する。さらに、環境変数により明示的に指示することも可能である。

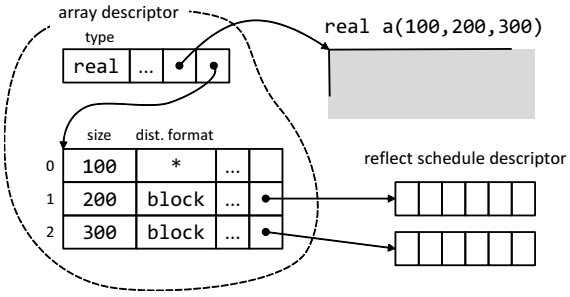

図 3 Omni XMP におけるディスクリプタ

4.1 基本設計

一般に、reflect を含む Omni XMP ランタイムルーチンは「汎用」であり、任意の配列形状、分散および指示節の組合わせに対応できなければならない。汎用性を保ちつつ、開発の規模を抑える一つの手段として、対象の配列の複数の次元のシャドウを更新する「多次元 reflect」は、次元ごとの reflect の繰り返しとして実現する。

Omni XMP のランタイムは、各々の分散配列に対しディスクリプタを管理する。このディスクリプタは、実行時にランタイムルーチンによって、必要に応じて参照される。その有効期間は、対応する配列と同じである。さらに、配列がシャドウ領域を持つ場合、ランタイムは、Reflect Schedule Descriptor (RSD) と呼ばれるデータ構造を生成する。RSD には reflect 通信のスケジュール [*2] が格納され、当該配列のディスクリプタからリンクされる。生成された RSD は、指示節の指定が異なる別の reflect 指示文が実行され、通信スケジュールが変更されるまで繰り返し再利用される。**表 1** は、RSD の内容を示す.

4.2 実装 1: 派生データ型

任意の reflect 通信は、長さ 1 のベクトル型メッセージの一対一ノンブロッキング通信として実行できる。ここで、「ベクトル」は、等間隔に並ぶ複数のブロックから成る、MPI の組込み派生データ型の一つを指す。ベクトル型は、MPI_TYPE_VECTOR 関数によって作られる。

ベクトル型は、3 つのパラメータを持つ。ブロックの数を表す *count*、各ブロックに含まれる要素の数を表す *blocklength*、ブロックの間隔を表す *stride* である。

[*2] ある通信を実行するのに必要な情報。MPI では、通信関数の引数並び、ないしはそれと等価な情報。

表 1 Reflect Schedule Descriptor の内容

型	メンバ名	説明
int	lo_width hi_width	最近の更新幅
int	is_periodic	最近の周期的フラグ
MPI_Datatype	datatype_lo datatype_hi	ベクトルデータ型
MPI_Request	req[4]	上側/下側シャドウの送信/受信の通信要求ハンドル
void*	lo_send_buf lo_recv_buf	下側シャドウのためのバッファ
void*	hi_send_buf hi_recv_buf	上側シャドウのためのバッファ
void*	lo_send_array lo_recv_array	上側シャドウのための配列内の送受信位置
void*	hi_send_array hi_recv_array	下側シャドウのための配列内の送受信位置
int	count blocklength stride	ベクトルのパラメータ(パック/アンパックで使用)
int	lo_rank hi_rank	隣接ノードの MPI ランク

N 次元配列の k 次元目に対する reflect のためのベクトル型の *count*, *blocklength* および *stride* は次のように求められる [*3]。

$$
\begin{aligned}
count &= lsize_{k+1} \times \cdots \times lsize_{N-1} \\
blocklength &= shadow_k \times lsize_0 \times \cdots \times lsize_{k-1} \\
stride &= lsize_0 \times \cdots \times lsize_k
\end{aligned}
$$

ここで、$lsize_i$ と $shadow_i$ は、それぞれ、配列の i 次元目のローカルサイズ (各ノードに割り当てられている要素の数) とシャドウ幅を表す。ローカルサイズはシャドウ幅を含むことに注意されたい。

