メニーコア環境におけるノード内通信の高速化および効率化を実現するタスクモデルの研究

2017 年度

島 田 明 男

メニーコア環境におけるノード内通信の高速化および効率化を実現するタスクモデルの研究

島田 明男

慶應義塾大学大学院理工学研究科
開放環境科学専攻

博士（工学）の学位申請論文

2017 年度

メニーコア環境におけるノード内通信の高速化および効率化を実現するタスクモデルの研究

島田明男

論文要旨

近年，電力効率の観点から，コア単体の処理性能を向上させるよりも，コア数を増加させる CPU アーキテクチャが一般的になっており，High-performance Computing (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．一方，1 コアあたりのメモリ量は減少する傾向にある．このようなメニーコア環境では，同一ノード内で動作する並列プロセスの数が従来のマルチコア環境よりも増加する．よって，ノード内通信（同一ノード内の並列プロセス間の通信）の発生回数が，従来よりも多くなる．ノード内通信の性能が，並列アプリケーションの性能に従来よりも大きな影響を与えることになり，より高速なノード内通信が求められる．また，1 コアあたりのメモリ量は減少する傾向にあるため，ノード内通信に掛かるメモリ消費量の低減も求められる．しかし，従来のマルチコア環境向けのノード内通信では，並列プロセスのアドレス空間をまたいでデータ転送するために，通信遅延の増加やノード内通信に掛かるメモリ消費量の増加をまねいてしまう問題がある．

そこで，本論文は，メニーコア環境において高速かつ効率的なノード内通信を実現するため，Partitioned Virtual Address Space (PVAS) と呼ぶ新たなタスクモデルを提案する．PVAS タスクモデルは，並列処理を行うノード内の並列プロセス群を同一アドレス空間で動作させることを可能にする．並列プロセスを同一アドレス空間で動作させることで，アドレス空間をまたいでデータ転送することなく，ノード内通信を実行することが可能になり，高速かつメモリ消費量の少ないメニーコア環境向けのノード内通信を実現することができる．PVAS タスクモデルを Linux カーネルのメモリ管理を対象に実装した．

本論文では，PVAS タスクモデルを Message Passing Interface (MPI) の通信に適用し，PVAS タスクモデルの検証を行う．PVAS タスクモデルを MPI 通信に適用し，高速かつメモリ消費量の少ない，よりメニーコア環境に適した MPI ノード内通信を実現する．PVAS の適用は，MPI における連続データの送受信と不連

続データの送受信を対象にして行う．実装は，MPI の Open Source Software (OSS) 実装の一つである Open MPI に，PVAS を利用するメニーコア環境向けのノード内通信モジュールを組み込むことで行う．これを，マイクロベンチマークとミニアプリケーションにより評価し，マルチコア環境向けの MPI ノード内通信モジュールを用いた場合と比較する．マイクロベンチマークによる評価では，連続データの送受信において，通信遅延が大きく改善することを示す．また，不連続データの送受信においても，小さなデータが多数不連続にメモリ上に配置される通信パターンを除き，通信遅延が大きく改善することを示す．連続データを送受信するミニアプリケーションによる評価では，最大で約 18%，実行性能が改善することを示す．不連続データを送受信するミニアプリケーションによる評価では，最大で約 21%，実行性能が改善することを示す．また，マイクロベンチマークによって MPI ノード内通信のメモリ消費量を測定し，最大で約 18%，メモリ消費量を削減可能であることを示す．これらの評価により，本論文で提案する PVAS タスクモデルによって，メニーコア環境において高速かつ効率的なノード内通信を実現可能であることを示す．

A Study on Task Models for High-performance and Efficient Intra-node Communication in Many-core Environments

Akio Shimada

Abstract

Having reached the evolutionary limits of single-core performance in terms of power-efficiency, increasing the number of cores being incorporated into processors is becoming popular in recent years. Thus, the number of cores in High-performance Computing (HPC) system is growing rapidly. Meanwhile, per-core memory size is becoming smaller. In such many-core environments, the number of parallel processes running in a node becomes larger. Therefore, the number of times of intra-node communications (communications among parallel processes in the same node) taking places in a node becomes larger than that in multi-core environments. Then, the performance of intra-node communication gives a greater impact on parallel application performance. As a result, high-performance intra-node communication is required. Moreover, reducing memory footprint for intra-node communication is required because per-core memory size becomes smaller in many-core environments. However, in case of intra-node communication for multi-core environments, transmitting data crossing address space boundaries among parallel processes results in large communication latency and large memory footprint.

In this thesis, Partitioned Virtual Address Space (PVAS), which is a new task model for achieving high-performance and efficient intra-node communication in many-core environments, is proposed. PVAS task model enables parallel processes in the same node to run in the same address space. Parallel processes running in the same address space can perform intra-node communication without crossing address space boundaries. As a result, the intra-node communication for many-core environments, which is high-performance and whose memory usage is small, can be achieved. PVAS task model was implemented by modifying the memory management of Linux kernel.

In this thesis, the effectiveness of PVAS task model is verified by applying it to Message Passing Interface (MPI) communication. MPI intra-node communication for many-core environments, which is high-performance and whose memory usage is small,

can be achieved by utilizing PVAS task model. PVAS task model is applied to MPI intra-node communications for transmitting both contiguous and non-contiguous data. Intra-node communication module for many-core environments, which utilizes PVAS task model, has been implemented into Open MPI (Open MPI is one of the open source MPI implementations) and compared to existing intra-node communication module for multi-core environments by micro-benchmarks and mini-applications. Micro-benchmark results show that PVAS task model accelerates MPI intra-node communication for transmitting contiguous data and accelerates MPI intra-node communication for transmitting non-contiguous data excluding several communication patterns, which sends a large number of small data discontinuously stored on the memory. PVAS task model improves the performance of mini-application performing contiguous data transfer by up to 18%, and the performance of mini-application performing non-contiguous data transfer by up to 21%. Moreover, micro-benchmark results show that PVAS task model can reduce memory footprint for MPI intra-node communication by up to 18%. These results shows that PVAS task model achieves high-performance and high-efficient intra-node communication in many-core environments.

目次

図目次

表目次

第1章　序論

1.1　研究の背景

1.1.1　HPC システムの動向

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．マルチコアプロセッサのコア数は年々増加する傾向にある．また，数 100 の論理コアを搭載するメニーコアプロセッサを採用する HPC システムの数が増加してきている[1]．ノード 1 台あたりのコア数が増加する一方で，1 コアあたりのメモリ量は減少する傾向にある[2]．次世代 HPC システムでの採用が期待される Hybrid Memory Cube[3]のような高性能メモリは容量が少なく，多数のコアが存在する場合，1 コアに割り振ることができるメモリ量が制限される．表 1-1 は，Intel 社のメニーコア製品である Xeon Phi プロセッサの製品仕様を示している．Xeon Phi プロセッサは通常のメインメモリの他に，プロセッサが高速にアクセス可能な内部メモリを備えるが，1 論理コアあたりの内部メモリ量は，最小で 57 MB となる．また，PCIe カードデバイスとして提供されるコプロセッサ型のメニーコアプロセッサは，搭載するメインメモリの量が制限される．表 1-2 は，Intel 社のコプロセッサ製品である Xeon Phi コプロセッサの製品仕様となっており， 1 論理コアあたりのメインメモリ量は最小で約 27MB となる．

1.1.2　メニーコア環境における並列処理

1 プロセッサあたりのコア数が 1 のシングルコア環境の場合，HPC システムを構成する各ノードに 1 つのプロセスを起動し，並列に計算処理を実行させる．このようなプロセスを並列プロセスと呼ぶ．並列プロセスは，処理を実行するために必要なデータをノード間通信によって互いに送受信し，並列計算を実行する．このようなプロセスレベルでの並列化をサポートする並列化モデルとして，Message Passing Interface (MPI)[4]や PGAS 言語[5-8]がある．

表 1-1 Xeon Phi プロセッサ

型番	7290 系	7250 系	7230 系	7210 系
コア数	72	68	64	64
論理コア数	288	272	256	256
内部メモリ	16 GB	16 GB	16 GB	16 GB
内部メモリ/論理コア	57 MB	60 MB	64 MB	64 MB

表 1-2 Xeon Phi コプロセッサ

型番	7100 系	5100 系	3100 系
コア数	61	60	57
論理コア数	244	240	228
メインメモリ	16 GB	8 GB	6 GB
メインメモリ/論理コア	67 MB	34 MB	27 MB

1 プロセッサあたりのコア数が数個から 10 数個のマルチコア環境の場合, ノード間の並列化だけではなく，ノード内の並列化も考慮する必要がある．本論文では，ノード間およびノード内ともにプロセスレベルで並列化する手法をピュア型の並列化，ノード間はプロセスレベルで並列化し，ノード内はスレッドレベルで並列化する並列化手法をハイブリッド型の並列化と呼ぶ．ピュア型の並列化では，同一ノード上に複数の並列プロセスを起動して並列処理を実行させる．同一ノード上の並列プロセスはノード内通信を実行して並列処理に必要な計算データを送受信する．ハイブリッド型の並列化では，ノード上の並列プロセス内に複数のスレッドを起動して並列処理を実行させる．スレッドレベルの並列化をサポートする並列化モデルとして, OpenMP[32]が広く用いられている．

ピュア型の並列化では，並列アプリケーションの実行時にノード内通信によるオーバヘッドが発生する．対して，ハイブリッド型の並列化では，スレッド同士が同一アドレス空間で動作するため，ノード内通信を実行する必要が無い．しかし, ハイブリッド型の並列化には, メモリアクセスのローカリティ低下や, 異なるレベルの並列化が混在することによって性能チューニングが困難になる等の問題があり，ピュア型よりも常に有利であるとは限らないことが報告されている[41-43]．そこで，マルチコアプロセッサの登場以来，ノード内通信を高速化することにより，ピュア型の並列アプリケーションの性能を向上させる研究が盛んになっている．代表例として，OS カーネルの支援によりノード内通信を高速化する KNEM[14]/LiMIC[16]や，プロセス間のメモリマッピングにより

ノード内通信を高速化する XPMEM[25]が挙げられる．

1 プロセッサあたりのコア数が数 10 から数 100 となるメニーコア環境の場合，ピュア型の並列化において，従来のマルチコア環境よりも，より多くの並列プロセスがノード上で動作することになり，並列アプリケーションの実行時に発生するノード内通信の発生回数が多くなる．よって，メニーコア環境でピュア型の並列アプリケーションを実行する場合，ノード内通信をより重要視する必要がある．具体的には，並列アプリケーションの実行性能を向上させるため，より高速なノード内通信がメニーコア環境では求められる．また，メニーコア環境では，ノード内通信のメモリ消費量の低減も求められる．ノード内通信によるメモリ消費量は，通信を行う同一ノード内の並列プロセスの数に伴い増加する．しかし，メニーコア環境では，同一ノード内の並列プロセスの数が従来よりも増加する反面，コアあたりのメモリ量は少なくなる傾向にある．このような環境で並列アプリケーションにより多くのメモリを割り当てるためには，ノード内通信のメモリ消費量を低減する必要がある．

1.2 提案と目標

そこで本研究は，ピュア型の並列アプリケーションをメニーコア環境において高速かつ効率的に実行するための新たなタスクモデルとして Partitioned Virtual Address Space (PVAS)[9-12]を提案する．PVAS タスクモデルは並列処理を行うノード内の並列プロセス群を同一アドレス空間で動作させることを可能にする．既存のタスクモデルでは計算処理を実行するノード内の各並列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，既存のタスクモデルを前提とするマルチコア環境向けのノード内通信では，アドレス空間をまたいでデータ転送するために，通信遅延の増加やメモリ消費量の増加をまねいてしまう．対して，PVAS タスクモデルによって同一アドレス空間で動作する並列プロセス同士は互いのメモリに直接アクセスすることができるため，アドレス空間をまたいでデータ転送することなく，ノード内通信を実行することができる． PVAS タスクモデルによって，高速かつメモリ消費量の少ない，よりメニーコア環境に適したノード内通信を実現することを本研究の目標とする．

