情報処理学会論文誌　コンピューティングシステム　Vol.9　No.2　46–63 (July 2016)

メニーコアにおける派生データ型を用いたMPIノード内通信の高速化

島田 明男[1,2,a)]　須藤 敦之[1,b)]　堀 敦史[3,c)]　石川 裕[3,d)]　河野 健二[2,e)]

受付日 2015年11月6日, 採録日 2016年4月17日

概要：近年，マルチコアプロセッサのコア数は増加する傾向にある．また，低性能だが電力効率が高いコアを多数集約して高い並列処理性能を実現するメニーコアプロセッサの活用も進んでおり，HPC システムを構成するノード 1 台あたりのコア数は飛躍的に増加してきている．Partitioned Virtual Address Space（PVAS）は，このようなメニーコア環境において，効率的に並列処理を実行するための新たなタスクモデルである．PVAS を用いると，並列処理を行うノード内のプロセス群を同一アドレス空間で動作させることが可能になる．よって，アドレス空間の境界越しにデータを送受信するためのオーバヘッドなしに，ノード内通信を実行することができる．本研究では，PVAS を派生データ型を用いる Message Passing Interface（MPI）ノード内通信に適用し，同一ノードのプロセスがメモリ上で不連続なデータを高速に送受信することを可能にした．ベンチマークプログラムによって高速化した MPI ノード内通信を評価したところ，通信遅延を低減させ，実行性能を最大で約 21%向上させることができた．

キーワード：メニーコア，OS，MPI，ノード内通信，派生データ型

Accelerating MPI Intra-node Communication Using Derived Data Types on Many-core

Akio Shimada[1,2,a)]　Atsushi Sutoh[1,b)]　Atsushi Hori[3,c)]　Yutaka Ishikawa[3,d)]　Kenji Kono[2,e)]

Received: November 6, 2015, Accepted: April 17, 2016

Abstract: Recently, the number of cores in multi-core processors is expected to continue to grow and many-core processors, which aggregate massive number of weak but power-efficient cores, are widely used. Then, the number of cores in an HPC system node grows rapidly. Partitioned Virtual Address Space (PVAS) is a new task model, which enables efficient parallel processing in such many-core systems. The PVAS task model allows multiple processes to run in the same address space, which means that the processes running on the PVAS task model can exchange data without overheads for crossing address space boundaries. In this paper, we apply the PVAS task model to the Message Passing Interface (MPI) intra-node communication using Derived Data Types and accelerate the transfer of non-contiguous data. The accelerated MPI intra-node communication reduces communication latency and improves application performance by up to 21%.

Keywords: many-core, OS, MPI, intra-node communication, derived data types

1　株式会社日立製作所研究開発グループ
Hitachi, Ltd., Research & Development Group, Yokohama, Kanagawa 244–0817, Japan
2　慶應義塾大学
KEIO University, Yokohama, Kanagawa 223–8522, Japan
3　理化学研究所計算科学研究機構
RIKEN AICS, Kobe, Hyogo 650–0047, Japan
a)　akio.shimada.ht@hitachi.com
b)　atsushi.sutoh.ff@hitachi.com
c)　ahori@riken.jp
d)　yutaka.ishikawa@riken.jp
e)　kono@ics.keio.ac.jp

1. はじめに

近年では電力効率の観点からコア単体の処理性能を高めるよりも 1 プロセッサあたりのコア数を増加させることで高い処理能力を実現する CPU アーキテクチャが一般的になっており，HPC システムを構成するノード 1 台あたりのコア数は飛躍的に増加してきている．従来のマルチコアプロセッサに加え，低性能だが電力効率が高いコアを多数集約することで高い並列計算処理能力を実現するメニーコ

アプロセッサの活用も一般的になっている．実際，Intel® 社のメニーコア製品である Intel® Xeon Phi™ が多数の HPC システムにおいて現在用いられている [29]．

Message Passing Interface（MPI）[16] は並列計算処理のための通信規格であり，並列アプリケーションの開発に広く用いられている．MPI は複数のプロセスに並列処理を実行させる並列化モデルになっている．並列処理を実行するプロセスどうしは，並列処理に必要なデータを MPI ライブラリの提供する通信 API を用いて互いに送受信する．MPI によって並列処理を行うプロセスを MPI プロセスと呼ぶ．MPI を利用して実装されたアプリケーションを HPC システム上で実行する場合，システムの並列計算処理能力を効率的に利用するため，利用可能なコア数に応じて並列に動作する MPI プロセスの数を調整するのが一般的になっている．多数のコアを利用可能な環境では，より多くの MPI プロセスが動作することになる．

MPI アプリケーションの実行性能は MPI 通信の性能に大きな影響を受ける．MPI 通信は，異なるノード上で動作する MPI プロセスどうしの通信である MPI ノード間通信と，同一ノード上で動作する MPI プロセスどうしの通信である MPI ノード内通信に分けられる．ノード 1 台あたりのコア数が増加するメニーコアシステムでは多数の MPI プロセスが同一ノード上で動作する．よって，アプリケーション実行時に発生する MPI ノード内通信の回数が従来よりも増加する．MPI ノード間通信はもちろんのこと，MPI ノード内通信の性能がアプリケーションの実行性能に従来よりも大きな影響を与えることになり，高速な MPI ノード内通信が求められる．マルチコアプロセッサの登場以来，様々な方式の MPI ノード内通信 [3], [6], [7], [10] が提案されている．また，メニーコアシステムにおいて，MPI ノード内通信の性能向上が MPI アプリケーションの実行性能向上に寄与することが報告されている [30]．

並列処理を行う各 MPI プロセスは個別のアドレス空間で動作するため，ノード内通信の際，アドレス空間の境界越しにデータを送受信する必要がある．既存の MPI ノード内通信の実装では，共有メモリを用いてアドレス空間の境界越しにデータを送受信するのが一般的になっている．この方式では，共有メモリ上のバッファを経由して，送信プロセスの送信バッファから受信プロセスの受信バッファにデータをコピーして転送する．よって，共有メモリを経由してデータを転送するためのオーバヘッドが通信時に発生する．これは，アドレス空間の境界越しにデータを送受信するためのオーバヘッドといえる．このオーバヘッドにより，MPI ノード内通信の通信遅延が増加する．

そこで本研究では，Partitioned Virtual Address Space（PVAS）[24], [25], [32] を用いて，MPI ノード内通信を高速化することを提案する．PVAS は，メニーコアシステムにおいて効率的に並列処理を行うために，本研究の著者らが提案，開発した新たなタスクモデルである．PVAS は並列処理を行うノード内のプロセス群を同一アドレス空間で実行することを可能にする．既存のタスクモデルでは，並列処理を行うノード内の各プロセスが個別のアドレス空間で動作するため，互いのメモリに直接アクセスすることができない．よってノード内通信を行う際，アドレス空間の境界越しにデータを送受信するためのオーバヘッドが発生していた．対して，PVAS によって同一アドレス空間で動作するプロセスどうしは互いのメモリに直接アクセスすることができるため，アドレス空間の境界越しにデータを送受信するためのオーバヘッドなしに，ノード内通信を実行することができる．PVAS は MPI や Partitioned Global Address Space（PGAS）[1], [5], [11], [12] のような，ノード内通信が行われる並列化モデルに幅広く適用することができる．本研究では，並列処理を実行するノード内の MPI プロセスを，PVAS によって同一アドレス空間で実行し，送信プロセスの送信バッファから受信プロセスの受信バッファにデータを直接コピーして転送することを可能にする．共有メモリを経由せずにデータをコピーして転送することで，共有メモリを経由してデータを転送するためのオーバヘッドを MPI ノード内通信から排除し，通信遅延を低減させることができる．

MPI の派生データ型を用いると，メモリ上で不連続なデータを 1 つの型として定義し，不連続なデータの送受信をひとまとめに処理することができる．本研究の著者らは先行研究 [32] において，PVAS による高速化を MPI ノード内通信に適用し，メモリ上で連続したデータを同一ノード内の MPI プロセスが高速に送受信することを可能にした．本研究ではこれを拡張し，派生データ型を用いる MPI ノード内通信にも PVAS を適用することで，メモリ上に不連続に配置されたデータを同一ノード内の MPI プロセスが高速に送受信することを可能にする．

派生データ型を用いる MPI ノード内通信の高速化は，PVAS の特徴を利用して容易に実装することができる．不連続なデータを送信バッファから受信バッファに直接コピーするためには，送受信プロセスが，データ型の情報（送受信データのメモリ上での配置）を交換する必要があるが，PVAS によって同一アドレス空間で動作する送受信プロセスは，互いの MPI ライブラリの管理オブジェクトにアクセスし，この情報を容易に交換することができる．

高速化は代表的な MPI ライブラリの 1 つである Open MPI [18] のノード内通信の実装に対して行った．高速化した Open MPI のノード内通信を評価したところ，不連続なデータの送受信の通信遅延を低減させ，ベンチマークプログラムの実行性能を最大で約 21%向上させることができた．

2. 背景

本章では，MPI の派生データ型について説明する．ま

た，派生データ型を用いる MPI ノード内通信の実装とその問題点について述べる．実装については，Open MPI の実装をもとに述べるが，MPICH [17] をはじめとする他の主要な MPI ライブラリにおいても，ほぼ同様の方式で派生データ型を用いる MPI ノード内通信が実装されている．

2.1 MPI の派生データ型

MPI では，プロセス間で送受信するデータの型を通信時に指定する必要がある．たとえば，`MPI_Send` と `MPI_Recv` でデータを送受信する場合，**表 1** に示すように，引数 *datatype* で送受信するデータのデータ型を指定し，引数 *count* で，そのデータ型を連続して何個送受信するかを指定する．MPI では，使用できるデータ型として基本データ型があらかじめ定義されている．主要な基本データ型とそれらの C 言語での対応を**表 2** に示す．`MPI_Send` の引数 *datatype* に `MPI_FLOAT`，引数 *count* に 10 を指定した場合，`float` 型が 4 Bytes だとすると，40 Bytes の連続したデータが送信される．

MPI では基本データ型のほかに，ユーザプログラムが定義した派生データ型を通信に用いることができる．派生データ型を用いると，メモリ上で不連続なデータを 1 つの型として定義し，不連続なデータの送受信をひとまとめに処理することができる．たとえば，**図 1** (A) に示すような 2 次元配列のデータがあり，ブロック $\{0, 1, 4, 5\}$ のデータを送信する場合，送信するデータは図 1 (B) で示すように，メモリ上に不連続に配置される．基本データ型を用いて全データを送信するためには複数回通信を行う必要があるが，派生データ型を用いると，1 回の通信でデータを送信することができる．

派生データ型は，すでに定義されているデータ型のメモリ上での配置を指定することで定義できる．派生データ型を定義するための主要な関数を**表 3** に示した．図 1 (A) で示したようなデータ配置はベクター型のデータ配置と呼ばれ，図 1 (C) に示すように，`MPI_Type_vector` を用いることで型の定義を行うことができる．`MPI_Type_vector` の引数 *count* でブロックの数，*blklen* でブロックサイズ（*otype* で指定したデータ型を何個でひとかたまりのブロックとして扱うか），*stride* でブロック間の間隔を指定する．`MPI_Type_indexed` や `MPI_Type_struct` を用いれば，個々のブロックのサイズを異なるものにしたり，ブロック間の間隔をブロックごとに変えたりすることも可能である．`MPI_Type_contiguous` を用いることで，不連続なデータだけでなく，指定したデータ型がメモリ上に連続して配置される型も定義することができる．新たに定義したデータ型 *ntype* は，`MPI_Type_commit` を実行することで使用できるようになる．定義したデータ型を `MPI_Send` や `MPI_Recv` といった関数に用いてデータを送受信する．送信側と受信側で異なるデータ型を用いて通信することが可能である．データ型の情報は，データ型を定義したプロセスの MPI ライブラリ内に保存され，一度定義したデータ型を繰り返

表 1　MPI_Send と MPI_Recv（C 言語）
Table 1　MPI_Send and MPI_Recv (C language).