ベクトルに対するノンブロッキング通信のスケジュールは、**持続的な通信要求**（*Persistent Communication Request*）の形で、RSD に保存される。保存されたスケジュールは、MPI_Startall と MPI_Waitall によって、それぞれ持続的通信の開始および完了に用いられる (**図 4**)。

通信スケジュールは配列の次元毎に生成されるが、現在の実装では、全次元に対する持続的通信がまとめて非同期的に開始される。したがって、配列の「頂点」の位置のシャドウは正しく更新されない可能性があることから、9 点差分を扱うことができない。この問題は、各次元に対する持続的通信を順に同期的に行うことにより解決できる。ただし、4.4 章で述べる非同期 reflect ではその解決策は使えないため、正しくシャドウを更新するために「斜めに隣接する」ノード間の通信を実装する必要がある。次章以降で述べるパック/アンパックによる実装および RDMA による実装でも、同様の問題が存在する。

[*3] これは、Fortran 式の列優先の要素並びにおける計算法である。C に対する計算法も容易に得られる。

```
1  // 派生データ型を生成
2  for (i = 0; i < ndims; i++){
3   MPI_Type_vector(count, blocklength*lwidth,
4           stride, MPI_BYTE, &reflect->dt_lo);
5   MPI_Type_commit(&reflect->dt_lo);
6   ...
7  }
8
9  // 持続的通信を初期化
10 for (i = 0; i < ndims; i++){
11  MPI_Recv_init(rbuf_lo, 1, reflect->dt_lo,
12           src, tag, comm, &reflect->req[0]);
13  ...
14  MPI_Send_init(sbuf_hi, 1, reflect->dt_hi,
15           dst, tag, comm, &reflect->req[3]);
16 }
17
18 // 持続的通信を実行
19 MPI_Startall(4*ndims, reflect->req);
20 MPI_Waitall(4*ndims, reflect->req, status);
```

図 4 派生データ型による実装の概要

4.3 実装 2: パック/アンパック

MPI ライブラリの内部でベクトル型メッセージの通信がどのように処理されるかは実装依存である。ある実装は、送信前にベクトルを一つの連続なバッファへパックするかもしれないが、別の実装はパックをせずにベクトルの各ブロックを一つずつ送信するかもしれない。したがって、一般には、ベクトルの送受信時の内部的な前処理および後処理は十分には最適化されず、マルチコア環境であってもマルチスレッド化されないと考えるべきである。なお、一般のデータ型の場合とは異なり、ベクトルのパック/アンパック処理は理論的には並列化可能である。

主にマルチコア環境においてより高い性能を達成するため、reflect 通信のためのベクトルのパック/アンパック処理をマルチスレッド化した（**図 5**）。本処理を行うルーチンは、OpenMP 指示文によってマルチスレッド化されている。XMP 言語仕様によると、各 XMP 指示文 (したがって、それに対応するランタイムライブラリルーチン) は、シングルスレッドであることに注意されたい。

ただし、このような並列化が有効なのは、ノード内に一つ以上のプロセッサコアが存在している場合のみである。したがって、Omni XMP のランタイムは、OpenMP API のランタイムライブラリルーチン omp_get_num_procs（プログラムが利用可能なプロセッサコアの個数を返す）を用い、パック/アンパック処理を並列に実行するべきか否かを判断する。

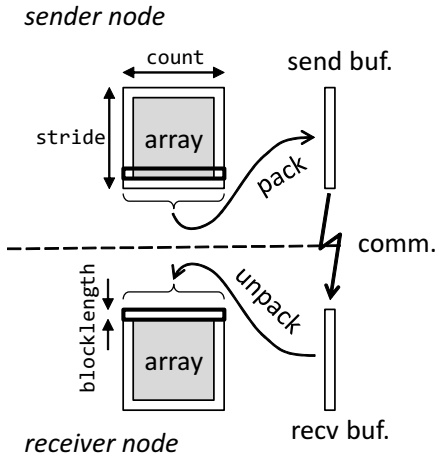

図 5 `reflect` におけるベクトルのパック/アンパック

図 6 は、Omni XMP の内部パッキングルーチン `_XMPF_pack_vector` である。図より、利用可能なプロセッサコアが 2 個以上かつパック/アンパック処理の量が十分に大きい場合のみ当該ループが並列化されることがわかる。