1.3 検証方針

PVAS タスクモデルは MPI や一部の PGAS 言語のような，プロセスレベルでの並列化を実現す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，PVAS タスクモデルを MPI の通信に適用し，メニーコア環境上で評価することで，PVAS タスクモデルの効果の検証を行う．

MPI 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 は複数のプロセスを起動し，並列処理を実行させる並列化モデルになっている．並列処理を行なうプロセス同士は，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，互いに送受信することができる．MPI で実装された並列アプリケーションを MPI アプリケーション，MPI アプリケーションを実行する並列プロセスを MPI プロセスと呼ぶ．

MPI 通信は，異なるノード上で動作する MPI プロセス同士の通信である MPI ノード間通信と，同一ノード上で動作する MPI プロセス同士の通信である MPI ノード内通信に分けられる．マルチコアプロセッサの登場以来，様々な方式の MPI ノード内通信が提案されている[23-24][26-27][36-17]．また，メニーコア環境において，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[13]．

本研究では，PVAS タスクモデルを用いて，並列処理を行うノード内の MPI プロセスを同一アドレス空間で実行し，アドレス空間をまたいでデータ転送することなく，MPI ノード内通信を実行することを可能とする．これにより，高速かつメモリ消費量の少ない，よりメニーコア環境に適した MPI ノード内通信を実現する．MPI 通信は，連続したデータを送受信するための通信と不連続なデータをひとまとめにして送受信するための通信をサポートしている．本研究では，連続データを送受信するための MPI ノード内通信と不連続データを送受信するための MPI ノード内通信の双方に PVAS タスクモデルを適用する．

PVAS タスクモデルを利用したメニーコア環境向け MPI ノード内通信を主要な MPI ランタイムの一つである Open MPI[14]に実装し，既存のマルチコア環境向け MPI ノード内通信と比較することで，PVAS タスクモデルの有用性を検証する．まず，マイクロベンチマークによって，MPI ノード内通信の通信性能を評価する．次に，MPI で実装されたミニアプリケーションにより，通信と計算処理を含めた総合的な実効性能の評価を行う．性能の評価は，連続データを送受信する場合と，不連続データを送受しする場合の双方に対して実施する．また，マイクロベンチマークとミニアプリケーションを用いて，MPI ノード内通

信に掛かるメモリ消費量の評価を実施する．

1.4 本論文の構成

本論文の構成は以下の通りである．次章にて既存のマルチコア環境向けノード内通信の仕組みと，その他の関連研究について説明する．第 3 章では，PVAS タスクモデルの概要と設計，PVAS タスクモデル上でのノード内通信について述べ，第 4 章で PVAS タスクモデルの実装について述べる．第 5 章で PVAS タスクモデルを用いた MPI ノード内通信の実装について説明する．第 6 章で評価について述べる．第 7 章にて考察について述べ，最後に本研究の結論を述べる．

第2章　既存方式と関連研究

本章では，まず，代表的な既存のマルチコア環境向けノード内通信の方式とその問題点について述べる．そして，関連研究として，プロセス間通信を対象とした研究について述べる．

2.1　既存のノード内通信

本節では，既存のマルチコア環境向けノード内通信の方式とその問題点について述べる．代表的な既存のマルチコア環境向けノード内通信として，パイプ/ソケット，共有メモリ，KNEM/LiMIC，XPMEM が挙げられる．

これらでは，アドレス空間越しにデータ転送するために，通信遅延の増加とメモリ消費量の増加をまねいてしまう．具体的には，以下の問題により，通信遅延の増加とメモリ消費量の増加をまねく．

- 通信遅延の増加
  - 中間バッファを経由するデータ転送によるメモリコピー回数の増加
  - システムコールの実行によるオーバヘッド
- メモリ消費量の増加
  - 中間バッファの確保
  - メモリマッピングによるページテーブルの肥大化

表 2-1 各方式の特徴

方式	メモリコピー回数	システムコールオーバヘッド	中間バッファ	ページテーブル肥大化
パイプ/ソケット	２度以上	発生する	必要	発生しない
共有メモリ	２度以上	発生しない	必要	発生する
KNEM/LiMIC	１度	発生する	不要	発生しない
XPMEM	１度	発生する	不要	発生する
PVAS	１度	発生しない	不要	発生しない

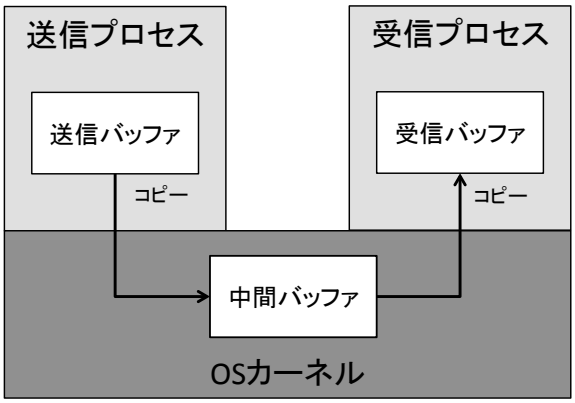


図 **2-1** パイプソケットによるノード内通信

各方式の特徴を表 3-1 にまとめた．表には，本論文で提案する PVAS タスクモデルを用いるノード内通信についても示した．以下，各既存方式について説明する．PVAS タスクモデルについては，第 3 章で説明する．

2.1.1 パイプ/ソケット

代表的なノード内通信の方式の 1 つとして, パイプとソケットがあげられる．図 2-1 に，パイプとソケットによるノード内通信の概要を示す．パイプやソケットを用いる通信では，OS カーネルが提供するシステムコールを用いて送信プロセスが送信バッファから OS カーネル内の中間バッファにデータをコピーする．そして, 受信プロセスが OS カーネル内の中間バッファからデータをコピーすることで，データを送受信する．中間バッファを経由してデータ転送するため，2 度以上のメモリコピーが必要になる．また，システムコールを実行する際に OS カーネルへのコンテキスト切り替えが必要になり，そのオーバヘッドによって通信遅延が増加する．通信のために中間バッファを確保する必要があり，その

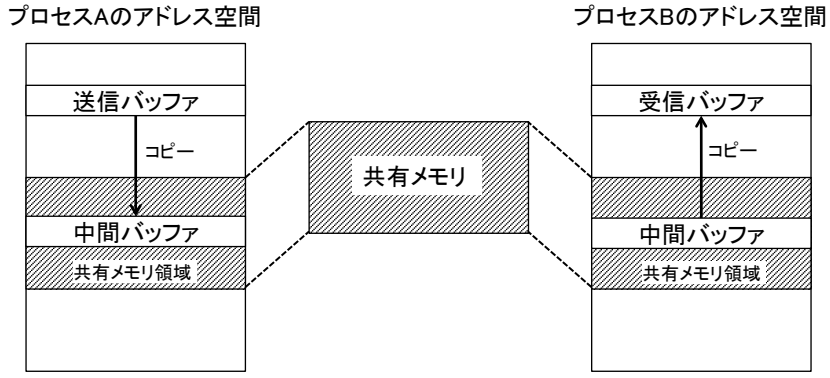


図 **2-2** 共有メモリによるノード内通信

分メモリ消費量が増加する.

2.1.2 共有メモリ

HPC 分野で最も一般的なノード内通信の方式として共有メモリによるノード内通信が挙げられる．この方式では，共有メモリ領域上の中間バッファを経由してデータを送受信する．図 2-2 に示すように，共有メモリを作成し，通信を実行するプロセスのアドレス空間にマッピングすることで共有メモリ領域を設ける. 通信を実行するときは, 共有メモリ領域に通信用の中間バッファを作成し，送信側のプロセスが送信バッファから中間バッファにデータをコピーする．そして，受信側のプロセスが，中間バッファからデータを受信バッファにデータをコピーすることで, データを送受信する. ソケット/パイプによる通信と同様,

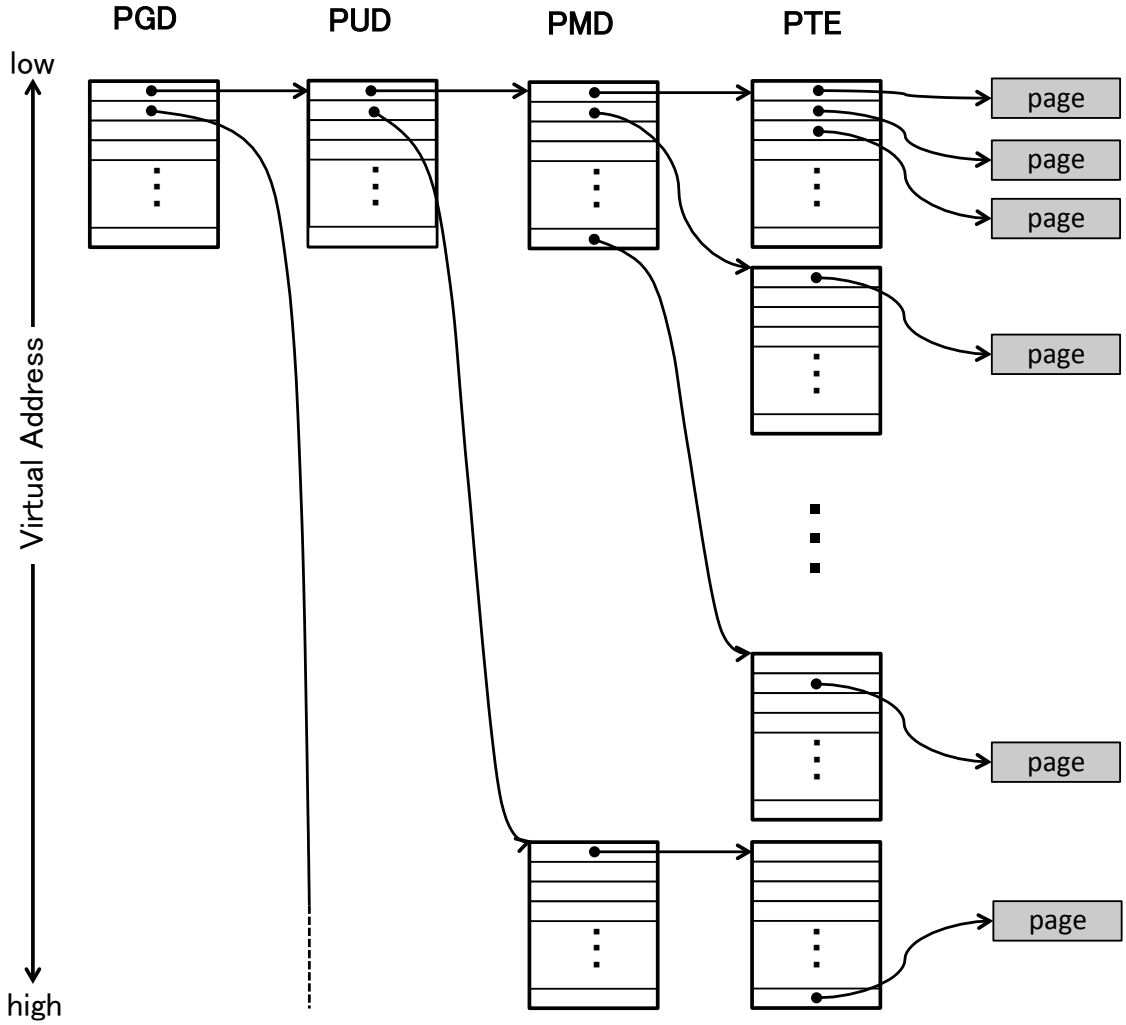


図 **2-3 x86_64** アーキテクチャのページテーブルの構造

データ転送の際に 2 度以上のメモリコピーが発生し，通信遅延が大きくなる．

また，中間バッファを確保する必要があるため，その分メモリ消費量が増加する．さらに，共有メモリによるノード内通信では，共有メモリをプロセスのアドレス空間にマッピングするために，ページテーブルが肥大化し，メモリ消費量が増加する．仮想記憶をサポートする近年の OS では，メモリをページという単位に区切り管理している．各プロセスがどのメモリページを自身のアドレス空間のどの位置にマッピングしているかという情報は，ページテーブルという OS カーネル内の管理テーブルに記録される．CPU はこのページテーブルを参照し，現在実行しているプロセスが使用しているメモリのアドレスを物理メモリ上のアドレスに変換する．