Synopsis of MPI_Send and MPI_Recv
int MPI_Send(const void *buf, int count, MPI_datatype datatype, int dest, int tag, MPI_comm comm)
int MPI_Recv(void *buf, int count, MPI_datatype datatype, int source, int tag, MPI_comm comm, MPI_status *status)

表 2　基本データ型（C 言語）
Table 2　Basic data type (C language).

MPI_datatype	**C datatype**
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	singed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double

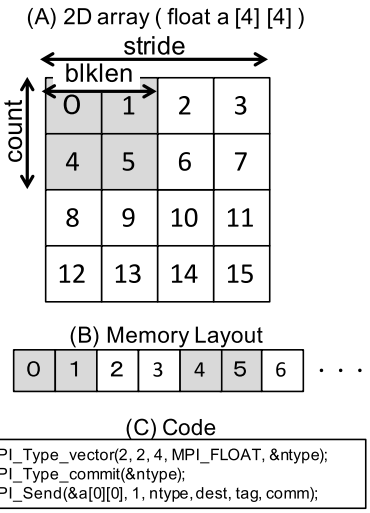

図 1　派生データ型を用いる通信
Fig. 1　Communication using derived data types.

表 3　派生データ型を定義するための主要な関数（C 言語）
Table 3　Functions for defining derived data types (C language).

Constructors for Derived Data Types
int MPI_Type_contiguous(int count, MPI_Datatype otype, MPI_Datatype *ntype)
int MPI_Type_vector(int count, int blklen, int stride, MPI_Datatype otype, MPI_Datattype *ntype)
int MPI_Type_indexed(int count, int blocklength[], int offsets[], MPI_Datatype otype, MPI_Datatype *ntype)
int MPI_Type_struct(int count, int blocklength[], MPI_Aint offsets[], MPI_Datatype otype[] MPI_Datatype *ntype)

し使用して通信を行うことができる．

2.2　派生データ型を用いる MPI ノード内通信の実装

複数のノードで構成されるクラスタ上で MPI アプリケーションを実行すると，ノード内通信とノード間通信が混在する．Open MPI の実装はモジュール構造となっており，MPI プロセスが通信を実行する際，通信先が同一ノード上のプロセスの場合はノード内通信用の通信モジュールが，通信先が異なるノード上のプロセスの場合はノード間通信用の通信モジュールが呼び出されるようになっている．Open MPI では，ノード内通信用の通信モジュールとして，共有メモリによるノード内通信が実装されている．本節では，派生データ型を用いる不連続なデータの送受信が，共有メモリを用いてどのように実装されているかを述べる．Open MPI は，他の主要な MPI ライブラリと同様に，同期通信である Rendezvous 通信と非同期通信である Eager 通信を，送受信するデータサイズに応じて使い分ける仕様になっている．まず，Rendezvous 通信の実装について述べる．

Rendezvous 通信では，まず送信プロセスが送信リクエストを受信プロセスの通信キューに追加する．受信プロセスは，受信リクエストが発行されたら，自身の通信キューを走査し，対応する送信リクエストを検索する．対応する送信リクエストを発見したら，それに対する応答（Ack）を送信プロセスの通信キューに追加する．送信プロセスが Ack を受信した時点で同期が完了し，データの転送が開始される．通信キューは，送信プロセスと受信プロセスの双方がアクセスするため，共有メモリ上に配置される．

データの転送は共有メモリ上に用意した中間バッファを経由したメモリコピーにより実行する．データのコピーは，複数の中間バッファを利用することでパイプライン的に処理される．まず送信プロセスは，用意した中間バッファに送信バッファから送信対象のデータをコピーする．そして，データをコピーした中間バッファのアドレス（実際には共有メモリ領域上のオフセット値）を含むメッセージを，受信プロセスの通信キューに追加する．これを，送信バッファからすべてのデータをコピーするまで繰り返す．受信プロセスは，送信プロセスから受信したメッセージに含まれる中間バッファのアドレスを参照し，共有メモリ上の中間バッファから受信バッファにデータをコピーする．この処理を，全データを受信バッファにコピーするまで繰り返す．受信プロセスが全データを受信バッファにコピーしたら，送受信処理が完了する．

このようにパイプライン的にデータのコピーを処理することで，送信バッファから共有メモリへのメモリコピーと，共有メモリから受信バッファへのメモリコピーをオーバラップさせることが可能になる．**図 2** は Rendezvous 通信における共有メモリを用いたデータの送受信を示している．図のケースでは，送信プロセスが送信バッファから中

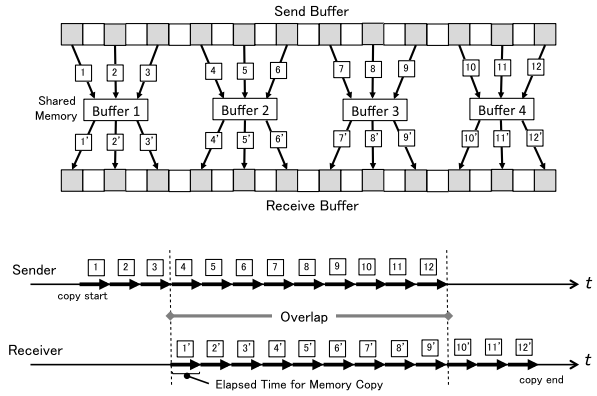

図 2　パイプライン的にデータをコピー
Fig. 2　Copying data in pipeline fashion.

間バッファ 2〜4 へのメモリコピーを実行している間に，受信プロセスが中間バッファ 1〜3 から受信バッファへのメモリコピーを実行可能になり，1 つの中間バッファに送信プロセスが全データをコピーしてから受信プロセスが受信バッファへのコピーを開始する場合よりも，データの転送にかかる時間が小さくなる．

この実装では，共有メモリ上の中間バッファを経由してデータを転送するため，メモリコピーの回数が多くなり，それがオーバヘッドとなって通信遅延が増加する．特に，送受信対象のデータが多数不連続にバッファ上に配置されるケースではメモリコピーの回数が大きく増加することになる．しかし，パイプライン的にデータを転送し，送信プロセス側と受信プロセス側のメモリコピーをオーバラップさせることができるため，メモリコピーの回数増加による通信遅延の増加をある程度抑制することができる．中間バッファのサイズを小さくすれば，オーバラップ可能なメモリコピーの割合が増えるが，中間バッファを経由したデータ転送の回数が増え，中間バッファの確保および中間バッファのアドレスを送受信する処理によるオーバヘッドで通信遅延が増加する．一方，中間バッファのサイズを大きくすれば，中間バッファを経由したデータ転送の回数を少なくすることができるが，パイプラインの段数も少なくなり，全体のメモリコピーのうちオーバラップできなかったメモリコピーの割合が増加し，通信遅延が増加してしまう．Open MPI では，デフォルトの中間バッファのサイズは 32 KBytes で，MPI プログラムの実行時に変更可能となっている．

次に Eager 通信の実装について説明する．Eager 通信では，まず送信プロセスが送信バッファ上の全データを共有メモリ上の中間バッファにコピーする．そして，中間バッファのアドレスを送信リクエストに含めて受信プロセスに送信し，処理を完了する．受信プロセスは，受信リクエストが発行されたら，自身の通信キューを走査し，該当する送信リクエストが発行されているか検索する．該当する送信リクエストを発見したら，送信リクエスト中の中間バッ

ファのアドレスを参照し，データを自身の受信バッファにコピーして処理を完了する．中間バッファに送信する全データをバッファリングすることで，受信プロセスが対応する受信リクエストを発行するのを待つことなく，送信プロセスが処理を完了することができる．

この実装では，共有メモリを経由してデータをコピーするため，Rendezvous 通信と同様，メモリコピーの回数が増加し，通信遅延が増加する．同期のための処理がない分，Rendezvous 通信よりも通信遅延が小さくなるが，単純に共有メモリを経由してデータを転送するため，送受信するデータサイズが大きくなると，データの転送をパイプライン的に処理する効果により，Rendezvous 通信の方が通信遅延が小さくなる．Open MPI では，Eager 通信と Rendezvous 通信を切り替えるデータサイズ（*Eager Threshold*）を MPI プログラムの実行時に指定することができる．*Eager Threshold* 以下のデータサイズの送受信は，Eager 通信で実行される．Open MPI では，*Eager Threshold* のデフォルト値は 4 KBytes となっている．

Rendezvous 通信，Eager 通信ともに，送信プロセスが共有メモリ上の中間バッファにデータをコピーする際と受信プロセスが共有メモリ上の中間バッファからデータをコピーする際に，当該通信で利用するデータ型の情報（通信対象のデータのメモリ上でのレイアウト）が必要となる．メモリ上で連続したデータを転送する場合，送受信プロセスは送受信バッファのアドレスとデータサイズさえ分かればデータのコピーを実行することができる．しかし，派生データ型を用いて不連続なデータの送受信を行う場合，送受信対象のデータが送受信バッファにどのように配置されているかを送受信プロセスが知る必要がある．通信バッファのアドレスと通信に利用するデータ型の情報は，Open MPI の実装では，*convertor* と呼ばれる管理オブジェクトに格納されている．*convertor* は通信を実行するたびに，通信を実行するプロセスのローカルメモリ（非共有メモリ）上に作成され，データ型の情報と通信バッファのアドレスが格納される．送信プロセスは自身の *convertor* を参照し，送信バッファから送信対象のデータを中間バッファにコピーする．一方受信プロセスは自身の *convertor* を参照し，受信データを格納する位置を確認して，共有メモリから受信バッファにデータをコピーする．

3. PVAS

PVAS は，並列処理を行うノード内のプロセスを同一アドレス空間で動作させることを可能にするメニーコアシステム向けの新たなタスクモデルである．本章では PVAS について説明する．

3.1 概要

図 3 は，既存のタスクモデルと PVAS のアドレス空間

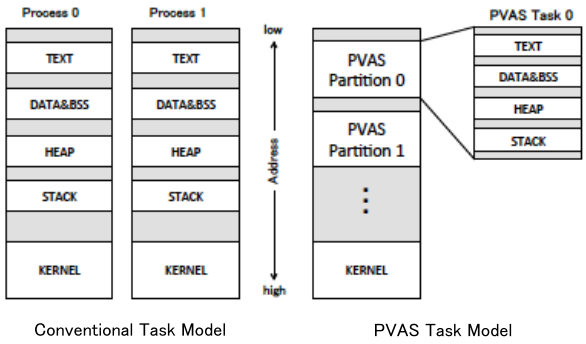

図 3　アドレス空間レイアウト
Fig. 3　Address space layout.