パック/アンパック処理のために用いられる通信バッファはランタイムシステムによって管理される。現在の実装では、配列のある次元に対して割り当てられたバッファは、その配列が生存している間は存続し繰り返し利用される。

なお、前述のように、任意の XMP プログラムは、マルチスレッド化されたパック/アンパック処理を呼び出す可能性がある。したがって、XMP プログラムは、常に OpenMP API ランタイムルーチンとリンクされることになるが、実行時のオーバヘッドは発生しない。

4.4 非同期通信

2.3.2 章で述べたように、`reflect` 指示文が `async` 節を伴う場合、`reflect` 通信は非同期的に実行される。

Omni XMP ランタイムは、ノンブロッキング通信に対応するリクエストハンドルによって、このような非同期的な `reflect` を管理する。Omni XMP における非同期 `reflect` は、実行時に次のように処理される。

(1) `reflect` 指示文の位置において、一連のノンブロッキング通信が発行され、それらの通信要求ハンドルは *Asynchronous Communication Table (ACT)* に格納される。ACT は、*async-id* をキーとするハッシュテーブルである。

(2) ノンブロッキング通信が行われる。このとき、計算とオーバラップされる可能性がある。

(3) `wait_async` 指示文の位置において、指定された *async-id* をキーとして ACT を検索し、対応する通信要求ハンドルを得る。得られた通信要求ハンドルに対して `MPI_Waitall` を発行し、ノンブロッキング通信を完了させる。

`wait_async` 指示文は、`reflect` 以外の非同期通信に対しても用いられる。したがって、上述の仕組みはそれらにも適用できるよう設計されている。

現在の実装では、非同期 `reflect` は、4.2 章で示した派生データ型よる実装に基づく。なぜなら、(パック/アンパック処理を省略して) できるだけ早くノンブロッキング通信を発行し、できるだけ多くの通信を後続する計算とオーバラップさせる方が、性能上有利であると考えられるためである。

表 2 拡張 RDMA インタフェースの関数

名称	機能概要
FJMPI_Rdma_init	拡張 RDMA インタフェースの初期化処理
FJMPI_Rdma_finalize	拡張 RDMA インタフェースの終了処理
FJMPI_Rdma_reg_mem	メモリ登録
FJMPI_Rdma_dereg_mem	メモリ登録解除
FJMPI_Rdma_get_remote_addr	リモート DMA アドレスの取得
FJMPI_Rdma_put	RDMA WRITE
FJMPI_Rdma_get	RDMA READ
FJMPI_Rdma_poll_cq	RDMA の完了確認

5. RDMA に基づく実装

本章では、京コンピュータの拡張 RDMA インタフェースに基づく、`reflect` 通信の実験的実装を示す。

5.1 拡張 RDMA インタフェース

京コンピュータおよび富士通 PRIMEHPC FX10 の MPI ライブラリでは、**拡張 RDMA インタフェース**と呼ばれる機能を利用できる。本機能を用いれば、ネットワークインタフェースコントローラ (NIC) などのハードウェア資源を最大限に活用してノード間通信を行うことができる。**表 2** は、本機能に関する関数のリストである。

本機能の RDMA WRITE を用いて `reflect` 通信を実装するとき、以下の各項目を考慮する必要がある。

- 本機能を通じてアクセスされるのに先立ち、`FJMPI_Rdma_reg_mem` 関数を用いて配列を「登録」し、メモリ ID と関連付けておく必要がある。現在の実装では、シャドウを持つ全ての分散配列は、本機能を通じてアクセスされる可能性があるものとして登録の対象になる。
- 対象の配列は、MPI_COMM_WORLD に対応する全体ノード集合に分散されていなければならない。これは、RDMA の相手プロセスは、MPI_COMM_WORLD におけるランクによって識別されるためである。
- RDMA WRITE の発行に先立ち、各ノードは、隣接ノードのシャドウ領域を更新可能か否かを明示的に確

```
1  void _XMPF_pack_vector(char * restrict dst, char * restrict src,
2                         int count, int blocklength, int stride){
3
4    if (_xmp_omp_num_procs > 1 && count * blocklength > THRESHOLD){
5  #pragma omp parallel for
6      for (int i = 0; i < count; i++){
7        memcpy(dst + i * blocklength, src + i * stride, blocklength);
8      }
9    }
10   else {
11     for (int i = 0; i < count; i++){
12       memcpy(dst + i * blocklength, src + i * stride, blocklength);
13     }
14   }
15
16 }
```