共有メモリを用いたノード内通信では，通信を行なう双方のプロセスのアドレス空間に共有メモリをマッピングし，マッピングした共有メモリを経由してデータの送受信を行う．プロセスがマッピングする共有メモリが増加する場合，マッピング情報を記録するページテーブル内のエントリ（ページテーブルエントリ）の数も併せて増加するためにページテーブルが肥大化し，ページテーブルによるメモリ消費量が増加する．

同一ノード内で動作する N 個のプロセスが全対全の通信を行う場合，N 個のプロセスが(N-1)個の通信先を持つことになる．N 個のプロセスが，(N-1)個の通信先が使用している共有メモリをマッピングするので，システム全体としては，N^2 のオーダのメモリマッピングが生成される．よって，システム全体のページテーブルによるメモリ消費量も N^2 のオーダで増加する．1 ノードあたりのコア数および並列プロセス数は今後も増加していくことが予想されるため，N^2 のオーダで増加するページテーブルのメモリ消費量は，メニーコア環境では大きな問題となる．

ページテーブルがフラットな構造をとる場合，メモリページをマッピングしていないアドレス領域についても，ページテーブルエントリを持つ必要があり，ページテーブルサイズは必要以上に大きくなってしまう．そこで多くの CPU アーキテクチャでは，階層構造のページテーブルをサポートしている．例えば，x86_64 アーキテクチャのページテーブルは，図 2-3 に示す通り，4 段階(PGD/PUD/PMD/PTE)の階層構造となっている．各階層のテーブルは 512 のエントリをもっており，256 TB のアドレス空間を取り扱うことができる．4 階層目のテーブルである PTE は，1 つで最高 512 個のメモリページをマッピングすることができる．連続したアドレス領域にメモリページをマッピングする場合は，必要な PTE の数は少なくなるが，メモリページをマッピングすべきアドレスが

点在する場合は, 必要な PTE の数が多くなる. ページサイズが 4 KB のとき, 2 MB のメモリをマッピングするには合計で 512 のエントリが必要となる. メモリページをマッピングするアドレス領域が連続している場合，1 つの PTE が 512 のエントリを持つため，必要な PTE の数は最低 1 つでよい．しかし，メモリページをマッピングするアドレスが点在している場合，最悪のケースでは 1 つの PTE が 1 つのメモリページをマッピングすることになり，最大で 512 の PTE が必要となる．PTE のサイズは 4 KB なので，この場合は 2 MB のメモリをマッピングするために 2 MB のメモリがページテーブルのために消費されてしまう.

同一ノード内で動作する 100 個の並列プロセスが全対全の通信を行なうとき，1 つの通信先についてマッピングしなければならない共有メモリのサイズが 2 MB だとする．この場合，システム全体で必要な PTE の個数は最低で（メモリページをマッピングするアドレスが連続する場合で），9,900（1×99×100）個，最高で（メモリページをマッピングするアドレスが点在する場合で），5,068,800（512×99×100）個となる．これをメモリ消費量に換算すると，最低で約 39MB，最高で約 19.3 GB ものメモリが消費されてしまう.

2.1.3 KNEM/LiMIC

OS カーネルの支援によってノード内通信を高速化する方式として，KNEM[14]と LiMIC[16]が提案されている．KNEM/LiMIC によるノード内通信をサポートするための Linux カーネルモジュールが, それぞれリリースされている. また，Linux のバージョン 3.2 以降では，KNEM/LiMIC と類似する方式のノード内通信として Cross-Memory-Attach（CMA）[17]が実装されている.

この方式では，受信プロセスがシステムコールを実行し，通信メッセージのコピーを OS カーネルに依頼する．この際，受信データを格納する送信プロセス

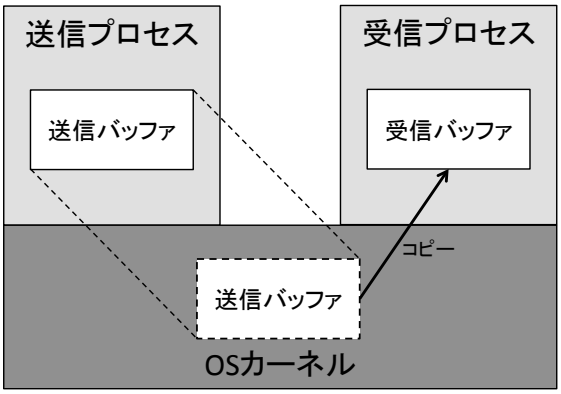


図 **2-4 KNEM/LiMIC** によるノード内通信

の送信バッファを OS カーネルに通知する必要がある．KNEM や LiMIC では，送信プロセスが送信バッファのアドレスとサイズを，システムコールを実行して OS カーネルに登録する．OS カーネルは登録されたバッファ領域に対応するディスクリプタをシステムコールの引数として送信プロセスに返す．送信プロセスは，送信バッファに対応するディスクリプタの番号を受信プロセスに何らかの手段（例：共有メモリによるノード内通信）で通知する．受信プロセスはシステムコールの引数としてディスクリプタの番号を渡すことで，データをコピーする送信バッファを OS カーネルに指定することができる．CMA では，受信プロセスが送信プロセスのプロセス ID と送信バッファのアドレスをシステムコールの引数として渡すことで, データをコピーする送信バッファを OS カーネルに指定することができる．

依頼を受けた OS カーネルは，送信バッファが格納されているメモリを OS カーネルが使用しているアドレス空間にマッピングする．そして，マッピングした領域からメッセージを受信プロセスの受信バッファにコピーする．メッセージのコピー終了後，マッピングしたメモリはアンマッピングされる．この方式では，図 2-4 に示すように，1 度のメモリコピーでメッセージの送受信を終えることができる．しかし，通信のたびにシステムコールを実行する必要があり，パイプやソケットを用いる通信と同様に，OS カーネルへのコンテキスト切り替えによるオーバヘッドが発生してしまい，通信遅延が増加してしまう．

2.1.4 XPMEM

共有メモリとは異なるメモリ共有の仕組みとして, XPMEM[25]が挙げられる．XPMEM は，Linux 向け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ことを可能にする．共有メモリの場合は，あるプロセスが明示的に共有メモリを作成し，その共有

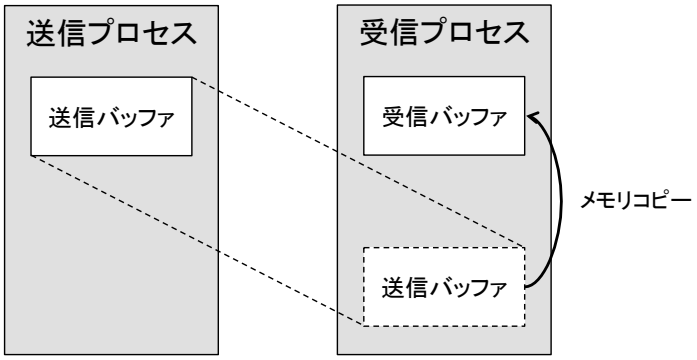


図 **2-5 XPMEM** によるノード内通信

メモリを他のプロセスが自身のアドレス空間にマッピングする. 対して XPMEM では，既に他のプロセスが使用しているメモリを，通信を行ないたいプロセスが自身のアドレス空間にマッピングする点が，共有メモリとは異なっている.

通信先のプロセスが使用しているメモリを，通信元のプロセスが自身のアドレス空間にマップすることで，ノード内通信に必要なデータの送受信を高速に行なうことができる．送信プロセスの送信バッファを，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, 1 回のメモリコピーで通信を完了できるようになるため（図 2-5 参照），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よりも通信遅延が小さくなる.

同じ通信バッファを用いて繰り返し通信を行う場合，メモリの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. よってこのケースでは, 最初の通信以外では, システムコールを実行するためのオーバヘッドが発生しない．しかし，通信のたびに新たな通信バッファを作成して使用するアプリケーションでは，通信のたびにシステムコールを実行する必要があり，そのオーバヘッドにより通信遅延が増加してしまう.

XPMEM を用いても, プロセス間で共有したいメモリを相互にマッピングしなければならない点は共有メモリを用いたときと同じである. よって, XPMEM を用いてノード内通信を実行しても，共有メモリを用いたときと同様に，メモリマッピングによりシステム全体のページテーブルサイズが増加し，メモリ消費量が大きくなってしまう.

2.2 関連研究

本節では，関連研究として，シングルアドレス空間やプロセス間通信を対象とする研究について述べる．シングルアドレス空間に関連する研究としては，主に以下が挙げられる.

- Single Address Space Operating Systems
- SMARTMAP

本研究の検証に用いる MPI のノード内通信に焦点を絞った研究として，以下が挙げられる.

- Hybrid MPI
- MPI のスレッド実装

- Shared Memory Window

また，本研究と関連するノード間通信の研究として，以下について述べる．

- User-mode Memory Registration
- 分散共有メモリ

2.2.1 Single Address Space Operating Systems

64 bit CPU の登場により，CPU が広大なアドレス空間を扱うことが可能になった．この 64 bit CPU の広大なアドレス空間を有効利用するため，Single Address Space Operating System (SASOS)[33-35][62-66]が提案されている．SASOS では，ノード上で動作するプロセスを一つのアドレス空間で動作させる．一つのアドレス空間を分割し，各プロセスに割り当てる．各プロセスは，自身に割り当てられたアドレス空間の範囲のみを使用することができる．

一般的に，SASOS では，メモリのアクセス権限をドメインという概念で管理する．SASOS では，各プロセスが個別のページテーブルでメモリのマッピング情報を管理している．各プロセスは，ページテーブルのうち，自身に割り当てられたアドレス範囲に対応する部分のみを使用する．プロセスは，同一ドメインに存在するプロセスが使用しているページテーブルの一部分を自身のページテーブルにコピーする．あるプロセスがページテーブルを更新したら，更新は，コピーを持つ全てのプロセスのページテーブルに反映される．こうすることで，同一ドメインに存在するプロセス同士が互いのメモリにアクセスすることが可能になり，一度のメモリコピーでデータ転送することができる．よって，PVAS タスクモデルを用いる場合と同様に高速なノード内通信が可能になる．ただし，各プロセスが個別のページテーブルを用い，通信対象のプロセスのメモリをマッピングするため，メモリマッピングによるページテーブルの肥大化を招いてしまう．

2.2.2 SMARTMAP

SMARTMAP[26]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．これにより，プロセスが他のプロセスのメモリに直接アクセスできるようになり，高速なノード内通信を実現できる．

SMARTMAP は，x86_64 アーキテクチャのページテーブルの構造を利用して実装されている．x86_64 のページテーブルは，図 2-3 に示すように，4 段階の階層構造になっている. SMARTMAP を利用するプロセスは, 第 1 階層のページテーブルである PGD の 512 エントリの内，最初のエントリのみを使用することができる．残りのエントリは，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．図 2-6 に示すように，同一ノード内の各プロセスの PGD の最初のエントリを，自身の PGD の残り 511 エントリのいずれかにコピーすることで，同一ノード内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. SMARTMAP によって相互にメモリマッピングするプロセス同士は，第 2 階層以降のページテーブルを共有するため，共有メモリを用いる場合のように，システム全体のページテーブルサイズが肥大化することはない．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，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.