レイアウトを示している．図に示すように，既存のタスクモデルでは，複数のプロセスが同一ノード内で動作する場合，各プロセスが個別のアドレス空間で動作する．TEXT/BSS/DATA/HEAP/STACK といったメモリセグメントは個別のアドレス空間にマッピングされる．それに対し，PVAS では，複数のプロセスが同一アドレス空間で動作することを可能にする．PVAS は，1 つのアドレス空間を PVAS パーティションと呼ぶ領域に分割し，同一アドレス空間で動作させる各プロセスに割り当てる．PVAS で実行されるプロセスを，通常のプロセスと区別するため，PVAS タスクと呼ぶ．各 PVAS タスクは TEXT/BSS/HEAP/STACK といったメモリセグメントを，各自個別の PVAS パーティション内にマッピングする．各 PVAS タスクは，自身の PVAS パーティションを自身が利用可能なアドレス領域として認識する．

PVAS タスクと通常のプロセスの違いは，他のプロセスとアドレス空間を共有しているか否かのみである．PVAS タスクは，通常のプロセスと同様，独自のメモリセグメント（TEXT/BSS/HEAP/STACK），ファイルディスクリプタ，プロセス ID，シグナルハンドラ等を持つ．よって，ソースコードへの改変なしに，既存のプログラムを PVAS タスクとして実行することができる．

Intel®がリリースする Xeon Phi™ 用の Linux®カーネル [9] に PVAS を実装した．PVAS の仕様と実装の詳細については，文献 [25], [32] に詳しい．

3.2 PVAS を用いたノード内通信

プロセスが使用するメモリのマッピング情報は，ページテーブルと呼ぶ OS カーネル内の管理テーブルに格納されている．CPU は現在実行されているプロセスのページテーブルを参照して，アドレス空間上の論理アドレスを物理メモリ上の実アドレスに変換し，メモリにアクセスする．既存のタスクモデルでは，各プロセスが個別のページテーブルを持ち，独自のアドレス空間で動作する．よって，他のプロセスが使用しているメモリに直接アクセスすることができず，アドレス空間の境界越しにデータを送受信する

ためのオーバヘッドがノード内通信の際に発生する．対して，同一アドレス空間で動作する PVAS タスク群のメモリのマッピング情報は，同一ページテーブルを使用して管理される．よって，他の PVAS タスクが使用しているメモリに load/store 命令で直接アクセスすることが可能になり，アドレス空間の境界越しにデータを送受信するためのオーバヘッドなしにノード内通信を実行することができる．

先行研究 [32] では，メモリ上で連続したデータを送受信する際の MPI ノード内通信を PVAS によって高速化した．メモリ上で連続したデータを送受信する場合も，共有メモリを用いた実装が一般的になっており，共有メモリを経由してデータを転送するためのオーバヘッドがノード内通信時に発生する．MPI プロセスを PVAS タスクとして起動することで送信プロセスの送信バッファから受信プロセスの受信バッファにデータを直接コピーすることを可能にし，連続したデータを送受信する際の通信遅延を低減させることができた．

3.3 PVAS タスク間のメモリ保護について

PVAS を用いると，高速なノード内通信が可能になる一方，プログラムの生産性が低下する懸念がある．PVAS では，プロセス（PVAS タスク）間のメモリ保護がないため，プロセスをまたいだメモリ破壊が発生する可能性がある．プロセスをまたいだメモリ破壊は，同一プロセス内で発生するメモリ破壊と比べ，メモリ破壊の箇所やメモリ破壊を起こしているコードを特定するのが困難である．また，プログラムの異常がプロセスをまたいだメモリ破壊に起因することを検出する作業も大きな負担となる．

MPI アプリケーションの開発を考えた場合，テストの段階においては，通常のタスクモデルと既存の MPI 通信を用いることで，プロセスをまたいだメモリ破壊が発生しないことを前提に開発を進めることができる．しかし，実際に PVAS を適用した MPI 通信を用いて MPI アプリケーションの実行性能を評価する段階では，プロセスをまたいだメモリ破壊が発生する可能性があり，その場合は対処が必要である．また，PVAS を適用した MPI 通信を MPI ライブラリに実装するケースでは，テストの段階においても，プロセスをまたいだメモリ破壊が発生する可能性があることを前提に開発を行わなくてはならない．

プロセスをまたいだメモリ破壊を検出するためには，PVAS タスクをロードするアドレス領域（PVAS パーティション）をランダムに変えて複数回プログラムを実行する方式が考えられる．もし，ロードしたアドレス領域によって，プログラムの挙動に変化が生じるのであれば，プログラムの異常がプロセスをまたいだメモリ破壊に起因することを疑うことができる．

4. 派生データ型を用いる MPI ノード内通信の高速化

2 章で示したように，既存の MPI ノード内通信の実装では，共有メモリを経由してデータを転送するために以下のオーバヘッドが発生し，派生データ型を用いる不連続なデータの送受信の際に通信遅延が増加する．

- 共有メモリを経由したデータ転送によるメモリコピー回数の増加
- 共有メモリ上に中間バッファを確保する処理およびデータをコピーした中間バッファのアドレスを送受信する処理

これらはアドレス空間の境界越しにデータを送受信するためのオーバヘッドと定義することができる．通信を行う MPI プロセスが異なるアドレス空間で動作するため，共有メモリを経由してアドレス空間の境界越しにデータの転送を行う必要があり，これらのオーバヘッドが発生していた．

そこで本研究では，アドレス空間の境界越しにデータを送受信するためのオーバヘッドを PVAS によって排除し，派生データ型を用いる MPI ノード内通信を高速化することを提案する．PVAS によって同一ノード内の MPI プロセスを PVAS タスクとして起動し，MPI プロセスを同一アドレス空間で動作させる．MPI プロセスを同一アドレス空間で動作させることで，送信プロセスの送信バッファから受信プロセスの受信バッファにデータを直接コピーすることが可能になり，共有メモリを経由してデータを転送するためのオーバヘッドを，派生データ型を用いる MPI ノード内通信から排除することができる．

PVAS による高速化は Rendezvous 通信のみを対象にして行った．送信バッファから受信バッファに直接データをコピーするためには，送信バッファと受信バッファの双方の準備が通信開始時に完了していなければならない．Eager 通信の場合，送信プロセスと受信プロセスが非同期に送受信処理を行う．よって，通信開始時に送信バッファと受信バッファの双方の準備が完了しているとは限らず，本研究の提案を適用するのは困難である．実装は，共有メモリを用いる既存のノード内通信用モジュールを改良することで行った．Rendezvous 通信のみを改良し，Eager 通信については，既存の実装をそのまま採用している．以下に，PVAS によって高速化した Rendezvous 通信の Open MPI への実装について述べる．

本実装では，まず送信プロセスが送信リクエストを受信プロセスの通信キューに追加する．この送信リクエストには，送信プロセスがこの通信に使用する *convertor* のアドレスを含めておく．送信リクエストを受信した受信プロセスは，Ack を送信プロセスの通信キューに追加する．Ack には受信プロセスがこの通信で使用する *convertor* のアドレスを含めておく．Ack を送信した受信プロセスは送信

バッファから受信バッファへのデータのコピーを開始する．受信プロセスは，送信リクエストに含まれたアドレスを参照して送信プロセスの *convertor* にアクセスし，送信側のデータ型の情報と送信バッファのアドレスを得る．送信プロセスの *convertor* から送信側のデータ型の情報を得た受信プロセスは，それを自身の *convertor* から得た受信側のデータ型の情報と照らし合わせて，送信バッファから受信バッファにデータを直接コピーする．このとき，**図 4** A に示すように，全データのコピーは行わず，全体の 2 分の 1 のデータを送信バッファの先頭から受信バッファにコピーする．

一方，Ack を受信した送信プロセスは，Ack に含まれたアドレスを参照して受信側の *convertor* にアクセスし，受信側のデータ型の情報と受信バッファのアドレスを得る．受信プロセスの *convertor* から受信側のデータ型の情報を得た送信プロセスは，それを自身の *convertor* から得た送信側のデータ型の情報と照らし合わせて，図 4 A に示すように，受信プロセスがコピーを行わない残りの 2 分の 1 のデータを送信バッファから受信バッファにコピーする．

送受信プロセスは，データのコピーを終えたら，データのコピーが完了したことを示すメッセージを相手に送信する．これにより送受信プロセスは，データのコピーを終えたことを互いに確認し，通信処理を完了する．

この方式では，送信バッファから受信バッファに直接データをコピーするので，共有メモリ上の中間バッファを経由してデータを転送する際のオーバヘッドを回避することが可能になり，以下の理由により通信遅延が低減する．

- 共有メモリ上の中間バッファを経由してデータを転送する必要がないので，データの転送に必要なメモリコピーの回数が少なくなる．
- データの転送時に中間バッファを確保する処理および中間バッファのアドレスを送受信する処理が必要ない．

また，この方式では，送信プロセスと受信プロセスが送受信するデータの 2 分の 1 ずつを並列にコピーするため，図 4 B に示すように 1 プロセスでデータを順にコピーする場合とくらべ，データのコピーにかかる時間が 2 分の 1 になる．スレッド実装の MPI のノード内通信において，連続したデータを 2 分の 1 に分割して受信プロセスと送信プロセスが並列にコピーする方式が提案されており [21]，連続したデータの転送にこの方式を用いる場合は，ノード内通信の通信遅延が大きく低減されることがすでに報告されている．

共有メモリ上の中間バッファを経由して不連続なデータの送受信を行う場合，送信プロセスと受信プロセスは各自の *convertor* にアクセスし，データ型の情報を確認すれば，データのコピーを実行することができた．しかし，送受信プロセスが並列して，送信バッファから受信バッファに直接データをコピーする場合，送受信プロセスが互いの *convertor* にアクセスし，送信側と受信側双方のデータ型を確認する必要がある．PVAS を用いると通信バッファだけでなく，MPI ライブラリの管理オブジェクトにも送受信プロセスが互いにアクセスできるようになる．送信側の *convertor* は送信プロセスのローカルメモリ上に，受信側の *convertor* は受信プロセスのローカルメモリ上に存在するが，PVAS を用いる場合は，送受信プロセスが互いの *convertor* にアクセスすることができる．

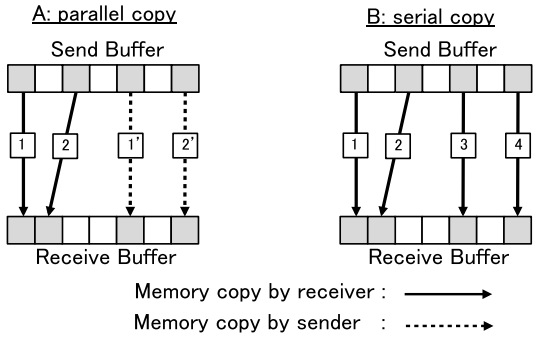

図 4　高速化した派生データ型の MPI ノード内通信

Fig. 4　Optimized MPI intra-node communication for derived data types.