図 6 パッキングルーチン

認する必要がある。したがって、reflect の直前に何らかの同期処理が必要になる。

- 各ノードは、FJMPI_Rdma_poll_cq 関数を用いて NIC をポーリングすることにより、自身が発行した RDMA WRITE の完了を確認できる。集団的な処理としての reflect 通信の完了（全ノードにおいてシャドウ領域の更新が完了）を保証するために、何らかの同期処理が必要になる。
- 各 RDMA には 0 から 14 の任意の整数値を「タグ」として割り当てることができる。現在の実装では async-id をタグとして用いるので、その値は 0 から 14 に制限される。

3 番目と 4 番目の項目は、reflect 通信が集団的であることに起因する。

5.2 実装 3: RDMA

5.2.1 通常モード

拡張 RDMA インタフェースに基づく通常モードの（非同期的でない)reflect 通信は、次の手順で実行される。

(1) 全ノードが到着するまで待ち合わせる (バリア同期)。
(2) ベクトルの各ブロックに対して RDMA WRITE を発行する。
(3) 自身が発行したすべての RDMA WRITE が完了するまで NIC をポーリングする。
(4) 全ノードが到達するまで待ち合わせる (バリア同期)。

一つ目のバリア同期は、リモート側 (隣接ノード) の準備ができている（シャドウ領域の更新が可能である）ことを、二つ目は、自ノードと隣接ノードの両方で reflect に関するすべての処理が完了したことを、それぞれ保証するためのものである。

なお、パック/アンパックを行わないのは、RDMA WRITE のレイテンシは小さく、複数回の RDMA WRITE を発行するオーバヘッドの方が、パック/アンパック処理のそれよりも小さいためである。

5.2.2 非同期モード

非同期 reflect では、上述のステップ 1 および 2 は reflect の位置で、3 および 4 は wait_async の位置で、それぞれ実行する。ただし、以下の点が通常モードとは異なる。reflect では，RDMA WRITE を発行する際に、その async-id をタグとして設定するとともに、発行した RDMA の個数を ACT に保存しておく。wait_async では、ACT から抽出した個数の RDMA が完了するまで NIC をポーリングする。

6. 評価

気候モデル SCALE-LES [7] の力学コアプロトタイプを XMP で並列化し、京コンピュータ [11] で実行することにより、本報告で提案する reflect の各実装の性能を評価した。なお、SCALE-LES は、Fortran で書かれた典型的なステンシルコードである (図 7)。評価に用いた言語環境は K-1.2.0-13 である。水平方向の格子点数は 512x512，鉛直方向の格子点数は 128 であり，時間発展ループ 500 回転に要する時間を評価した.

計算ノード内のスレッド並列処理では 8 スレッドを用いる．すなわち，XMP の 1 ノードを、京コンピュータの 1 台の計算ノードに割り当て，自動スレッド並列化によって計算ノード内の 8 コアでスレッド並列処理を行う.

評価結果を、**図 8**, **図 9**, **図 10** に示す。各グラフの縦軸はシングルノード実行に対する速度向上を示す。なお、各実装における演算そのものの時間はほぼ同等であり、実行

```
1  !$xmp nodes p(N1,N2)
2  !$xmp template t(IA,JA)
3  !$xmp distribute t(block,block) onto p
4    ...
5    real(8) :: dens(0:KA,IA,JA)
6    ...
7  !$xmp align (*,i,j) &
8  !$xmp&   with t(i,j) :: dens, ...
9  !$xmp shadow (0,2,2) :: dens, ...
10   ...
11 !$xmp reflect (dens, ...) width &
12 !$xmp&     (0,/periodic/2,/periodic/2)
13   ...
14 !$xmp loop (ix,jy) on t(ix,jy)
15         do jy = JS, JE
16          do ix = IS, IE
17           ...
18           do kz = KS+2, KE-2
19            ... dens(kz,ix+1,jy) + ...
20            ...
21           end do
22           ...
23          end do
24         end do
```