SMARTMAP のメモリマッピングの仕組みでは, 他のプロセスのメモリにアク

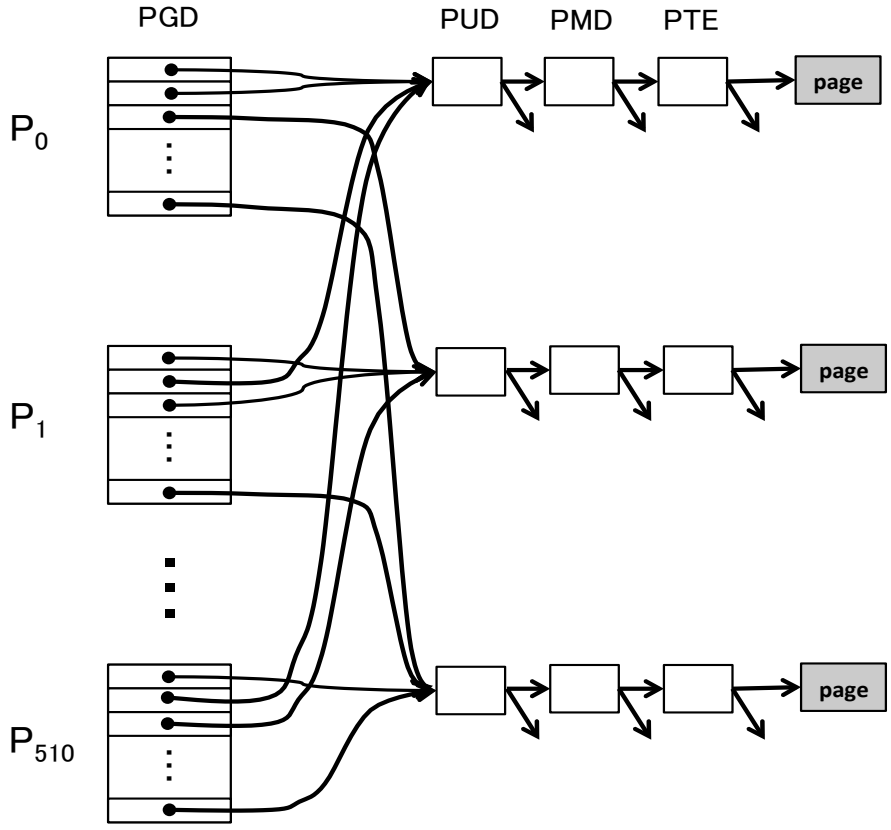


図 **2-6 SMARTMAP** のメモリマッピング

セスする際は，当該メモリのアドレスに適切なオフセット値を加算する必要がある．メモリアクセスの際に，アクセスするメモリのアドレスが，自プロセスのものか他プロセスのものかを意識する必要がある．

SMARTMAP の実装は x86_64 アーキテクチャのページテーブルの構造に深く依存しているため，ポータビリティが低いという問題がある．また，511 プロセス間でしか，相互にメモリのマッピングを行うことができないという制限がある．

現在 SMARTMAP は米国 Sandia 国立研究所の開発した軽量 OS カーネルである kitten[44]と Catamount[45]に実装されている．これらの OS カーネルはプロセスの起動時に当該プロセスが利用する全メモリの割当を行なう方式をとっており，オンデマンドページングをサポートしていない．よって，SMARTMAP を Linux のような一般的な OS に移植する際は，SMARTMAP を利用しているプロセス同士でページテーブル更新の同期を取る仕組みを導入し，オンデマンドページングに対応させる必要がある．

2.2.3 Hybrid MPI

高速な MPI ノード内通信を可能にする MPI の実装として Hybrid MPI[27]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，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や，スタック上に確保するケースでは，MPI ノード内通信を高速化することはできない．また，共有メモリを用いるため，共有メモリのマッピングによるページテーブルの肥大化が発生する．

2.2.4 MPI のスレッド実装

通常 MPI は，1 プロセスを 1MPI 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 1MPI 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[23-24][36-37]．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，共有

メモリを用いなくても，MPI ノード内通信に必要なデータの送受信を行なうことができる．共有メモリのマッピングによってページテーブルが肥大化することもないので，高速かつメモリ消費量の少ない MPI ノード内通信を実現することができる．しかし，1 スレッドを 1MPI プロセスとしてプログラミングを行なう場合，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．それに対し，PVAS タスクモデルを用いると，複数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．プロセス同士は個別のグローバル変数を保持することが可能なため，MPI のプログラミングモデルを維持しながら，スレッド実装の MPI と同等の性能とメモリ消費量を，理論上実現することができる．

2.2.5 Shared Memory Window

MPI 3.0 から，Shared Memory Window[69]と呼ぶ機能がサポートされている．Shared Memory Window を用いると，同一ノード内の MPI プロセスが load/store 命令でアクセス可能なバッファを共有メモリ上に作成することができる．

この機能を用いることで，同一ノード内の MPI プロセス間では，通信を実行することなく，計算処理に必要なデータを共有することができる．この機能を利用し，データを特定の領域に分割して領域ごとに計算を実行するステンシル計算を実装する研究が実施されている[70].

Shared Memory Window を用いることで，ノード内通信を実行する必要がなくなり，高速に計算処理を実行することが可能になるが，既存の MPI プログラムの構造を大きく変更する必要がある．また，共有メモリのマッピングのために，ページテーブルが肥大化してしまう問題がある．

2.2.6 User-mode Memory Registration

Mellanox の Infiniband がサポートする User-mode Memory Registration(UMR)[28]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一回の 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 MPI ノード間通信における不連続データの送受信を高速化する実装が提案されている[29]．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パッキングしてから 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，パッキングされた連続データを不連続な

データへアンパックする処理を行う．UMR を用いると，このパック/アンパック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 によって高速化した MPI ノード間通信と本研究で実装した MPI ノード内通信を組み合わせることで，不連続なデータの送受信を行う MPI アプリケーションの実行性能を大きく向上することができると考えられる．

2.2.7 分散共有メモリ

Memory Channel[30]のような通信アーキテクチャや Cenju[31]のような分散共有メモリアーキテクチャを用いると，リモートノード上のプロセスが使用しているメモリをローカルノード上のプロセスのアドレス空間にマッピングすることができる．リモートノード上のプロセスが使用しているメモリをローカルノード上のプロセスのアドレス空間にマッピングすることで，ノードをまたいだグローバルなアドレス空間を構築することが可能になり，ハードウェアのサポートによる高速なノード間通信を実現できる．

PVAS タスクモデルを用いてローカルノード内のプロセスを同一アドレス空間で動作させ，さらにリモートノードのプロセスが使用するメモリをそのアドレス空間にマッピングすることで，システム上で動作する全ての並列プロセスを同一アドレス空間で動作させることが，理論上可能になる．

2.2.8 関連研究のまとめ

関連研究をまとめると，表 2-2 のようになる．表には，各関連研究の研究対象を示した．比較のため，PVAS タスクモデルについても表に記載した．比較すると, SMARTMAP が本研究の内容と最も類似しているが, SMARTMAP によるノード内通信は，不連続データを送受信する MPI 通信には適用されていない．

また，ノード内通信を研究の対象とする SASOS，SMARTMAP，Hybrid MPI，MPI のスレッド実装，Shared Memory Window について，性能とメモリ消費量の観点から，PVAS タスクモデルを用いるノード内通信と比較した．比較結果を表 2-3 に示す．PVAS タスクモデルと同様，SMARTMAP と MPI のスレッド実装を用いると，高速かつ効率的なノード内通信を実行することができる．しかし，SMARTMAP には，実装が深くアーキテクチャに依存している，並列プロセスの数に制限がある等の問題がある．また，MPI のスレッド実装には，従来の MPI のプログラミングモデルから逸脱してしまう問題がある．

表 2-2 関連研究のまとめ

項目	研究対象				
	シングルアドレス空間	ノード内通信	ノード間通信	MPI 通信	
				連続データの送受信	不連続データの送受信
SASOS	○	○			
SMARTMAP	○	○		○	
Hybrid MPI		○		○	
MPI のスレッド実装		○		○	
Shared Memory Window		○		○	
User-mode Memory Registration			○		○
分散共有メモリ			○		
PVAS	○	○		○	○

表 2-3 関連研究との比較

方式	メモリコピー回数	システムコールオーバヘッド	中間バッファ	ページテーブル肥大化
SAOS	1 度	発生しない	不要	発生する
SMARTMAP	1 度	発生しない	不要	発生しない
Hybrid MPI	1 度	発生しない	不要	発生する
MPI のスレッド実装	1 度	発生しない	不要	発生しない
Shared Memory Window	通信不要			発生する
PVAS	1 度	発生しない	不要	発生しない

第3章　PVAS タスクモデルの設計

既存のタスクモデルでは，ノード内の並列プロセスが，それぞれ個別のアドレス空間で動作するため，ノード内の並列プロセス同士が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，既存のタスクモデルを前提とするマルチコア環境向けノード内通信では，アドレス空間をまたいでデータ転送するために，通信遅延の増加とメモリ消費量の増加をまねいてしまう．具体的には，アドレス空間をまたいでデータ転送するため，OS カーネルの支援や中間バッファ，メモリマッピングが必要となり，以下の問題が起こってしまう．

- 通信遅延の増加
  - 中間バッファを経由するデータ転送によるメモリコピー回数の増加
  - システムコールの実行によるオーバヘッド
- メモリ消費量の増加
  - 中間バッファの確保
  - メモリマッピングによるページテーブルの肥大化

そこで，本研究は，並列処理を実行するノード内の並列プロセスを同一アドレス空間で動作させる新たなタスクモデルを提案する．ノード内の並列プロセスを同一アドレス空間で動作させることで，並列プロセス同士が互いのメモリに直接アクセスすることを可能にする．アドレス空間をまたいでデータ転送する必要性をノード内通信から排除し，高速かつ効率的な，メニーコア環境向けのノード内通信を実現することを目標とする．

本研究で提案する新たなタスクモデルを，Partitioned Virtual Address Space (PVAS)と名付ける．PVAS タスクモデルでは，同一アドレス空間で複数の並列プロセスを動作させることを可能とするために，アドレス空間の分割と割当てを行う．一つのアドレス空間を複数の領域に分割し，分割した領域をノード内の各並列プロセスに割り当てることで，複数の並列プロセスがアドレス空間を共有可能とする. 本章では, PVAS タスクモデルの概要および設計について述べる．

3.1　アドレス空間レイアウト

図 3-1 は, 既存のタスクモデルと PVAS タスクモデルのアドレス空間レイアウ

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，各並列プロセスが個別のアドレス空間で動作する．TEXT/BSS/DATA/HEAP/STACK といったプロセス固有の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 タスクモデルでは，複数の並列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ため，アドレス空間の分割と割当を行う．PVAS タスクモデルでは，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並列プロセスに割り当てる．PVAS タスクモデル上で実行される並列プロセスを，通常のプロセスと区別するため，PVAS プロセスと呼ぶ．各 PVAS プロセスには，TEXT/BSS/HEAP/STACK といったプロセス固有の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させる．このようにして，プロセス間のアドレス空間の共有を実現する．通常のプロセスと同様に，TEXT セグメントのような同一データを格納しているメモリは，複数の PVAS プロセスから同一メモリページを参照するようにし，PVAS プロセス間で共有される仕様とする（図 3-2 参照）．