5. 評価

高速化した MPI ノード内通信をベンチマークプログラムによって評価した．測定には代表的なメニーコアプロセッサである Intel® Xeon Phi™ を用いた．評価環境を**表 4** に示す．

クラスタ環境を用意することができなかったため，単一ノードによるノード内通信のみの評価を行った．先行研究 [6], [30] では，ノード間通信が混在している状況でも，ノード内通信の高速化がアプリケーションの実行性能向上に寄与することが報告されており，ノード内通信のみの評価でも，本研究の有用性を検証できると判断した．

5.1 通信遅延

DDTBench [23] により，派生データ型を用いる MPI ノード内通信の通信遅延を測定した．DDTBench は，メモリ上に不連続に配置されているデータの送受信を行う各種 MPI

表 4　評価環境

Table 4　Evaluation environments.

項目	内容
プロセッサ	Intel® Xeon Phi™ coprocessor 5110P ・物理コア数 60，論理コア数 240（4HT/コア） ・32 KB L1 キャッシュ，512 KB L2 キャッシュ ・8 GBytes メインメモリ
OS カーネル	Xeon Phi™ 用 Linux®カーネル + PVAS
MPI	Open MPI version 1.8

表 5　DDTBench の再現する通信パターン（文献 [23] の 4 ページより引用）
Table 5　Communication patterns in DDTBench (paper [23], p.4).

Application	Test name	Communication Pattern
Atmospheric Science	WRF_x_vec WRF_y_vec WRF_x_sa WRF_y_sa	struct of 2D/3D/4D face exchanges in different directions (x,y), using different (semantically equivalent) datatypes: nested vectors (_vec) and subarrays (_sa)
Quantum Chromodynamics	MILC_su3_zd	4D face exchange, z direction, nested vectors
Fluid Dynamics	NAS_MG_x NAS_MG_y NAS_MG_z	3D face exchange in each direction (x,y,z) with vectors (y,z) and nested vectors (x)
	NAS_LU_x NAS_LU_y	2D face exchange in x direction (contiguous) and y direction (vector)
Matrix Transpose	FFT	2D FFT, different vector types on send/recv side
	SPECFEM3D_mt	3D matrix transpose
Molecular Dynamics	LAMMPS_full LAMMPS_atomic	unstructured exchange of different particle types (full/atomic), indexed datatypes
Geophysical Science	SPECFEM3D_oc SPECFEM3D_cm	unstructured exchange of acceleration data for different earth layers, indexed datatypes

アプリケーション（WRF [26], SPECFEM3D_GLOBE [13], MILC [2]，LAMMPS [20]，NAS Parallel Benchmarks [4]）の通信パターンを派生データ型の通信で再現して実行する．DDTBench は各種 MPI アプリケーションの通信のみを再現して計算処理は行わない．DDTBench が再現，実行する各種 MPI アプリケーションの通信パターンを**表 5** に示す．2 つの MPI プロセスが表に示す内容の通信パターンを派生データ型を用いる MPI 通信によって実行する．2 つの MPI プロセス間で Ping-pong 通信が実行され，その通信遅延が測定される．これを用い，共有メモリを用いる既存の Rendezvous 通信と PVAS によって高速化した Rendezvous 通信の通信遅延を比較した．

DDTBench による通信遅延の測定結果を**図 5** に示す．*SM* は共有メモリを用いる既存の Rendezvous 通信の通信遅延を，*PVAS* は PVAS によって高速化した Rendezvous 通信の通信遅延を示している．測定において，いずれのデータサイズにおいても Rendezvous 通信が実行されるように *Eager Threshold* の値を 0 にした．また，既存の Rendezvous 通信については，共有メモリ上に確保する中間バッファのサイズを 4 KBytes から 64 KBytes に設定して測定を行った．

PVAS によって高速化した Rendezvous 通信は，前章で述べた理由により，NAS_MG_x を除いて，既存の Rendezvous 通信より通信遅延が小さくなった．NAS_MG_x では，PVAS によって高速化した Rendezvous 通信と既存の Rendezvous 通信の通信遅延の差が他の通信パターンよりも小さく，送受信するデータサイズが 2 KBytes および 32 KBytes のときは，既存の Rendezvous 通信よりも通信遅延が約 20〜30%大きくなった．これは，データのコピーを実行している間に発生する CPU のキャッシュミスに起因すると考えられる．NAS_MG_x において，PVAS によって高速化した Rendezvous 通信の通信遅延が既存の Rendezvous 通信の通信遅延よりも大きくなってしまう理由を以下に述べる．

図 6 は，NAS_MG_x のソースコードにおいて，データ型の定義を行っている部分を抜粋したものである．新たなデータ型 `dtype_tmp_t` を定義し，そのデータ型から，また新たなデータ型 `dtype_face_x_t` を定義している．NAS_MG_x では，このデータ型を用いて送受信プロセスが不連続なデータを送受信する．`dtype_face_x_t` のデータ配置を図示すると**図 7** のようになる．図に示すように，小さなサイズ（*MPI_DOUBLE* [*1] Bytes）のデータが多数メモリ上に不連続に配置される．このようなデータ型どうしで不連続なデータの送受信を行うと，**図 8** に示すように，データを送信バッファから受信バッファにコピーしている間に，キャッシュのブロックをまたいだメモリアクセスによる CPU のキャッシュミスが頻繁に発生し，データの転送にかかる時間が大きくなる．共有メモリを用いる既存の Rendezvous 通信の場合，共有メモリ上の中間バッファを経由してデータを転送する．送受信プロセスがデータのコピーを行う際にアクセスするバッファのうち，共有メモリ上の中間バッファへのアクセスは連続したメモリアクセスとなるため，多数の不連続なデータが配置されるバッファ間でデータを直接コピーする場合よりも，各プロセスを実行している CPU 上で発生するキャッシュミスの回数が少なくなる場合がある．その結果，PVAS によって高速化した Rendezvous 通信と共有メモリを用いる既存の Rendezvous 通信の通信遅延の差が小さくなり，場合によっては，既存の Rendezvous 通信の方が通信遅延が小さくなってしまう．

[*1] 本評価環境では，MPI_DOUBLE のサイズは 8 Bytes.

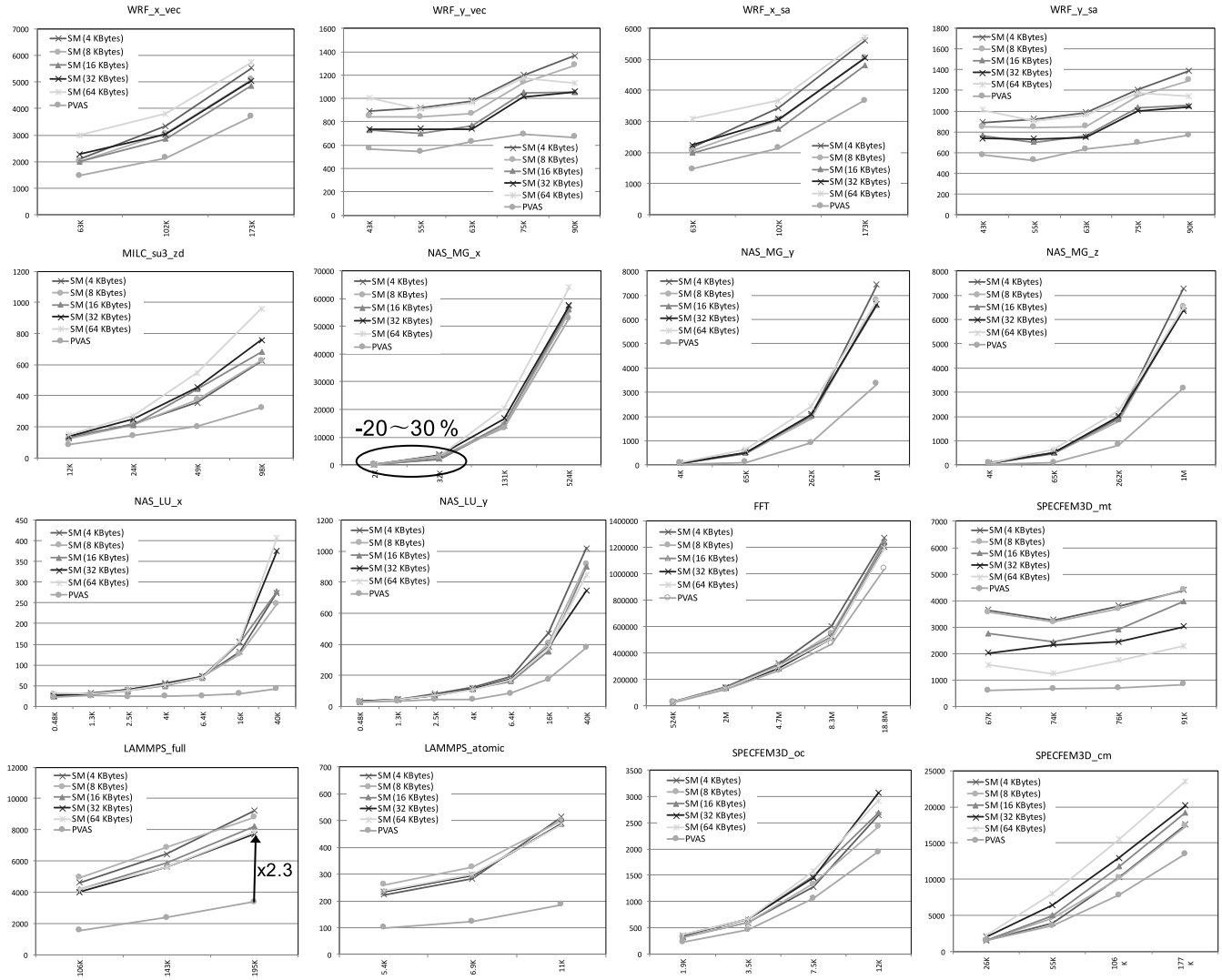

図 5　DDTBench の実行結果（横軸：データサイズ [Bytes]，縦軸：通信遅延 [usec]）
Fig. 5　DDTBench results (horizontal-axis: Data size [Bytes], vertical-axis: Latency [usec]).

```
MPI_Type_vector( DIM2-2, 1, DIM1, MPI_DOUBLE, &dtype_temp_t );
・・・
MPI_Type_create_hvector( DIM3-2, 1, stride, dtype_temp_t, &dtype_face_x_t );
MPI_Type_commit( &dtype_face_x_t );
```

図 6　NAS_MG_x のデータ型の定義
Fig. 6　Codes for defining the data type for NAS_MG_x.

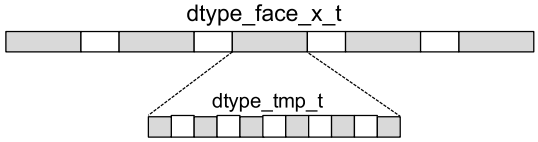

図 7　NAS_MG_x のデータ型
Fig. 7　Data type for NAS_MG_x.

実際，NAS_MG_x では，図 12（詳細は後述）に示すように，PVAS によって高速化した Rendezvous 通信の方が，通信時に発生するキャッシュミスの回数が多くなっている．

NAS_MG_x の測定において，送受信するデータサイズ

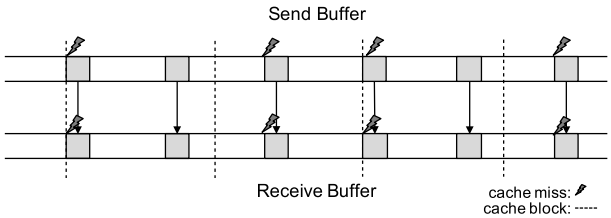

図 8　NAS_MG_x のデータコピー
Fig. 8　Data copy of NAS_MG_x.