図 7 評価対象の気候モデルコード (一部)

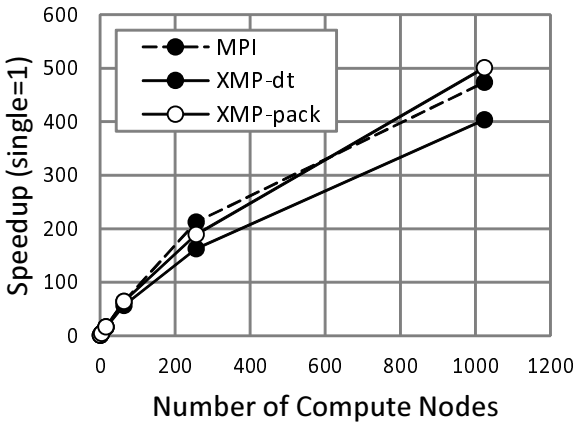

図 8 通常 `reflect` の評価結果

時間の差は通信時間に起因している。

図 8 は、通常モードの `reflect` の性能を示す。図中の “MPI” は手で作成した MPI 版を、“XMP-dt” は派生データ型による実装を、“XMP-pack” はパック/アンパックによる実装を示す。パック/アンパックによる実装は MPI 版に匹敵する性能であり、1024 ノード実行においては MPI 版を上回っている。ただし、この結果は、SPARC64 VIIIfx [15] の高速なハードウェアバリア機能に依存するところが大きい。実際、通常の Linux クラスタ上では、パック/アンパックによる実装は、京コンピュータほど効果的でないことが観察されている。一方、派生データ型による実装は

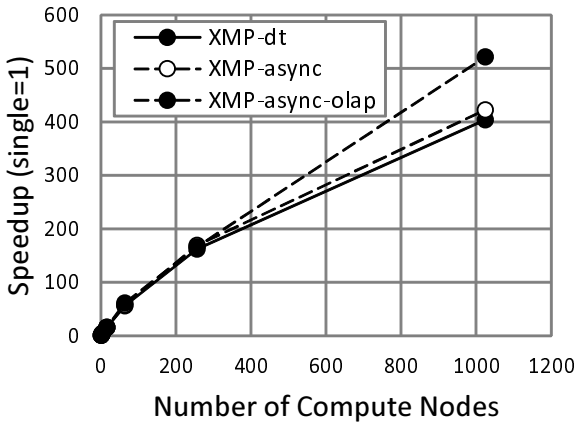

図 9 非同期 `reflect` の評価結果

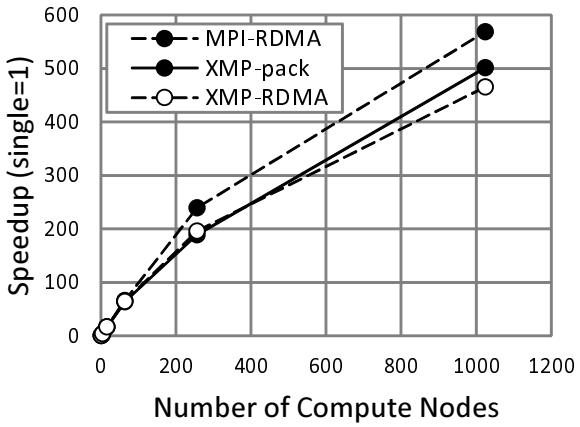

図 10 RDMA に基づく `reflect` の評価結果

MPI 版よりも遅いという結果である。しかしながら、フラット並列環境では、派生データ型による実装が他の 2 つよりも速いことが確認できている。

図 9 は、非同期モードの `reflect` の性能を示す。図中の “XMP-dt” は通常モードの派生データ型による実装 (比較用) を、“XMP-async” は非同期モードの `reflect`(通信と計算のオーバラップなし) を、“XMP-async-olap” は非同期モードの `reflect`(最大限に通信と計算をオーバラップさせたもの) を示す。グラフより、ACT の管理や検索などの、非同期通信のために導入されたオーバヘッドはそれほど大きくなく、特に 1024 ノード実行では、通信と計算のオーバラップによって大きく性能が向上していることがわかる。