アドレス空間を論理的に分割するため，各 PVAS プロセスには，自身に割り当てられた PVAS パーティションのみを自身が利用可能なアドレス領域として認識させる．各 PVAS プロセスは，mmap や munmap といった仮想メモリ操作を，自身の PVAS パーティション内のアドレス領域を対象に実行することはできる

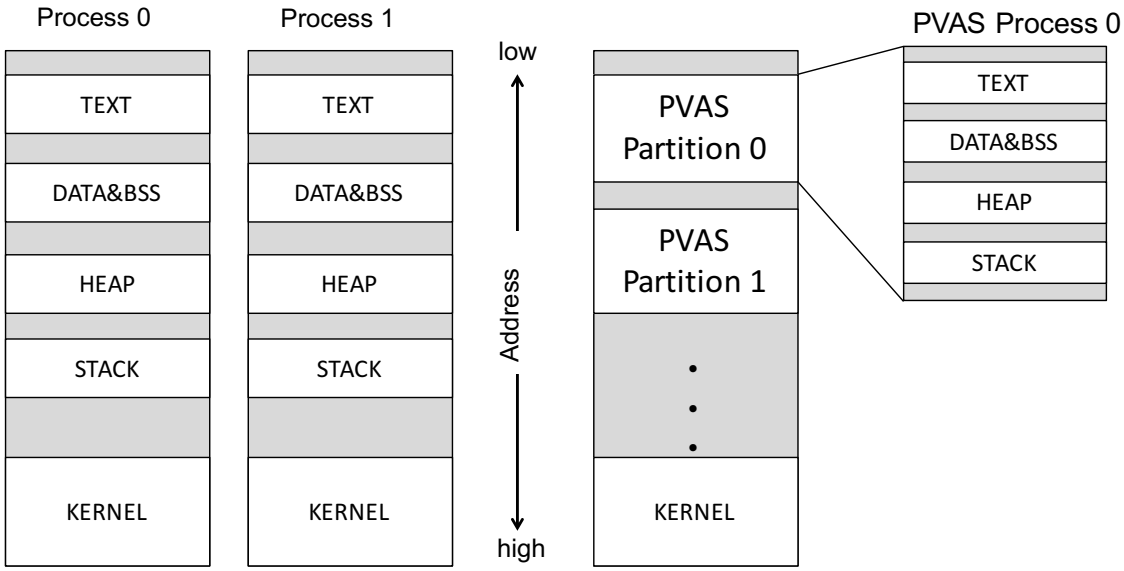


図 **3-1** アドレス空間のレイアウト

が, 他の PVAS プロセスに割り当てられた PVAS パーティション内のアドレス領域を対象にしては実行することはできないようにする．PVAS プロセスが mmap を実行すると, 自身の PVAS パーティション内からメモリをマッピングするためのアドレス領域を確保する．mmap では，メモリをマッピングするアドレス領域のアドレスを引数で指定することができるが, 他の PVAS プロセスの PVAS パーティション内のアドレスを指定する場合はエラーとする．また，PVAS プロセスは, 自身の PVAS パーティション内のアドレス領域に対して munmap を実行することができる．しかし，他の PVAS プロセスの PVAS パーティション内のアドレス領域を指定して munamap を実行した場合は, エラーとする. このようにして，アドレス空間を論理的に分割する．

同一アドレス空間内で動作可能な PVAS プロセスの数は，PVAS パーティションのサイズに依存するものとする．x86_64 アーキテクチャでは，256 TB のアドレス空間を扱うことができる. この場合, PVAS パーティションのサイズが 64 GB のときは，最大 4096 の PVAS プロセスが，同一アドレス空間で動作することが可能となる．

3.2 ページテーブル

既存のタスクモデルでは，各並列プロセスが異なるアドレス空間で動作するため，各並列プロセスが個別のページテーブルでメモリのマッピング情報を管理する．対して，PVAS タスクモデルでは，PVAS プロセスとして動作する並列プロセス群が同一アドレス空間で動作することを可能にするため，並列プロセス間で一つのページテーブルを共有し，メモリのマッピング情報を管理する．

SASOS のように，通信先のプロセスのページテーブルの内容を，通信元のプロセスのページテーブルにコピーすることでも，アドレス空間の共有を実現することはできる．しかし，この方式は，一種の共有メモリと考えることができる．図 3-3 で示すように，通信先のプロセスのメモリを通信元のプロセスがマッピングするため，共有メモリによるノード内通信を用いる場合と同様に，プロセス数の２乗のオーダでページテーブルサイズが増加してしまう．

これを回避するため，図 3-2 で示すように，PVAS タスクモデルでは，一つのページテーブルで, 同一アドレス空間で動作する PVAS プロセスのメモリのマッピング情報を管理する. 同一ページテーブルを複数の PVAS プロセスが共有するので，SASOS のような余計なメモリマッピングを行うことなく，アドレス空間を共有できる．

3.3 プログラムの実行

プロセスレベルの並列化を利用するアプリケーションを，ソースコードを改変することなく実行可能となるよう，PVAS プロセスは，アドレス空間を他のプロセスと共有する点以外については, 通常のプロセスと同等の仕様とする. PVAS プロセスには，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たせる.

これにより，既存のアプリケーションをそのまま実行することが可能となる.

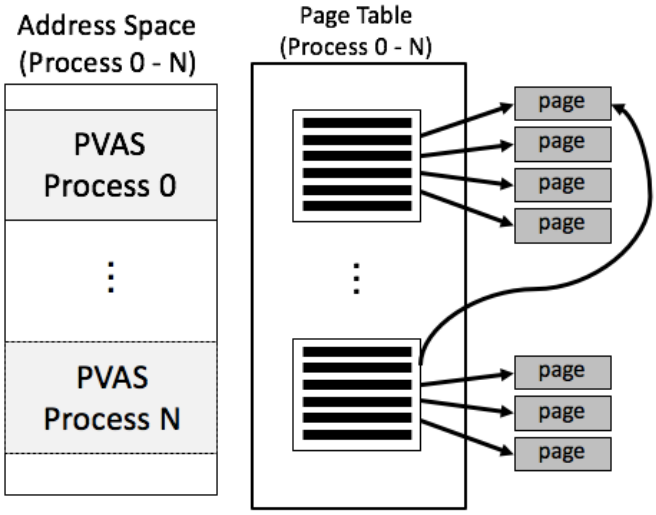


図 **3-2 PVAS** のアドレス空間共有

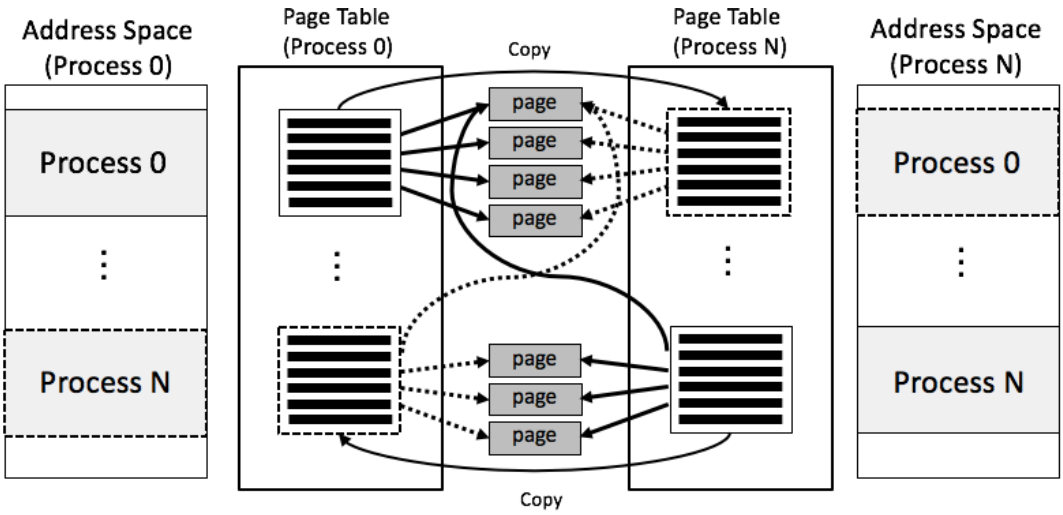


図 **3-3 SASOS** のアドレス空間共有

ただし，PVAS タスクモデルでは，プログラムがロードされるアドレスが PVAS プロセスごとに異なるため，プログラムを位置独立実行形式(Position Independent Executable(PIE))としてビルドする必要がある．GNU コンパイラ[18]や Intel コンパイラ[19]では，-fPIE オプションを用いてプログラムをコンパイルし，-pie オプションを用いてプログラムのリンクを行えば，PIE 形式のバイナリを生成することができる．

PIE としてビルドされたプログラムでは，シンボルへのアクセスがシンボルテーブルを経由した間接参照になるため，頻繁にグローバル変数にアクセスするようなプログラムでは，非 PIE としてビルドした場合と比べて，性能が低下する可能性がある[40]．しかし，HPC アプリケーションのような大規模な計算データを扱うプログラムでは，メモリアクセスの大半はこの大規模データへのアクセスとなる．メモリアクセスのうち，シンボル解決が必要となるメモリアクセスの比率は相対的に少なくなるため，PIE と非 PIE の性能差は一般的なアプリケーションよりも小さくなる．よって，プログラムを PIE としてビルドしても大きな問題にはならないと考え，3.1 節で示すようなアドレス空間レイアウトとした．

これを確認するため，NAS Parallel Benchmarks (NPB)の各アプリケーションを PIE と非 PIE でビルドした場合の性能比較を行なった．NPB には，HPC 分野でよく利用される科学計算を MPI による並列処理で実行するアプリケーションが含まれている．また，アプリケーションごとに複数の問題サイズ（科学計算に

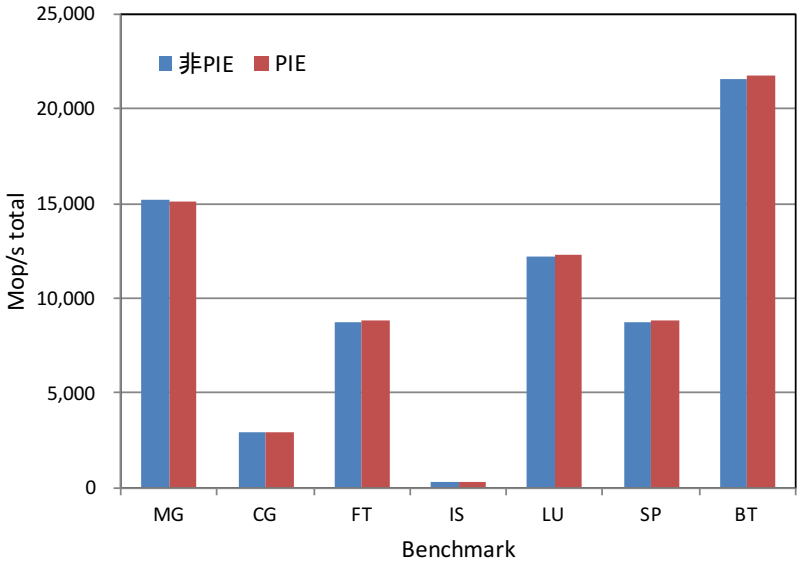


図 **3-4 PIE** と非 **PIE** の比較

よって処理するデータのサイズ）をサポートしている．NPB については，第 6 章で詳しく述べる．NPB の MG，CG，FT，IS，LU，SP，BT アプリケーションを，非 PIE としてビルドして実行した場合と，PIE としてビルドして実行した場合の実行性能を図 3-4 に示す．評価環境は第 6 章の表 6-1 で示すものと同様のものを用いた．問題サイズは，標準的な問題サイズとされるクラス B 用いた．また，MPI のノード内通信の方式は，共有メモリを選択した．MG，CG，FT，IS，LU においては，並列プロセス数を 128 プロセスとして実行した．SP，BT ベンチマークにおいては，プロセス数を 225 プロセスとして実行した．PIE としてビルドした場合と非 PIE としてビルドした場合の性能差は-0.5%から+1.4%の範囲にとどまっている．また，PIE と非 PIE の性能差に明確な相関関係はなく，両者の間で有意な差は見られない．以上の結果から，HPC アプリケーションについては，プログラムを PIE としてビルドしても大きな問題にはならないと考えられる．