が 2 KBytes の場合，キャッシュミスが減少する効果が大きく，共有メモリを用いる既存の Rendezvous 通信の方が PVAS によって高速化した Rendezvous 通信よりも通信遅延が小さくなる．送受信するデータサイズが 32 KBytes の場合，中間バッファのサイズが 4 KBytes から 16 KBytes のときは，メモリコピーがオーバラップ可能になることに加え，キャッシュミスが減少する効果により，共有メモリを用いる既存の Rendezvous 通信の方が PVAS によって高

```
MPI_Type_create_indexed_block( icount, 3, &index_displacement[0],
MPI_DOUBLE, &dtype_indexed3_t );
・・・
MPI_Type_create_struct( 6, &blocklength[0], &address_displacement[0],
&oldtype[0], &dtype_send_t );
MPI_Type_commit( &dtype_send_t );
・・・
MPI_Type_contiguous( 3*icount, MPI_DOUBLE, &dtype_cont3_t );
MPI_Type_create_struct( 6, &blocklength[0], &address_displacement[0],
&oldtype[0], &dtype_recv_t );
MPI_Type_commit( &dtype_recv_t );
```

図 9　LAMMPS_full のデータ型の定義
Fig. 9　Codes for defining the data type for LAMMPS_full.

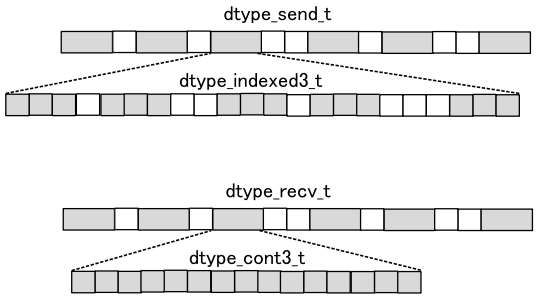

図 10　LAMMPS_full のデータ型
Fig. 10　Data type for LAMMPS_full.

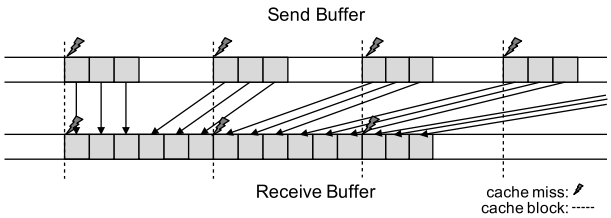

図 11　LAMMPS_full のデータコピー
Fig. 11　Data copy of LAMMPS_full.

速化した Rendezvous 通信よりも通信遅延が小さくなる．中間バッファのサイズが 32 KBytes 以上の場合は，たとえキャッシュミスの回数が減少しても，PVAS で高速化した Rendezvous 通信の方が通信遅延が小さくなる．送受信するデータサイズが 131 KBytes 以上の場合，中間バッファのサイズが小さいと，中間バッファを経由したデータ転送の回数が増加し，中間バッファの確保および中間バッファのアドレスを送受信する処理のオーバヘッドが大きくなる．中間バッファのサイズが大きい場合は，中間バッファを経由したデータ転送の回数が少なくなるが，オーバラップ可能なメモリコピーの割合は減少する．これらの損益がキャッシュミスの回数が減少する利得を上回り，データサイズが 131 KBytes 以上のときは，中間バッファがどのサイズでも，PVAS で高速化した Rendezvous 通信の方が通信遅延が小さくなると考えられる．

NAS_MG_x のように PVAS で高速化した Rendezvous 通信と共有メモリを用いる既存の Rendezvous 通信の通信遅延の差が小さく，場合によっては既存の Rendezvous 通信の方が通信遅延が小さくなるものがある一方で，LAMMPS_full のように，PVAS によって高速化した Rendezvous 通信と既存の Rendezvous 通信の通信遅延の差が，他の通信パターンと比べて大きいものもあった．LAMMPS_full では，既存の Rendezvous 通信の通信遅延が PVAS による実装の約 2.3 倍にもなっている．**図 9** は，LAMMPS_full のソースコードにおいて，データ型の定義を行っている部分を抜粋したものである．LAMMPS_full では，送信側と受信側で異なるデータ型を用いる．送信側では，新たなデータ型 `dtype_indexed3_t` を定義し，そのデータ型から，また新たなデータ型 `dtype_send_t` を定義している．受信側では，新たなデータ型 `dtype_cont3_t` を定義し，そのデータ型から，また新たなデータ型 `dtype_recv_t` を定義している．`dtype_send_t` と `dtype_recv_t` のデータ配置を図示すると **図 10** のようになる．図に示すように，送信側ではある程度大きなサイズ（$MPI\_DOUBLE \times 3$ Bytes）のデータが不連続に送信バッファ上に配置される，一方受信側では，送信側よりも大きなサイズ（$MPI\_DOUBLE \times (3 \times icount)$ Bytes）のデータが不連続に受信バッファ上に配置される．このようなデータ型を用いて通信を行うと，小さなサイズのデータが多数不連続に配置されるデータ型どうしで通信を行う場合に比べて，データの転送を行っている間に発生する CPU のキャッシュミスの頻度は低くなる．**図 11** に示すように，より少ない回数のキャッシュミスで，大量のデータを送信バッファから受信バッファに直接コピーすることが可能になる．キャッシュミスによる通信遅延への影響が小さくなり，より高速にデータを送受信できる．その結果，共有メモリを用いる既存の Rendezvous 通信との通信遅延の差が大きくなる．

図 12 は，NAS_MG_x と LAMMPS_full の通信時に，各 CPU 上で発生するキャッシュミスの回数を PAPI [28] によって測定した結果を示している．NAS_MG_x においては，PVAS によって高速化した Rendezvous 通信よりも共有メモリを用いる既存の Rendezvous 通信の方が通信遅延が小さかったデータサイズ（2 KBytes および 32 KBytes）の測定結果を示した．また，共有メモリを用いる既存の Rendezvous 通信の中間バッファのサイズは，最も通信遅延が小さかった 4 KBytes に設定して測定した．LAMMPS_full においては，中間バッファのサイズを最も通信遅延が小さかった 32 KBytes にして測定した結果を示した．Xeon Phi^{TM} は L1 および L2 キャッシュを持つが，Xeon Phi^{TM} のパフォーマンスカウンタは L2 キャッシュのイベントのカウントをサポートしていないため，L1 キャッシュのキャッシュミスのみを測定した．

NAS_MG_x の場合，共有メモリを用いる既存の Rendezvous 通信の方がキャッシュミスの回数が推測どおり少なくなっている．対して LAMMPS_full の場合は既存の Rendezvous 通信の方がキャッシュミスの回数が多くなっている．PVAS を用いて送信バッファから受信バッファに

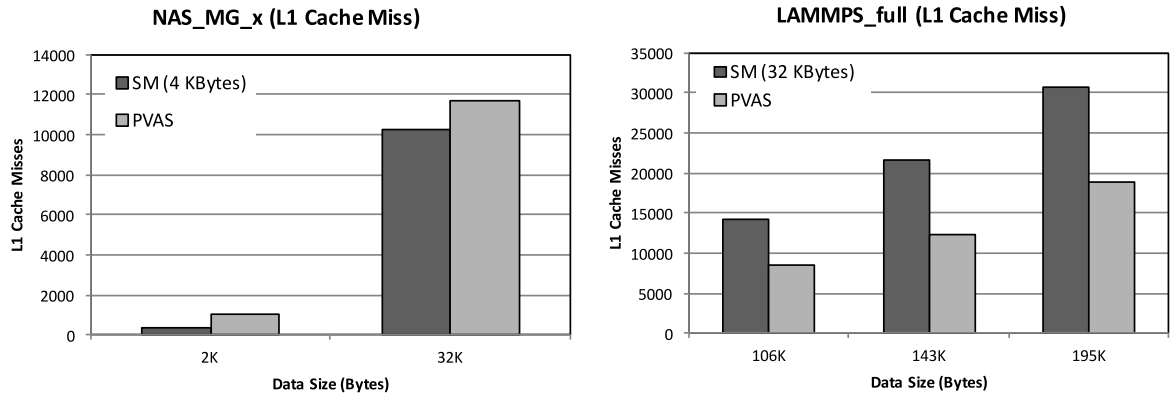

図 12　L1 キャッシュミスの回数
Fig. 12　L1 cache misses.

データを直接コピーして転送する場合，送受信プロセスがアクセスするのは送信バッファと受信バッファだけでよい．しかし，共有メモリを経由してデータをコピーして転送する場合，送受信プロセスが送信バッファと受信バッファに加え，共有メモリ上の中間バッファにもデータの転送中にアクセスするので，かえってキャッシュミスの回数が増加してしまっている．

DDTBench による測定の結果，通信に用いるデータ型や送受信するデータサイズによっては，PVAS によって高速化した Rendezvous 通信の方が，共有メモリを用いる既存の Rendezvous 通信よりも，不連続なデータの送受信において，かえって通信遅延が増加してしまうケースがあることが分かった．これは，PVAS によって高速化した MPI ライブラリにおいて，通信に使用するデータ型の内容を解析し，通信バッファ間で直接データをコピーする実装と共有メモリを経由してデータをコピーする実装のどちらを用いるのが適切かを自動的に判別して切り替えることで解決できると考える．現状では，mpiexec（または mpirun）コマンドの引数で，PVAS による実装と共有メモリによる実装のどちらを Rendezvous 通信に使用するかを，ユーザが MPI アプリケーションの起動時に選択する仕様になっているが，将来的には，MPI ライブラリが自動的に使用する実装を切り替える方式もサポートする予定である．

5.2　ミニアプリケーション

5.2.1　fft2d_datatype

次に，fft2d_datatype [22] を用いて PVAS によって高速化した Rendezvous 通信と共有メモリを用いる既存の Rendezvous 通信を比較した．fft2d_datatype は，派生データ型を用いる MPI 通信の性能を評価するためのベンチマークとしてチューリッヒ工科大学の Scalable Computing Laboratory によって開発された．fft2d_datatype は 2 次元フーリエ変換の計算コードを実行するミニアプリケーションで，並列プロセス間のデータの送受信を派生データ型を用いる MPI 通信で行う．DDTBench とは異なり，通信だけではなく，フーリエ変換の計算処理を実際に実行するため，通信と計算処理を含めた総合的な評価に用いることができる．fft2d_data_type では，送受信側ともに `MPI_Type_vector` によって定義したデータ型を用いる．ともにベクター型の不連続なデータ配置の型であるが，ブロックサイズとブロック数は送受信側で異なるものになっている．この 2 つの型を用いて，データを送受信する．

測定実行時のプロセス数は 240 とした．また，フーリエ変換の対象となる 2 次元配列のデータの要素数については，4,800 × 4,800 と 9,600 × 9,600 の場合で測定を行った．配列の要素数が 9,600 × 9,600 のときにメモリ消費量が約 8 GBytes となり，本評価で用いた Intel® Xeon Phi™ に搭載されているメインメモリの容量をほぼ消費する．測定は，つねに Rendezvous 通信が実行されるように *Eager Threshold* の値を 0 にして行った．また，共有メモリを用いる既存の Rendezvous 通信については，中間バッファのサイズをデフォルト値の 32 KBytes として測定した．fft2d_datatype の実行結果を**図 13** に示す．上部のグラフは fft2d_datatype の実行時間を，下部のグラフは PVAS によって高速化した Rendezvous 通信を用いたときの実行性能の改善率を示している．