図 10 は、RDMA に基づく `reflect` の性能を示す。図中の “MPI-RDMA” は手で作成した RDMA 版を、“XMP-pack” はパック/アンパックによる実装 (RDMA は利用していない、比較用) を、“XMP-RDMA” は本稿で提案した RDMA に基づく実装を示す。RDMA に基づく実装は、手で作成した RDMA 版およびパック/アンパックによる方法よりも遅いという結果である。これは、ステンシル通信を正しく実行数するという観点からすると、RDMA WRITE

の発行前および完了後のバリア同期の強度は大き過ぎるためである。実際、手で開発したRDMAコードでは、全ノードによるバリア同期ではなく、隣接ノード間の一対一の同期が使われている。RDMAに基づく実装において、同期の強度を低減することは将来の課題の一つである。

7. 関連研究

XMPの`reflect`指示文およびその非同期モードは、HPF拡張仕様であるHPF/JA [16], [17]によって最初に導入されたものである。シャドウの部分的更新は、それぞれNECのSXシリーズと地球シミュレータ用のHPFコンパイラであるHPF/SX V2 [18]とHPF/ES [19]が最初にサポートし、後にはライス大のdHPFコンパイラ[20]もサポートした。周期的なステンシル通信の機能は、HPFおよびその拡張の仕様には含まれておらず、我々の知る限り、明示的な指示文としてサポートしている言語仕様または処理系はこれまでない。

HPFにおけるステンシル通信の最適化について、蒲池らは「再整列」に基づく通信生成の方法を提案し、多次元ステンシル通信のコンパイル時最適化の方法を示した[21]。

文献[22], [23]では、メッシュベースの規則的なアプリケーションを、Co-Array FortranまたはFortran 2008標準の片側通信機能であるcoarrayを用いて実装し、メモリレイアウトや通信バッファの利用などの観点からMPIによる実装と比較している。その結果、coarrayに基づくステンシル通信は、そのようなアプリケーションにおいて有効であり、MPIに基づく実装を性能において上回る場合もあると述べている。

8. おわりに

Omni XMPコンパイラにおいて、3つの方法でステンシル通信を実装した。派生データ型メッセージに基づく第一の実装は、シンプルかつプラットフォームを選ばないという利点を持ち、下位に位置するMPIライブラリの実装によっては効率的に機能し得る。パック/アンパックに基づく第二の実装は、マルチコア環境においてマルチスレッド化できるという利点を持つ。第三の、京コンピュータの拡張RDMAインタフェースに基づく実験的な方法は、先の2つよりも高い性能を発揮できる可能性を持つが、現在のところ、同期の強度が大き過ぎることから第二の実装をわずかに上回る性能に留まっている。

今後の課題としては、以下が挙げられる。

- 9点差分における「斜めに隣接する」ノードから/への`reflect`通信を正しく扱うこと。
- パック/アンパックによる実装において、パック/アンパックを並列化する閾値を適切に設定すること。
- 同期の強度を低減することにより、RDMAに基づく実装の性能を改善すること。
- MPI-3の片側通信を用いて、より高い移植性と性能を備える実装を開発すること。

謝辞

本論文の結果(の一部)は、理化学研究所のスーパーコンピュータ「京」を利用して得られたものです。評価で用いた気候モデルコードSCALE-LESとそのRDMAベース実装は、理研AICSのチームSCALEによって開発されたものです。