3.4 PVAS タスクモデルにおけるメモリ保護

PVAS タスクモデルでは，プロセス同士がアドレス空間を共有する．よって，並列プロセス間のメモリ破壊が発生する可能性がある．通常，ある並列アプリケーションのプロセスが異常な動作を起こした場合，HPC システムのジョブスケジューラは，当該並列アプリケーションの全プロセスを終了させ，チェックポイントから再始動するか，別の並列アプリケーションの実行を開始する．あるプロセスが，なんらかの異常な動作を起こした場合，それが同一アプリケーションの他のプロセスのメモリを破壊するものであろうとなかろうと，ジョブスケジューラによって全プロセスが終了させられるのに変わりはない．よって，同一アプリケーションのプロセス間のメモリ保護を行わなくても，大きな問題にはならないと考えられる．ただし，別の並列アプリケーションのプロセスのメモリを破壊するのは，防止する必要がある．

このため，PVAS タスクモデルでは，並列アプリケーションごとにアドレス空間を作成できる仕様とする．並列アプリケーション J の PVAS プロセス群をアドレス空間 A で実行し，並列アプリケーション K の PVAS プロセス群を別のアドレス空間 B で実行可能とすることで，並列アプリケーション間で，メモリ破壊が発生するのを，回避可能とする．

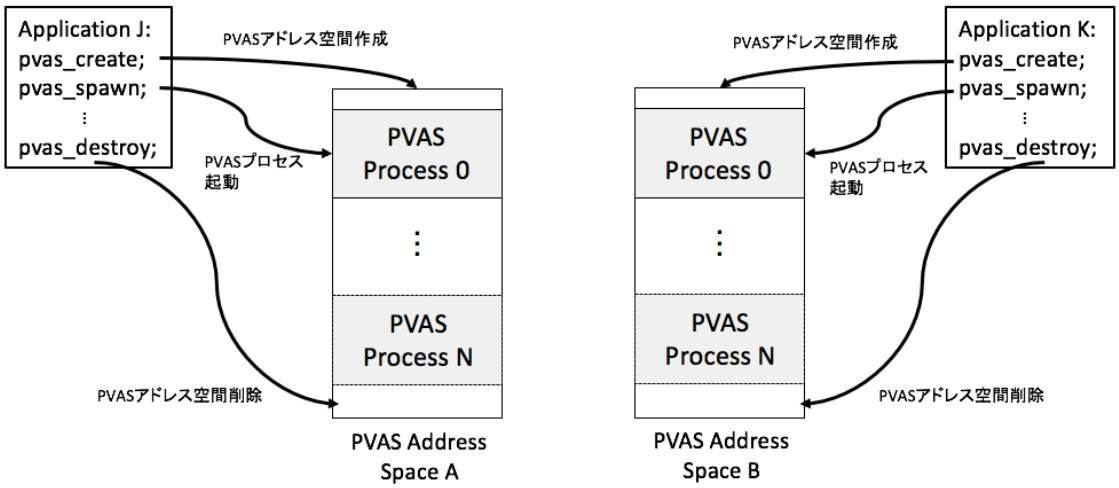


図 **3-5 API** の使用方法

表 3-1 PVAS API

関数名	内容
pvas_create	アドレス空間を作成する
pvas_destroy	アドレス空間を削除する
pvas_spawn	PVAS プロセスを起動する
pvas_get_pvid	PVAS プロセスの PVAS ID を取得する
PVAS_PROCESS_SIZE	PVAS パーティションのサイズを示すマクロ

このように，PVAS タスクモデルでは，HPC システムで最低限求められるアプリケーション間のメモリ保護を実現する．しかし，メモリ破壊によってデバッグの難易度が向上する問題は避けることはできない．メモリ破壊を起こすようなバグは，バグを起こしているソースコード上の箇所を特定するのが難しく，アプリケーションのデバッグが従来よりも困難になってしまうというデメリットがある．

3.5 API

これまで述べてきた PVAS タスクモデルを，ユーザプログラムが利用するための API を設計した．設計した API の一覧を表 3-1 に示す．

API の使用方法を，図 3-5 に示す．まず，pvas_create によって，PVAS プロセスを動作させるアドレス空間を作成する．このアドレス空間を PVAS アドレス空間と呼ぶ．3.4 節で述べた通り，PVAS アドレス空間は，ノード内に複数作成す

ることができる．そして，pvas_spawn によって PVAS プロセスを作成した PVAS アドレス空間に起動する．PVAS プロセスが実行を終了したら，pvas_destroy によって，PVAS アドレス空間を削除する．pvas_get_pvid と PVAS_PROCESS_SIZE は，PVAS プロセス間で通信を実行するときに用いる．これについては，3.6 節で詳しく述べる．以下，各 API の詳細を説明する．

3.5.1 pvas_create

```
int pvas_create (int *pvd);
```

pvas_create は，PVAS プロセスを動作させるための PVAS アドレス空間を作成する．PVAS アドレス空間のサイズは，CPU がサポートするアドレス空間のサイズに依存するものとする．例えば，x86_64 アーキテクチャでは，アドレス空間のサイズは 256 TB となる．PVAS アドレス空間は，PVAS パーティションに分割され，各 PVAS プロセスに割り当てられる．

pvas_create は作成した PVAS アドレス空間のディスクリプタを返す．引数 *pvd に作成した PVAS アドレス空間のディスクリプタが格納される．このディスクリプタによって，PVAS プロセスを起動する PVAS アドレス空間を指定することができる．pvas_create は，成功した場合，返り値として 0 を返す．失敗した場合は，発生したエラーに対応するエラーコードを返り値として返す．

3.5.2 pvas_destroy

```
int pvas_destroy (int pvd);
```

pvas_destroy は，pvas_create によって作成された PVAS アドレス空間を削除するために用いられる．引数 pvd によって，削除する PVAS アドレス空間を指定する．pvas_destroy は，成功した場合返り値として 0 を返す．失敗した場合は，発生したエラーに対応するエラーコードを返り値として返す．

3.5.3 pvas_spawn

```
int pvas_spawn (int pvd, int pvid, const char *filename, char *const argv[], ch
```

ar *const envp[], pid_t *pid);

pvas_spawn は PVAS プロセスを作成するために用いられる．引数 pvd には PVAS プロセスを作成する PVAS アドレス空間のディスクリプタを指定する．引数 pvid には, PVAS プロセスに割り当てる任意の ID を指定する. この ID を PVAS ID（≧0）と呼ぶ．作成した PVAS プロセスは，引数 *filename で指定したプログラムを実行する．引数 argv[]には実行するプログラムの引数を，引数 envp[]には，PVAS プロセスの環境変数を指定する．

PVAS プロセスは OS カーネルからは通常のプロセスとして認識される．よって，PVAS プロセスを作成した親プロセスは，wait 関数によって，PVAS プロセスの終了を検知することができる．また，PVAS プロセスは，通常のプロセスと同様に固有のプロセス ID (PID)を持つ．引数 *pid に作成した PVAS プロセスの PID が格納される．pvas_spawn は，成功した場合返り値として 0 を返す．失敗した場合は，発生したエラーに対応するエラーコードを返り値として返す．

3.5.4 pvas_get_pvid

int pvas_get_pvid (int *pvid);

pvas_get_pvid は, 呼び出した PVAS プロセスの PVAS ID を返す. PVAS ID は，引数 *pvid に格納される．通常のプロセスが pvas_get_pvid を呼び出すとエラーとなる．pvas_get_pvid は，成功した場合返り値として 0 を返す．失敗した場合は，発生したエラーに対応するエラーコードを返り値として返す．

3.5.5 PVAS_PROCESS_SIZE

PVAS_PROCESS_SIZE は PVAS パーティションのサイズを示すマクロである．PVAS アドレス空間に存在する PVAS パーティションのサイズは全て同一のサイズとする．PVAS パーティションのサイズはユーザが指定することはできず，一意に定められたものとする．

PVAS_PROCESS_SIZE は，PVAS プロセスが，通信先の PVAS プロセスの通信バッファのアドレスを計算するためなどに用いる．詳しくは，3.6 節で述べる

3.5.6 API の使用例

図 3-6 は，PVAS タスクモデルの API を使って PVAS プロセスを作成するサンプルコードである．まず，pvas_create を用いて，PVAS プロセスを動作させる PVAS アドレス空間を作成する．

そして，pvas_spawn を用いて PVAS プロセスの作成を行う．引数に pvas_create

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pvas.h>

#define PROCESS_NUM 32

int main (int argc, char **argv)
{

   pid_t pid[PROCESS_NUM];
   int pvd;
   int i;
   int status;
   int error;

  error = pvas_create(&pvd);
  if(error)
     return error;

   for(int i=0; i<PROCESS_NUM; i++) {
     int pvid = i;
     error = pvas_spawn (pvd, &pvid, "./test", NULL, NULL, &pid[i]);
     if(error)
        return error;
   }

   for(int i=0; i<PROCESS_NUM; i++) {
     wait(&status);
   }

   error = pvas_destroy(pvd);
   if(error)
     return error;

   return 0;
}
```

図 **3-6 API** の使用例

で作成した PVAS アドレス空間のディスクリプを指定する. また, 作成した PVAS プロセスに割り当てる PVAS ID を指定する．作成した PVAS プロセスは引数で指定したプログラム（./test）を実行する．

次に，wait 関数を用いて，作成した PVAS プロセスが終了するまで待機する．全ての PVAS プロセスが終了したら，pvas_destroy を用いて，pvas_create で作成した PVAS アドレス空間を削除する．

3.6 PVAS プロセス間の通信

既存のタスクモデルでは，各プロセスが異なるアドレス空間で動作するため，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ない．よって，アドレス空間をまたいでデータ転送する必要があり，通信遅延の増加やメモリ消費量の増加といった問題が，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 プロセスは，他の PVAS プロセス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，アドレス空間をまたいでデータ転送することなく，ノード内通信を実行することができる．

図 3-7 は PVAS プロセス同士で通信を実行する際のサンプルコードである. サンプルコードでは，PVAS ID が 1 と 2 の PVAS プロセスが通信を実行している．互いの通信バッファ（src_buf）のデータを受信バッファ（dist_buf）にコピーすることで，通信を実行することができる．PVAS タスクモデルでは，PVAS ID によって，PVAS プロセスに割り当てる PVAS パーティションを管理している．例えば, PVAS パーティションのサイズ（PVAS_PROCESS_SIZE）が 4 GB で, PVAS パーティションを割り当てる PVAS プロセスの PVAS ID が 2 の場合は, 仮想アドレス 0x200000000 から 0x2fffffff の領域が，当該 PVAS プロセスの PVAS パーティションとして割り当てられる. よって，通信を実行する PVAS プロセスが同一プログラム（同一バイナリ）を実行し，通信バッファがグローバル変数として定義されている場合は，サンプルコードで示すように，通信先の PVAS ID と PVAS パーティションのサイズから，通信先の通信バッファのアドレス（remote_src_buf）を計算で求めることができる．

通信バッファが malloc 等によって動的に確保されている場合，通信バッファのアドレスをグローバル変数に格納することで，通信バッファのアドレスを，通信を実行する PVAS プロセス同士が確認することができる．また，通信バッファのアドレスは，既存のノード内通信を用いて交換しても良い．

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <pvas.h>

#define BUF_SIZE 1024;

int main (int argc, char **argv)
{
   int my_pvid, remote_pvid;
   char src_buf[BUF_SIZE];
   char *remote_src_buf;
   volatile int ready = 0;
   volatile int complete = 0;
   volatile int *remote_ready
   volatile int *remote_complete;
   char dist_buf[BUF_SIZE];

   int pvas_get_pvid(&my_pvid);
   if(error)
      return error;

   if(my_pvid == 1)
      remote_pvid = 2;
   else
      remote_pvid = 1;

   memset(buf, pvid, BUF_SIZE);
   remote_ready = &ready + PVAS_PROCESS_SIZE*(remote_pvid – my_pvid);
   remote_complete = &complete + PVAS_PROCESS_SIZE*(remote_pvid  – my_pvid);
   remote_src_buf = buf + PVAS_PROCESS_SIZE*(remote_pvid – my_pvid);
   ready = 1;

   while(*remote_ready == 0){}; //通信先のバッファの準備が完了するまで待つ

   memcpy(dist_buf, remote_src_buf, BUF_SIZE);
   complete = 1;

   while(*remote_complete == 0){}; //通信先がコピーを完了するまで待つ

   sleep(1); //通信相手がコピーの完了を検知する前に終了するのを回避

   return 0;
}
```