配列サイズが 9,600 × 9,600 の場合，PVAS によって高速化した Rendezvous 通信を用いると，実行性能が約 21%改善された．配列サイズが 4,800 × 4,800 の場合は，実行性能の改善率が約 5%にとどまっている．計算対象の配列サイズが大きい場合の方が，配列サイズが小さい場合よりも実行性能の改善率が高い．PVAS によって高速化した Rendezvous 通信では，送信バッファから受信バッファへのデータの転送にかかる時間が共有メモリによる実装よりも短くなる．よって，送受信するデータサイズが大きい方が高速化の効果が大きく，共有メモリによる既存の Rendezvous 通信との通信遅延の差が大きくなる．配列サイズが大きい場合，送受信するデータのサイズが大きくなるので，共有メモリを用いる既存の Rendezvous 通信と PVAS によって高速化した Rendezvous 通信との通信遅延

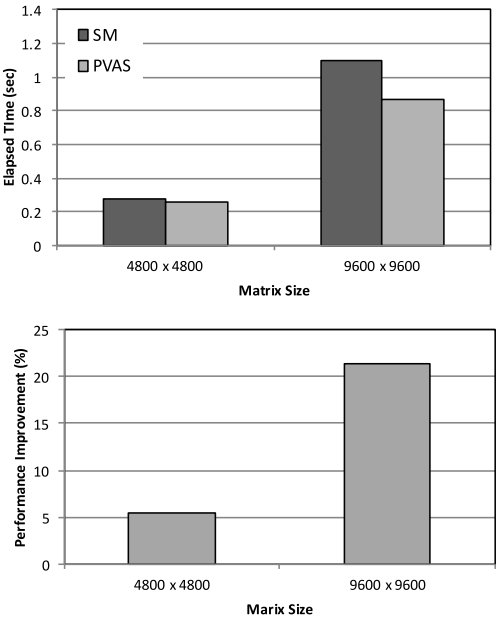

図 13　fft2d_datatype の実行結果
Fig. 13　fft2d_datatype results.

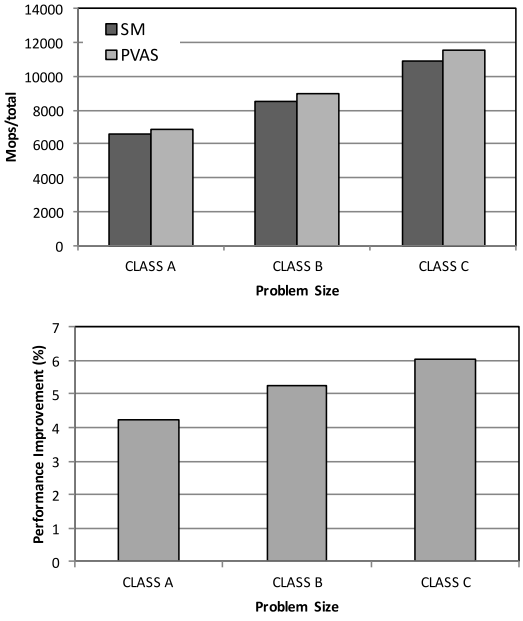

図 14　NAS LU の実行結果
Fig. 14　NAS LU results.

の差が大きくなり，配列サイズが小さいときと比べて実行性能の改善率が高くなっていると考えられる．

5.2.2　NAS LU

NAS Parallel Benchmarks は，MPI によって実装されたベンチマーク群によって構成されており，HPC システムの性能評価に用いられる．NAS Parallel Benchmarks に含まれるベンチマークの 1 つである LU ベンチマークは，流体力学の計算コードであり，本来 pack/unpack 処理によって不連続なデータの送受信を行う．pack/unpack による通信とは，送信側プロセスが不連続なデータを通信用バッファに連続データとして再配置する処理（pack）を行ってから送信し，受信側が受信した連続データを不連続なデータとして受信バッファに再配置して（unpack），データを送受信する方法である．しかし，これを改良し，派生データ型を用いる MPI 通信によって不連続なデータの送受信を行う LU ベンチマークがチューリッヒ工科大学の Scalable Computing Laboratory によって開発されている [22]．次に，このベンチマークによって評価を行った．

LU ベンチマークは，並列プロセスの数が 2 の累乗でなければならないという制限があるため，実行時の MPI プロセス数は 128 とした．NAS Parallel Benchmarks では，それぞれのベンチマークに対し，問題サイズが異なる 7 つのクラス（S < W < A < B < C < D < E）が用意されている．本測定では，標準的な問題サイズとされるクラス A，B，C を用いた．LU ベンチマークは表 5 に示すように，連続したデータの送受信と不連続なデータの送受信を実行する．本評価では，不連続なデータの送受信の高速化について評価を行うのが目的であるため，連続したデータの送受信については，共有メモリを用いる既存の Rendezvous 通信についても，先行研究 [32] において提案した PVAS で高速化した Rendezvous 通信が用いられるようにした．測定は，つねに Rendezvous 通信が実行されるように *Eager Threshold* の値を 0 にして行った．また，共有メモリを用いる既存の Rendezvous 通信については，中間バッファのサイズをデフォルト値の 32 KBytes として測定した．

LU ベンチマークの実行結果を**図 14** に示す．上部のグラフは実行性能を，下部のグラフは PVAS によって高速化した Rendezvous 通信を用いたときの実行性能の改善率を示している．グラフに示すとおり，実効性能を約 4%から約 6%改善することができた．fft2d_datatype は，全体の実行時間において通信の占める割合が大きいため，通信遅延が低減された効果が大きく現れるが，LU ベンチマークは，全体の実行時間において通信の占める割合が fft2d_datatype ほど大きくはないため，通信遅延の低減による実行性能の改善率は約 4%から約 6%にとどまっていると考えられる．

5.3　Eager 通信との比較

最後に，Eager 通信と PVAS によって高速化した Rendezvous 通信の比較を行った．まず，DDTBench によって，Eager 通信の通信遅延を測定し，PVAS によって高速化した Rendezvous 通信の通信遅延と比較した．Eager 通信の測定の際は，つねに Eager 通信が実行されるよう，*Eager Threshold* を十分大きな値に設定して測定を行った．結果を**図 15** に示す．

グラフに示すとおり，PVAS によって高速化した Rendezvous 通信は，一部のケース（NAS_MG_x でデータサイズが 2 KBytes，NAS_lu_x でデータサイズが 0.48K～1.3 KBytes，NAS_lu_y でデータサイズが 0.48 KBytes）を

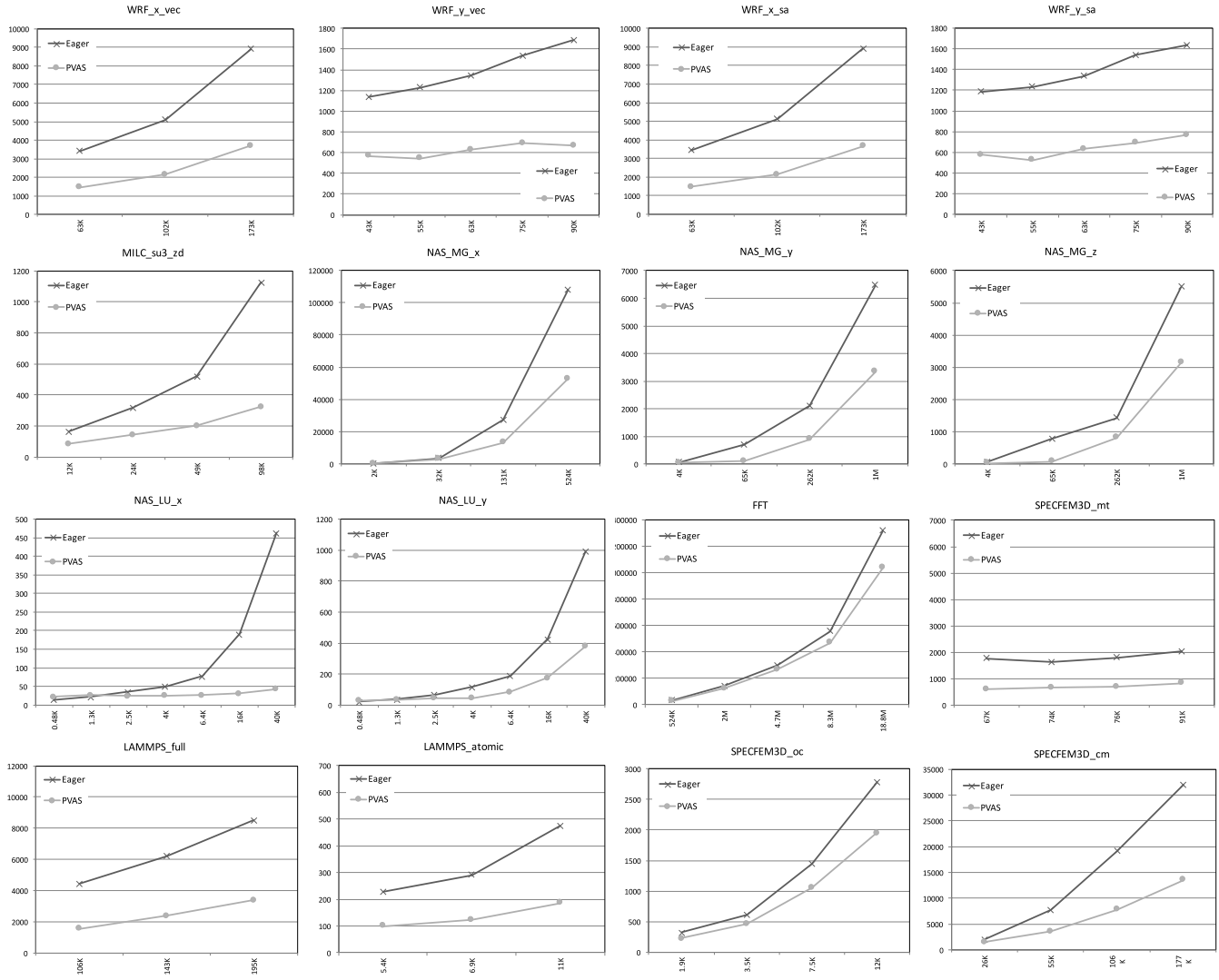

図 15　DDTBench による Eager 通信との比較（横軸：データサイズ [Bytes]，縦軸：通信遅延 [usec]）

Fig. 15　The comparison with Eager communication by using DDTBench (horizontal-axis: Data size [Bytes], vertical-axis: Latency [usec].)