参考文献

[1] XcalableMP Specification Working Group: XcalableMP Specification Version 1.1, `http://www.xcalablemp.org/xmp-spec-1.1.pdf` (2012).
[2] Numrich, R. W. and Reid, J.: Co-arrays in the next Fortran Standard, *ACM Fortran Forum*, Vol. 24, No. 2, pp. 4–17 (2005).
[3] UPC Consortium: UPC Specifications, v1.2, Technical report, Lawrence Berkeley National Lab (LBNL-59208) (2005).
[4] Cray Inc: Chapel Language Specification 0.93, `http://chapel.cray.com/spec/spec-0.93.pdf` (2013).
[5] Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V. and Sarkar, V.: X10: An Object–oriented Approach to Non–Uniform Cluster Computing, *Proc. OOPSLA 05* (2005).
[6] Kennedy, K., Koelbel, C. and Zima, H.: The Rise and Fall of High Performance Fortran: An Historical Object Lesson, *Proc. 3rd ACM SIGPLAN History of Programming Languages Conf. (HOPL-III)*, San Diego, California, pp. 7–1–7–22 (2007).
[7] 西澤誠也，佐藤陽祐，八代　尚，宮本佳明，吉田龍二，富田浩文，SCALE, T.：高領域・高解像度実験のための気象LESモデルの開発，ながれ，Vol. 32, pp. 149–152 (2013).
[8] Furumura, Takashi and Chen, Li: Parallel simulation of strong ground motions during recent and historical damaging earthquakes in Tokyo, Japan, *Parallel Computing*, Vol. 31, No. 2, pp. 149–165 (2005).
[9] Collins, William D and Bitz, Cecilia M and Blackmon, Maurice L and Bonan, Gordon B and Bretherton, Christopher S and Carton, James A and Chang, Ping and Doney, Scott C and Hack, James J and Henderson, Thomas B and others: The Community Climate System Model Version 3 (CCSM3), *J. Climate*, Vol. 19, No. 11, pp. 2122–2143 (2006).
[10] Message Passing Interface Forum: MPI: A Message Passing Interface Standard Version 3.0, `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf` (2012).
[11] Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M. and Watanabe, T.: Overview of the K computer, *FUJITSU Sci. Tech. J.*, Vol. 48, No. 3, pp. 255–265 (2012).
[12] FUJITSU LIMITED: *Parallelnavi Technical Computing Language MPI User's Guide* (2013).
[13] : Omni XcalableMP Compiler, `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/`.
[14] Bonachea, D.: GASNet specification, Technical report, University of California, Berkeley (CSD-02-1207) (2002).
[15] Yoshida, T., Hondo, M. and Sugizaki, R. K. G.: SPARC64 VIIIfx: CPU for the K computer, *FUJITSU*

Sci. Tech. J., Vol. 48, No. 3, pp. 274–279 (2012).

[16] Japan Association of High Performance Fortran: HPF/JA Language Specification, `http://www.hpfpc.org/jahpf/spec/hpfja-v10-eng.pdf` (1999).

[17] Seo, Y., Iwashita, H., Ohta, H. and Sakagami, H.: HPF/JA: extensions of High Performance Fortran for accelerating real-world applications, *Concurrency and Computation — Practice & Experience*, Vol. 14, No. 8–9, pp. 555–573 (2002).

[18] Murai, H., Araki, T., Hayashi, Y., Suehiro, K. and Seo, Y.: Implementation and Evaluation of HPF/SX V2, *Concurrency and Computation — Practice & Experience*, Vol. 14, No. 8–9, pp. 603–629 (2002).

[19] Yanagawa, T. and Suehiro, K.: Software System of the Earth Simulator, *Parallel Computing*, Vol. 30, No. 12, pp. 1315–1327 (2004).

[20] Chavarria-Miranda, D. and Mellor-Crummey, J.: An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications, *J. Instruction Level Parallelism*, Vol. 5 (2003).

[21] Kamachi, T., Kusano, K., Suehiro, K. and Seo, Y.: Generating Realignment-Based Communication for HPF Programs, *Proc. IPPS*, pp. 364–371 (1996).

[22] Barrett, R.: Co-array Fortran Experiences with Finite Differencing Methods, *The 48th Cray User Group meeting, Lugano, Italy* (2006).

[23] Hasert, M., Klimach, H. and Roller, S.: CAF versus MPI - Applicability of Coarray Fortran to a Flow Solver, *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, Berlin, Heidelberg, Springer-Verlag, pp. 228–236 (online), available from ⟨`ttp://dl.acm.org/citation.cfm?id=2042476.2042502`⟩ (2011).