図 **3-7 PVAS** プロセス間の通信

共有メモリやソケット/パイプを用いるノード内通信では，中間バッファを経

由してデータ転送をするためにメモリコピーの回数が増加していたが，PVAS タスクモデルによるノード内通信では，送信バッファから受信バッファに直接メモリコピーを行なって，1 回のメモリコピーでデータを送受信することができる．加えて，システムコールを実行する必要も無い．また，中間バッファの確保によるメモリ消費やメモリマッピングによるページテーブルサイズの肥大化が発生しないため，高速かつメモリ消費量が小さいノード内通信を実現することができる．

第4章　PVAS タスクモデルの実装

本章では，PVAS タスクモデルの実装について述べる．本研究では，Xeon Phi コプロセッサ用の Linux カーネル[20]に PVAS タスクモデルを実装した．

4.1　Linux のタスク管理とメモリ管理

まず，PVAS タスクモデルを実装する Linux カーネルのタスク管理とメモリ管理について述べる．

4.1.1　Linux カーネルのタスク管理

Linux カーネルは，プログラムを実行するスレッド（カーネルレベルスレッド）を task_struct 構造体（タスク構造体）によって管理する．タスク構造体には，スレッドが属するプロセスのプロセス ID や，プロセスが使用するメモリを管理するための mm_struct 構造体（MM 構造体）へのポインタ等が含まれている．Linux カーネルは，現在実行しているスレッドのタスク構造体に，*current* マクロによってアクセスすることができる．これにより，実行中のスレッドのタスク構造体のメンバを取得することができる．例えば，*current->mm* とすることで，実行中のスレッドが属するプロセスの MM 構造体を取得することができる．

4.1.2　Linux カーネルのメモリ管理

通常，ユーザプログラムは，mmap システムコールを実行することによって，データを格納するためのメモリ領域を確保する．ユーザプログラムが mmap システムコールを実行すると，mmap システムコールの引数で指定したサイズのメモリ領域が，Linux カーネルによって確保される．

Linux カーネルは，確保したメモリ領域を vm_area_struct 構造体（VMA 構造体）で管理する．VMA 構造体には，確保したメモリ領域のサイズ，先頭アドレス，終端アドレス等が格納されている．VMA 構造体は，図 4-1 で示すようなリスト

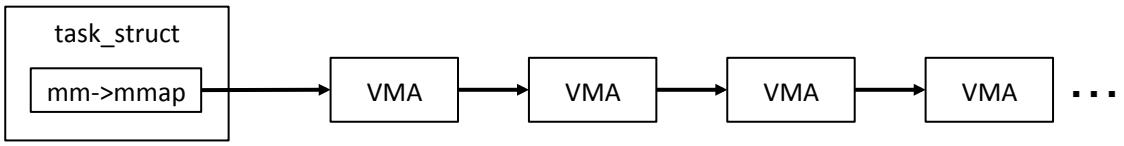


図 **4-1 VMA** 構造体

構造で管理される．図で示すように，プロセスが確保したメモリ領域に対応する VMA 構造体が，アドレス空間上でのアドレスを元にソートされ，リストとして連結される．この VMA リストを辿ることで，プロセスの確保しているメモリ領域やアドレス空間上の空き領域を確認することができる．VMA リストには，タスク構造体の mm->mmap メンバからアクセスすることができる．

mmap が新たに実行された場合，Linux カーネルは，VMA リストをたどり，要求されたサイズの空き領域を検索する．空き領域を見つけたら，確保するメモリ領域に対応する VMA 構造体を作成し，VMA リストに追加する．munmap によってメモリ領域の解放を行う場合は，VMA リストを辿って，解放するメモリ領域に対応する VMA 構造体を検索する．対応する VMA 構造体を見つけたら，その VMA 構造体を VMA リストから削除する．

同一プロセスのスレッド同士は，アドレス空間を共有するため，図 4-2 で示すように，共通の VMA リストを用いるようになっている．同一プロセスに属するスレッドは同一 MM 構造体を用いるため，タスク構造体の mm->mmap メンバが同一の VMA リストを指すことになり，VMA リストが共有される．VMA リストを共有するため，VMA リストを参照，更新する場合はスレッド間で排他制御を実行する必要がある．

Linux では，プロセスに割り当てる物理メモリをページという単位に区切って管理している．プロセスが，mmap によって確保したメモリ領域に対してメモリページの割り当てを行うことで，プロセス上で動作するプログラムがメモリにアクセスすることが可能になる．

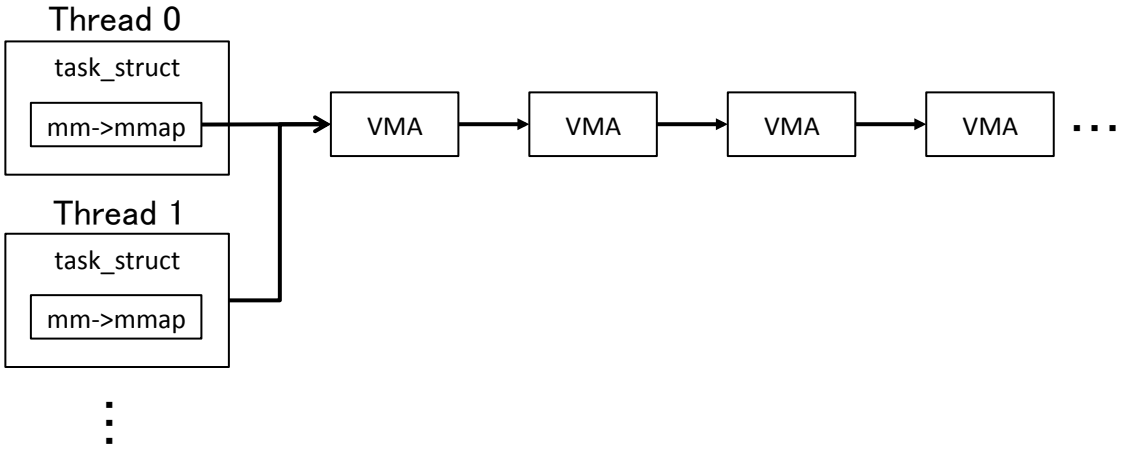


図 **4-2** スレッドによる **VMA** リストの共有

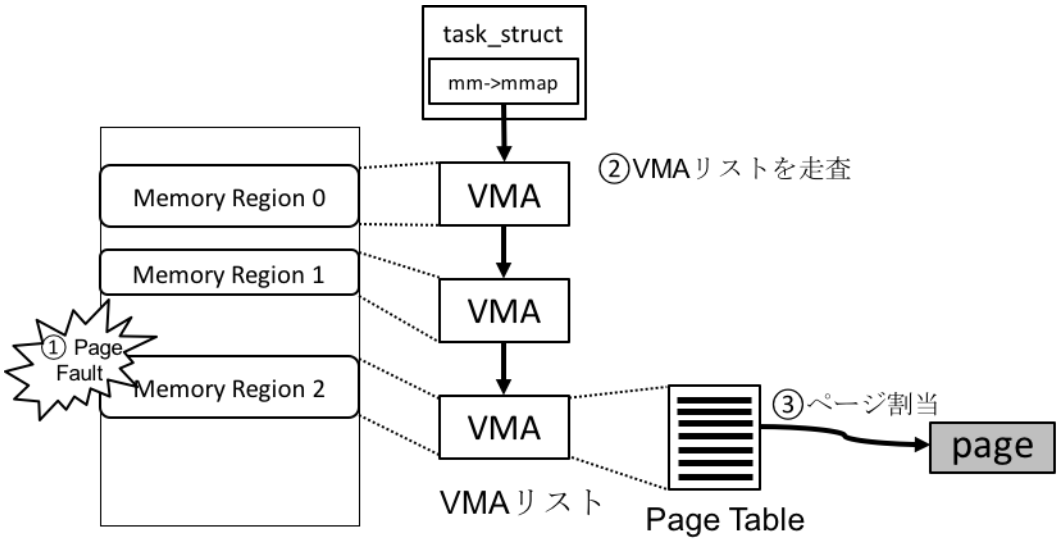


図 **4-3** オンデマンドページング

Linux は，メモリページとメモリページを割り当てたアドレスの対応表を，ページテーブルと呼ぶテーブルで管理している．ページテーブルを参照することで，メモリを割り当てたアドレス空間上のアドレス（仮想アドレス）と物理メモリ上でのアドレス（物理アドレス）の変換処理を，Memory Management Unit (MMU)が実行し，メモリへのアクセスが可能になる．プロセスが使用しているページテーブルへのポインタは，MM 構造体の pgd メンバに格納されている．

Linux はオンデマンドページングによって，プロセスにメモリページを割り当てる．オンデマンドページングとは，プロセスが mmap で確保したメモリ領域に対してプログラムがアクセスしたタイミングで，メモリページの割り当てを行う方式である．図 4-3 は，オンデマンドページングの処理を示している．メモリページをまだ割り当てていない領域にプログラムがアクセスしようとすると，ページフォルトという例外を MMU が発生させる．ページフォルトが発生すると，Linux カーネルのページフォルトハンドラに処理が移る．この際，ページフォルトが発生した仮想アドレスが CPU を経由して MMU から Linux カーネルに通知される．ページフォルトハンドラは，ページフォルトを発生させたプログラムを実行するスレッドのタスク構造体から VMA リストにアクセスして走査し，ページフォルトが発生した仮想アドレスが，mmap によって確保されたメモリ領域に含まれているかどうかを確認する．もし，仮想アドレスが mmap によって確保された領域に含まれていたら，使用されていないメモリページを Least Recently Used（LRU）リストから検索し，当該仮想アドレスに割り当てる．この際，ページテーブルの更新が発生する．もし，ページフォルトが発生したアド