除き，通信遅延が Eager 通信よりも小さくなった．通信遅延が低減するケースでは，Eager 通信と比べ，通信遅延が約 10〜80%低減された．一方，通信遅延が増加するケースでは，Eager 通信と比べ，通信遅延が約 20〜60%増加した．

Eager 通信では，送受信プロセスが同期することなく通信を実行することができる．しかし，共有メモリ上の中間バッファを経由して，データを送信バッファから受信バッファにコピーして転送する必要がある．一方，PVAS によって高速化した Rendezvous 通信では，送受信プロセスが同期する必要があるが，送信バッファから受信バッファにデータを直接コピーして転送することができる．データ転送のコストは，送受信するデータサイズが大きくなるほど高くなるため，送受信するデータサイズが大きい方が，PVAS による高速化の効果は大きくなる．送受信するデータサイズが小さい場合，送受信プロセスの同期に要する損失が，データを直接コピーして転送することができる利得を上回り，Eager 通信の方が，通信遅延が小さくなる．しかし，ある程度送受信するデータサイズが大きくなると，同期に要する損益を，データを直接コピーして転送することができる利得が上回り，PVAS によって高速化した Rendezvous 通信の方が，通信遅延が小さくなると考えられる．実際，送受信するデータサイズが小さい場合（0.48K〜2 KBytes）のみ，PVAS によって高速化した Rendezvous 通信の方が，通信遅延が大きくなっている．

既存の実装と同様，PVAS によって高速化した Rendezvous 通信についても，送受信するデータサイズが大きいケースでは Eager 通信よりも有用であるといえる．Open MPI では *Eager Threshold* のデフォルト値は 4 KBytes となっており，4 KBytes が Eager 通信と Rendezvous 通信を切り替えるデータサイズの目安になっているが，DDTBench のどの通信パターンにおいても，データサイズが 4 KBytes を超えるケースでは，PVAS によって高速化した

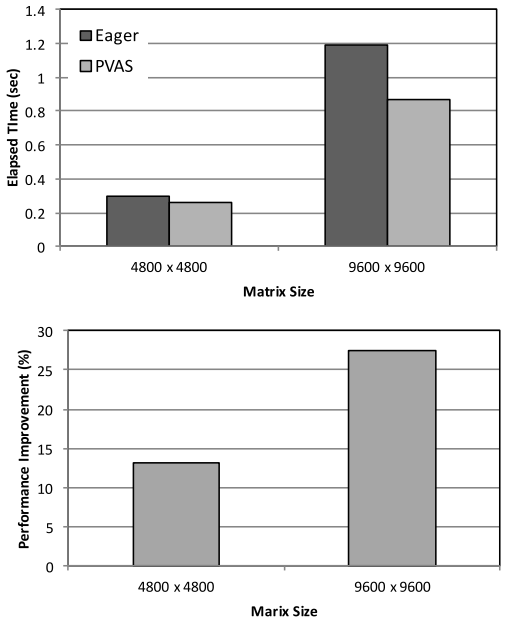

図 16　fft2d_datatype による Eager 通信との比較
Fig. 16　The comparison with Eager communication by using fft2d_datatype.

Rendezvous 通信の方が，Eager 通信よりも通信遅延が小さくなっている．

次に，fft2d_datatype によって比較を行った．ft2d_datatype では，プロセス数が 240 の場合，2 次元配列の要素数が 4,800 × 4,800 のときは 6.4 KBytes のデータを，要素数が 9,600 × 9,600 のときは 25.6 KBytes のデータを MPI プロセス間で送受信する．*Eager Threshold* を十分大きな値に設定し，通信がすべて Eager 通信で行われる設定で fft2d_datatype を実行したケースと，*Eager Threshold* を 0 にし，通信がすべて PVAS によって高速化した Rendezvous 通信で行われる設定で fft2d_datatype を実行したケースを比較した．結果を**図 16** に示す．

上部のグラフにおいて，*Eager* は通信をすべて Eager 通信で実行した場合の実行結果を，*PVAS* は通信をすべて PVAS によって高速化した Rendezvous 通信で実行した場合の実行結果を示している．下部のグラフは，*Eager* の実行性能を基準としたときの，*PVAS* の実行性能の改善率を示している．2 次元配列の要素数が 4,800 × 4,800 のときは約 13%，配列サイズが 9,600 × 9,600 のときは約 27%，PVAS によって高速化した Rendezvous 通信を用いたときの方が，実行性能が高くなった．送受信するデータサイズが大きいケースでは，PVAS によって高速化した Rendezvous 通信が有用であることが，fft2d_datatype による評価からも分かる．

6. 関連研究

6.1 MPI のスレッド実装

通常 MPI は，1 プロセスを 1 MPI プロセスとして並列計算処理を実行させるプログラミングモデルになっている．しかし，MPI の実装の中には，1 スレッドを 1 MPI プロセスとして並列計算処理を実行させるものも存在する [8], [19], [27]．同一ノード内に存在するスレッドは，同一アドレス空間で動作するため，PVAS を用いる場合と同様に，アドレス空間の境界越しにデータを送受信するオーバヘッドなしに MPI ノード内通信を実行することができる．

しかし，1 スレッドを 1 MPI プロセスとしてプログラミングを行う場合，MPI プロセス間でグローバル変数が共有されてしまい，既存の MPI のプログラミングモデルを大きく逸脱してしまうという問題がある．また，MPI ライブラリ内の管理オブジェクトが MPI プロセス間で共有されるため排他制御が必要になり，慎重に MPI ライブラリを実装しないと，排他制御のオーバヘッドで実行性能が低下してしまうという問題がある．

6.2 OS カーネルによるデータコピー

ノード内通信を行う方法の 1 つとして，OS カーネルによるデータコピーがあげられる．この方式では，OS カーネルが送信バッファから受信バッファに直接データをコピーする．OS カーネルはノード内の全プロセスのメモリにアクセスする権限を持つため，送信バッファと受信バッファを格納するメモリにアクセスすることができる．PVAS を用いる場合と同様，共有メモリを用いてデータを転送する場合に発生するオーバヘッドを回避できる．しかし，通信のたびに，OS カーネルにデータのコピーを指示するためのシステムコールを実行する必要があり，このオーバヘッドにより，送受信するデータサイズが小さいときは，共有メモリを用いる方式よりも通信遅延がかえって増加してしまうという問題がある．PVAS を用いる場合は，ユーザプログラムが送信バッファから受信バッファにデータをコピーできるので，システムコール実行のオーバヘッドは発生しない．

この方式をサポートする Cross-Memory-Attach（CMA）という機能が Linux に実装されている．また，この方式をサポートするための Linux カーネルモジュールとして，KNEM [7] と LiMIC [10] がリリースされている．Open MPI には，KNEM と CMA を用いた MPI ノード内通信も実装されている．しかし，KNEM と CMA による MPI ノード内通信がサポートするのは，連続したデータの送受信のみであり，派生データ型を用いた不連続なデータの送受信はサポートしていない．

6.3 XPMEM

XPMEM [31] は，Linux のカーネルモジュールで，プロセスが他のプロセスの使用しているメモリを，自身のアドレス空間にマッピングする機能を提供する．送信プロセスの送信バッファが格納されているメモリを受信プロセス

が自身のアドレス空間にマッピングすることで，送信バッファから受信バッファに直接データをコピーすることができるようになる．PVAS を用いる場合と同様に，共有メモリ上の中間バッファを経由してデータの送受信を行う際のオーバヘッドを回避できる．しかし，メモリマッピングを行うためにシステムコールを実行する必要があり，これがオーバヘッドとなる．同じ通信バッファを用いて繰り返し通信を行う場合，メモリマッピングを行うためのシステムコールを実行するのは最初の通信のときだけでよいが，通信バッファを頻繁に新規作成して用いるケースでは，そのつどシステムコールを実行する必要が生じる．

Open MPI は，XPMEM を用いて送信バッファから受信バッファに直接データをコピーするノード内通信の実装もサポートしている．しかし，これが適用されるのは連続したデータを送受信する場合のみである．派生データ型を用いて不連続なデータの送受信を行う場合，XPMEM で作成した送受信プロセスの双方がアクセス可能なメモリ領域に中間バッファを作成し，共有メモリの実装と同様の方式でデータの転送が行われる．

6.4 SMARTMAP

SMARTMAP [3] は，プロセスが他のプロセスのメモリを自身のアドレス空間にマッピングする機能を提供する．アドレス空間の先頭から 512 GBytes をプロセスが使用可能なアドレス空間とし，残りのアドレス空間は，自プロセスを含む同一ノード内のプロセスのメモリをマッピングするために用いられる．実装については文献 [3] に詳しい．

PVAS を用いる場合と同様に，送信プロセスの送信バッファから受信プロセスの受信バッファに直接データをコピーすることが可能になる．SMARTMAP を用いた MPI ノード内通信が提案されているが，これが適用されるのは送信データと受信データの双方が連続したデータである場合のみであり，派生データ型を用いて不連続なデータを送受信するケースは考慮されていない．

6.5 Hybrid MPI

高速なノード内通信を可能にする MPI ライブラリの実装として Hybrid MPI [6], [30] が提案されている．Hybrid MPI では，既存の malloc ライブラリを独自の malloc ライブラリに置き替え，MPI アプリケーションに共有メモリ領域上のメモリプールからメモリを確保させる．通信バッファを動的に確保するケースでは，通信バッファが共有メモリとして確保される．よって，PVAS を用いる場合と同様に，送信プロセスの送信バッファから受信プロセスの受信バッファにデータを直接コピーして転送することが可能になり，MPI ノード内通信を高速化することができる．ただし，通信バッファをグローバル変数として静的に確保するようなケースでは，MPI ノード内通信を高速化することはできない．また，文献 [6], [30] では，派生データ型を用いて不連続なデータを送受信するケースについては言及されていない．

6.6 User-mode Memory Registration

Mellanox®の Infiniband HCA がサポートする User-mode Memory Registration（UMR）[15] という機能を用いると，異なるノード上で動作するプロセス間の不連続なデータの送受信を 1 回の Remote Direct Memory Access（RDMA）によって実行することができる．UMR を利用して派生データ型を用いる MPI ノード間通信を高速化する実装が提案されている [14]．通常，不連続なデータの送受信を異なるノード上で動作する MPI プロセス間で行う場合，送信プロセスが不連続なデータを連続したデータに pack してから RDMA で受信プロセスに送信する．データを受け取った受信プロセスは，pack された連続データを不連続なデータへと unpack する処理を行う．UMR を用いると，この pack/unpack の処理が必要なくなるため，高速に不連続なデータを送受信することができる．UMR を用いる場合，送受信プロセスが互いのデータ型の情報を通信時に交換する必要がある．データ型の情報を交換する処理のオーバヘッドを最小限にするため，各 MPI プロセスは通信先のデータ型の情報を自身の MPI ライブラリ内にキャッシュしておき，同じデータ型を用いる通信が発生した際に再利用する．UMR によって高速化した MPI ノード間通信と本研究で提案する MPI ノード内通信を組み合わせることで，派生データ型を用いて不連続なデータの送受信を行う MPI プログラムの実行性能を大きく向上させることができる．

7. まとめと今後の課題

近年，HPC システムを構成するノード 1 台あたりのコア数は飛躍的に増加してきている．PVAS は，このようなシステムにおいて，より効率的な並列処理を実行することを可能にする新たなタスクモデルである．PVAS を用いると，並列処理を実行するノード内のプロセスを同一アドレス空間で動作させることが可能になるため．アドレス空間の境界越しにデータを送受信するためのオーバヘッドなしに，ノード内通信を実行できる．

本研究では PVAS を，派生データ型を用いる MPI ノード内通信に適用し，メモリ上で不連続なデータを高速に送受信することを可能にした．PVAS を用いてノード内の MPI プロセスを同一アドレス空間で動作させ，送信プロセスの送信バッファから受信プロセスの受信バッファにデータを直接コピーして転送することを可能にし，通信遅延を低減させた．

PVAS によって高速化した MPI ノード内通信を Open MPI に実装し，代表的なメニーコアプロセッサである

Intel® Xeon Phi™ を用いて評価した．DDTBench によって不連続なデータの送受信の通信遅延を測定したところ，PVAS によって高速化した MPI ノード内通信の実装は，共有メモリを用いる既存の実装と比べて，一部の通信パターンを除き，通信遅延が低減されることが分かった．また，派生データ型を用いて不連続なデータの送受信を行うミニアプリケーションによって評価したところ，最大で約 21%実行性能を改善することができた．

本研究では評価を 1 台のノードで行った．複数のノードを用い，ノード内通信だけでなくノード間通信が発生するケースでの評価も今後行っていく予定である．また，PVAS を用いて MPI の集団通信を高速化することも計画している．MPI の集団通信において，同一ノード内の MPI プロセスをグループ化し，グループの代表プロセスが他のノードの代表プロセスと通信を行うことで，ノード間でのデータの送受信の回数を抑制する最適化が考えられる．この最適化において，グループの代表プロセスが他のノードの代表プロセスから受信したデータをノード内のプロセスに送信する処理や，代表プロセスがノード内のプロセスからデータを集めて他のノードの代表プロセスに送信する処理を，PVAS によって高速に実行することができると考える．

謝辞　本研究の一部は，科学技術振興機構（JST）の戦略的創造研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」によるものである．また，本研究を推進するにあたり多数の有用なコメントをいただいたテネシー大学の George Bosilca 教授に深く感謝する．

参考文献

[1] Berkeley UPC – Unified Parallel C: The UPC Language, available from ⟨http://upc.lbl.gov/⟩.

[2] Bernard, C., Ogilvie, M.C., Degrand, T.A., Detar, C.E., Gottlieb, S.A., Krasnitz, A., Sugar, R. and Toussaint, D.: Studying Quarks and Gluons On Mimd Parallel Computers, *Int. J. High Perform. Comput. Appl.*, Vol.5, No.4, pp.61–70 (online), DOI: 10.1177/109434209100500406 (1991).

[3] Brightwell, R., Pedretti, K. and Hudson, T.: SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor, *Proc. 2008 ACM/IEEE Conference on Supercomputing*, Piscataway, NJ, USA, (online), available from ⟨http://dl.acm.org/citation.cfm?id=1413370.1413396⟩ (2008).

[4] Bailey, D.H. et al.: The NAS Parallel Benchmarks, *International Journal of Supercomputer Applications*, Vol.5, No.3 (1991).

[5] Deitz, S.J., Chamberlain, B.L. and Hribar, M.B.: Chapel: Cascade High-Productivity Language An Overview of the Chapel Parallel Programming Model, *Cray User Group*, Lugano, Switzerland (2006).

[6] Friedley, A., Bronevetsky, G., Hoefler, T. and Lumsdaine, A.: Hybrid MPI: Efficient Message Passing for Multi-core Systems, *Proc. 2013 ACM/IEEE Conference on Supercomputing*, NY, USA, (online), DOI: 10.1145/2503210.2503294 (2013).

[7] Goglin, B. and Moreaud, S.: KNEM: A Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework, *Journal of Parallel and Distributed Computing* (*JPDC*), Vol.73, No.2, pp.176–188 (online), DOI: 10.1016/j.jpdc.2012.09.016 (2013).

[8] Huang, C., Lawlor, O. and Kalé, L.V.: Adaptive MPI, *Proc. 16th International Workshop on Languages and Compilers for Parallel Computing* (*LCPC 2003*), LNCS 2958, College Station, Texas, pp.306–322 (2003).

[9] Intel Corporation: Intel Manycore Platform Software Stack, available from ⟨https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss⟩.

[10] Jin, H.-W., Sur, S., Chai, L. and Panda, D.K.: LiMIC: Support for High-Performance MPI Intra-node Communication on Linux Cluster, *ICPP*, pp.184–191 (2005).

[11] Jinpil, L. and Mitsuhisa, S.: Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems, *The 39th International Conference on Parallel Processing Workshops*, *ICPPW10* (2010).

[12] Kemal, E., Vijay, S. and Vivek, S.: X10: Programming for hierarchical parallelism and non-uniform data access, *International Workshop on Language Runtimes*, *OOPSLA* (2004).

[13] Komatitsch, D. and Tromp, J.: Modeling of Seismic Wave Propagation at the Scale of the Earth on a Large Beowulf, *Proc. 2001 ACM/IEEE Conference on Supercomputing*, *SC '01*, p.42, ACM (online), DOI: 10.1145/582034.582076 (2001).

[14] Li, M., Subramoni, H., Hamidouche, K., Lu, X. and Panda, D.K.: High Performance MPI Datatype Support with User-mode Memory Registration: Challenges, Designs and Benefits, *IEEE Cluster*, Chicago, IL, USA (2015).

[15] Mellanox: RDMA Aware Networks Programming User Manual, available from ⟨http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf⟩.

[16] MPI: A Message-Passing Interface Standard Version 3.1, available from ⟨http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf⟩.

[17] MPICH: High-Performance Portable MPI, available from ⟨http://www.mpich.org/⟩.

[18] Open MPI: Open Source High Performance Computing, available from ⟨http://www.open-mpi.org/⟩.

[19] Pérache, M., Carribault, P. and Jourdren, H.: MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption, *PVM/MPI*, Ropo, M., Westerholm, J. and Dongarra, J. (Eds.), Lecture Notes in Computer Science, Vol.5759, pp.94–103, Springer (2009).

[20] Plimpton, S.: Fast Parallel Algorithms for Short-range Molecular Dynamics, *J. Comput. Phys.*, Vol.117, No.1, pp.1–19 (online), DOI: 10.1006/jcph.1995.1039 (1995).

[21] Rico-Gallego, J.-A. and Díaz-Martín, J.-C.: Performance Evaluation of Thread-based MPI in Shared Memory, *Proc. 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, *EuroMPI '11*, Berlin, Heidelberg, pp.337–338, Springer-Verlag (online), available from ⟨http://dl.acm.org/citation.cfm?id=2042476.2042518⟩ (2011).

[22] Scalable Parallel Computing Laboratory: MPI Derived Datatype (Benchmark) Page, available from ⟨http://spcl.inf.ethz.ch/Research/Parallel_

Programming/MPI_Datatypes/⟩.
[23] Schneider, T., Gerstenberger, R. and Hoefler, T.: Micro-Applications for Communication Data Access Patterns and MPI Datatypes, *Proc. Recent Advances in the Message Passing Interface – 19th European MPI Users' Group Meeting, EuroMPI 2012*, Vienna, Austria, September 23–26, 2012, Vol.7490, pp.121–131, Springer (2012).
[24] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: PGAS Intra-node Communication towards Many-Core Architecture, *6th Conference on Partitioned Global Address Space Programing Model*, Santa Barbara, California, USA (2012).
[25] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: Proposing A New Task Model towards Many-Core Architecture, *Proc. ACM International Workshop on Many-core Embedded Systems 2013*, Tel Aviv, Israel, ACM (2013).
[26] Skamarock, W.C. and Klemp, J.B.: A Time-split Nonhydrostatic Atmosphericx Model for Weather Research and Forecasting Applications, *J. Comput. Phys.*, Vol.227, No.7, pp.3465–3485 (online), DOI: 10.1016/j.jcp.2007.01.037 (2008).
[27] Tang, H. and Yang, T.: Optimizing Threaded MPI Execution on SMP Clusters, *Proc. 15th International Conference on Supercomputing, ICS '01*, pp.381–392, ACM (online), DOI: 10.1145/377792.377895 (2001).
[28] The University of Tennessee Inovative Computing Laboratory: Performance Application Programming Interface, available from ⟨http://icl.cs.utk.edu/papi/⟩.
[29] TOP 500 Lists: TOP 500 Supercomputer Sites, available from ⟨http://www.top500.org⟩.
[30] Wickramasinghe, U.S., Bronevetsky, G., Lumsdaine, A. and Friedley, A.: Hybrid MPI: A Case Study on the Xeon Phi Platform, *Proc. 4th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS '14*, pp.6:1–6:8, ACM (online), DOI: 10.1145/2612262.2612267 (2014).
[31] Woodacre, M., Robb, D., Roe, D. and Feind, K.: The SGI AltixTM 3000 Global Shared-Memory Architecture.
[32] 島田明男，堀　敦史，石川　裕：新しいタスクモデルによるメニーコア環境に適した MPI ノード内通信の実装，情報処理学会論文誌コンピューティングシステム（ACS），Vol.8, No.2, pp.36–54 (2015).

島田 明男　（正会員）

2006 年慶應義塾大学理工学部情報工学科卒業．2008 年同大学大学院理工学研究科修士課程修了．同年株式会社日立製作所入社．2012 年理化学研究所計算科学研究機構出向．2015 年より日立製作所に復帰．同年慶應義塾大学大学院理工学研究科博士課程入学．システムソフトウェア，ストレージ，並列システム等に興味を持つ．

須藤 敦之　（正会員）

2000 年東京工業大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．同年（株）日立製作所中央研究所入社．その後，同社システム開発研究所，同社横浜研究所に勤務．2016 年より日立ヨーロッパ（株）勤務．ネットワークストレージ，オペレーティングシステムの研究開発に従事．

堀 敦史　（正会員）

1979 年早稲田大学理工学部電気工学科卒業．1981 年同大学大学院理工学研究科修士課程修了．同年株式会社三菱総合研究所に入社．1992 年技術研究組合新情報処理開発機構に出向．1999 年東京大学より博士（工学）の学位を取得．2001 年株式会社スイミーソフトウェア設立．2004 年英国 Allinea Software 社に移籍．2008 年東京大学情報基盤センター特任教授．2010 年より理化学研究所計算科学研究機構上級研究員．並列システムソフトウェアの研究に興味を持つ．

石川 裕　（正会員）

1987 年慶應義塾大学大学院工学研究科電気工学専攻博士課程修了．工学博士．同年電子技術総合研究所（現，産業技術総合研究所）入所．1993 年技術研究組合新情報処理開発機構出向．2002 年より東京大学大学院情報理工学系研究科コンピュータ科学専攻教授．2010 年より同大学情報基盤センターセンター長兼務．2014 年より理化学研究所計算科学研究機構プロジェクトリーダ，チームリーダ．

河野 健二　（正会員）

1993 年東京大学理学部情報科学科卒業．1997 年同大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．博士（理学）．電気通信大学情報工学科講師等を経て，現在，慶應義塾大学理工学部情報工学科教授．2000 年度情報処理学会山下記念研究賞，1999，2008，2009，2012 年度情報処理学会論文賞，2014 年日本ソフトウェア科学会ソフトウェア論文賞，2015 年 IBM Faculty Award 受賞．オペレーティングシステム，システムソフトウェア，ディペンダブルシステム等に興味を持つ．IEEE，ACM，USENIX 各会員．