レスが，mmap によって確保された領域に含まれていなければ，セグメンテーションフォルトとなる．

同一プロセスに属するスレッド群は，同一ページテーブルを共有する．よって，ページテーブルを参照，更新する際には，スレッド間で排他制御を行う必要がある．

4.2 システムコール

PVAS タスクモデルの機能をサポートし，それをユーザプログラムが使用可能とするために，以下のシステムコールを Linux カーネルに実装した．

(1) *sys_pvas_create*
(2) *sys_pvas_destroy*
(3) *sys_pvas_spawn*

本節では，上記システムコールの実装について述べる．

4.2.1 sys_pvas_create

long sys_pvas_create (unsigned long info_addr, int n);

sys_pvas_create は，PVAS アドレス空間を作成するためのシステムコールである．sys_pvas_create が実行されると，Linux カーネルは，PVAS プロセスを動作させるためのアドレス空間を作成する．作成した PVAS アドレス空間は，図 4-4

```
struct pvas_address_space {
    pid_t pid;     /* PID of the owner process */
    pgd_t *pgd;    /* PGD of the PVAS address space */

    ...

    atomic_t num_of_proc;   /* Number of PVAS tasks */
    pid_t child_pids[MAX_PVAS_PROCESS_NUM]; /* List of the PVAS processes */

    ...
};
```

図 **4-4 pvas_address_space** 構造体

で示す pvas_address_space 構造体で管理される．

pvas_address_space 構造体には，PVAS アドレス空間を作成したプロセスのプロセス ID（pid）が含まれる．また，pvas_address_space 構造体には，PVAS アドレス空間に割り当てたページテーブルへのポインタ（pgd_t *pgd）が格納される．既に述べたように，仮想アドレスと物理メモリへのマッピングは，ページテーブルによって管理される．よって，Linux カーネルは，sys_pvas_create が実行されると，PVAS アドレス空間を管理するためのページテーブルを作成し，それを作成した PVAS アドレス空間に割り当てる．その他，pvas_address_space 構造体には， PVAS アドレス空間内に作成した PVAS プロセスの数，作成した PVAS プロセスのプロセス ID のリスト等が格納される．

pvas_address_space 構造体は，配列 pvas_address_space_array によって管理される．pvas_address_space 構造体が作成されたら，作成された構造体へのポインタが配列に格納される．そして，ポインタを格納した配列上のインデックスを，作成した PVAS アドレス空間のディスクリプタとする．当該ディスクリプタは，sys_pvas_create の返り値として，sys_pvas_create を実行したユーザプログラムに渡される．

配列 pvas_address_space_array はプロセスごとに確保される．配列 pvas_address_space_array は専用のリストで管理され，プロセス ID をキーとして検索可能となっている．図 4-5 に示すように，pvas_address_space_array 管理リストを先頭からたどり，当該プロセスの pid を持つエレメントを検索する．該当するエレメントをリストから見つけたら，エレメント内に格納されている pvas_address_space_array へのポインタ（list_elem: void *ptr）を参照して，

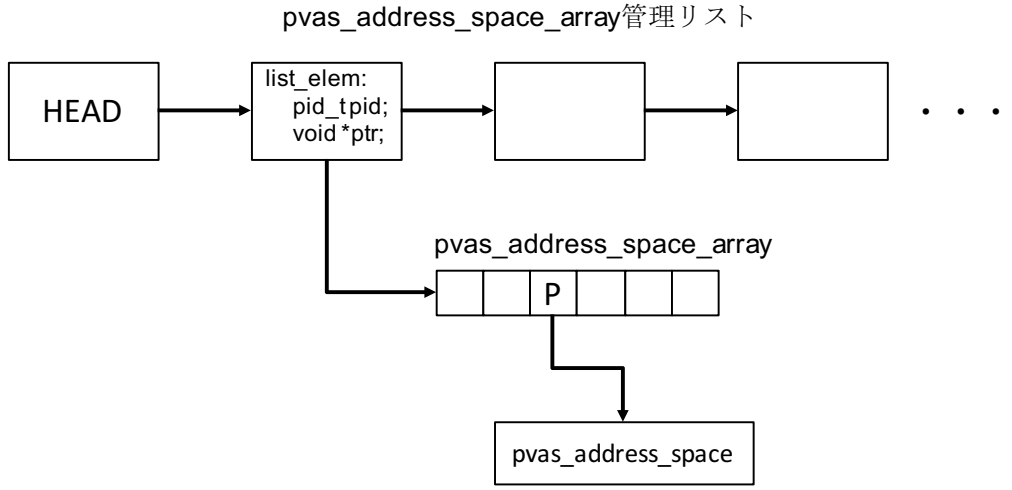


図 **4-5 pvas_address_space** 構造体の検索

pvas_address_space_array にアクセスすることができる．pvas_address_space 構造体へのポインタ（図中「P」）は，当該 pvas_address_space 構造体のディスクリプタをインデックスとして, pvas_address_space_array から検索することができる.

4.2.2 sys_pvas_destroy

```
long sys_pvas_destroy (int pvd);
```

sys_pvas_destroy は PVAS アドレス空間を削除するためのシステムコールである．sys_pvas_destroy は，削除する PVAS アドレス空間のディスクリプタを，システムコール引数としてユーザプログラムから受け取る.

sys_pvas_destroy がユーザプログラムによって実行されたら，Linux カーネルは，実行中のスレッドのプロセス ID と，引数として受け取った PVAS アドレス空間のディスクリプタから，削除対象の PVAS アドレス空間を管理する pvas_address_space 構造体を検索する.

Linux カーネルは，削除対象の PVAS アドレス空間に対応する pvas_address_space 構造体のポインタを配列 pvas_address_space_array から取得したら，配列 pvas_address_space_array に格納されているポインタの値を Null に変更する．そして，pvas_address_space 構造体のためのメモリを解放し，sys_pvas_destroy の処理を終了する.

4.2.3 sys_pvas_spawn

```
long sys_pvas_spawn (int pvd, int pvid, char __user *filename,
                     char __user * __user *argv, char __user * __user *envp);
```

sys_pvas_spawn は PVAS プロセスを作成し, ユーザプログラムから指定されたプログラムを実行させるためのシステムコールである. sys_pvas_spawn は, PVAS プロセスを作成する PVAS アドレス空間のディスクリプタ（int pvd），作成した PVAS プロセスに割り当てる PVAS ID（int pvid），PVAS プロセスが実行するプログラムへのパス（char __user *filename），実行するプログラムの引数（char __user * __user *argv），作成する PVAS プロセスの環境変数（char __user * __user *envp）を，システムコールの引数としてユーザプログラムから受け取る.

Linux カーネルは，sys_pvas_spawn が実行されたら，PVAS プロセスのためのタスク構造体を作成する．タスク構造体の mm メンバからポイントされる MM 構造体には，プロセスが使用するページテーブルへのポインタが格納される．これに，sys_pvas_spawn の引数 pvd（PVAS アドレス空間のディスクリプタ）で指定された PVAS アドレス空間のためのページテーブル（pvas_address_space 構造体の pgd メンバ）を設定する．PVAS プロセスを作成する PVAS アドレス空間の pvas_address_space 構造体は，sys_pvas_spawn の引数 pvd と sys_pvas_spawn を実行したスレッドのプロセス ID を用いて検索することができる. 作成した PVAS プロセスのページテーブルに PVAS アドレス空間のページテーブルを設定することで，作成した PVAS プロセスが，PVAS アドレス空間を，自身のアドレス空間として認識するようになる．

ページテーブルを設定したら，Linux カーネルは，PVAS プロセスの PVAS ID（sys_pvas_spawn の引数 pvid），PVAS プロセスが実行するプログラムのパス（sys_pvas_spawn の引数 filename），プログラムの引数（sys_pvas_spawn の引数 argv），PVAS プロセスが属する PVAS アドレス空間の pvas_address_space 構造体へのポインタを，タスク構造体のメンバとして格納する．また，ユーザプログラムから渡された環境変数（sys_pvas_spawn の引数 envp）を，作成した PVAS プロセスに設定する．そして，PVAS プロセスのタスク構造体を，カーネルのタスクスケジューラに登録する．以上で sys_pvas_spawn の処理が終了し，作成した PVAS プロセスのプロセス ID が，sys_pvas_spawn の返り値として，sys_pvas_spawn を呼び出したユーザプログラムに渡される．

Linux カーネルは，タスクスケジューラに登録された PVAS プロセスが，スケジューラによって CPU にスケジューリングされたタイミングで，PVAS プロセスが実行するプログラムを PVAS アドレス空間にロードする処理を実行する．Linux カーネルは，PVAS プロセスのタスク構造体を参照してロードするプログラムのパスを確認し, 当該プログラムのバイナリを PVAS アドレス空間上にロードする. そして, PVAS プロセスのためのスタック領域を PVAS プロセスの PVAS アドレス空間上に確保し，ユーザプログラムから渡された引数をスタックに格納する．格納する引数は，タスク構造体を参照することで取得することができる．バイナリをロードする位置とスタックを確保する位置は，PVAS プロセスに割り当てられた PVAS パーティションの範囲内から選択する．PVAS プロセスに割り当てられた PVAS パーティションの範囲は，PVAS プロセスのタスク構造体に格納されている PVAS ID から計算することができる．

プログラムのロードが終了したら，Linux カーネルは，作成した PVAS プロセ

スのインストラクションポインタをロードしたバイナリのメイン関数のアドレスに設定し，カーネル空間からユーザ空間へのコンテキストスイッチを実行する．こうすることで，作成した PVAS プロセスが，ユーザに指定されたプログラムを実行するようになる．

4.3 メモリ管理

本節では，PVAS タスクモデルでのメモリ管理について述べる．

4.3.1 メモリ領域の確保

PVAS プロセスは，自身の PVAS パーティションのアドレス領域を対象にしてしか，仮想メモリ操作を実行できない仕様になっている．よって，PVAS プロセスが mmap によってメモリ領域を新たに確保する場合，空き領域を自身の PVAS パーティション内からしか検索できないように，mmap の処理を改造した．PVAS プロセスの VMA リストを辿って，PVAS プロセスの PVAS パーティションの範囲に空き領域がなければ，エラーを返す．PVAS プロセスの PVAS パーティションの範囲は，タスク構造体の pvid メンバに格納されている PVAS ID から計算することができる．

同一 PVAS アドレス空間で動作する PVAS プロセス同士は，アドレス空間を共有する．しかし，スレッドの場合とは違い，図 4-6 で示すように，個別の MM 構造体でメモリ管理を行ない，各 PVAS プロセスは個別の VMA リストを持つ．各 PVAS プロセスの VMA リストは，PVAS プロセスに割り当てられた PVAS パー

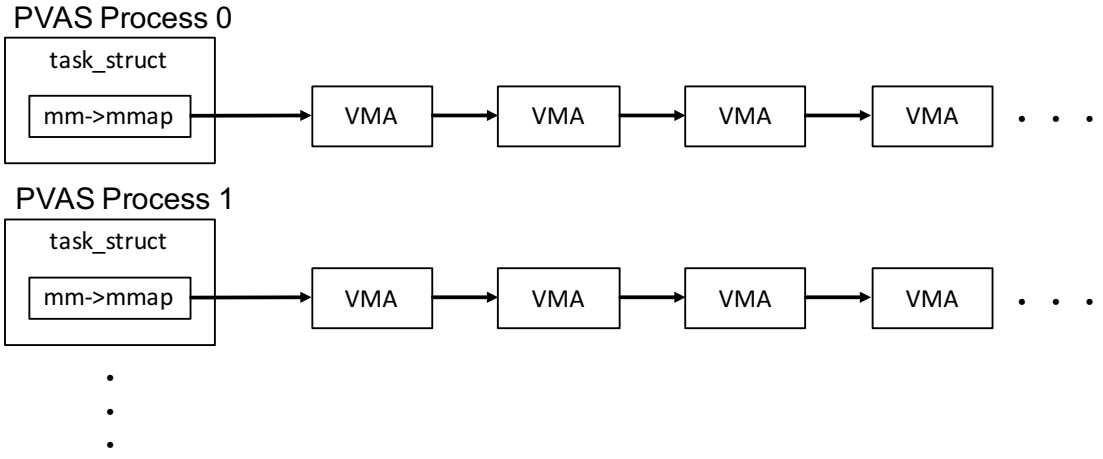


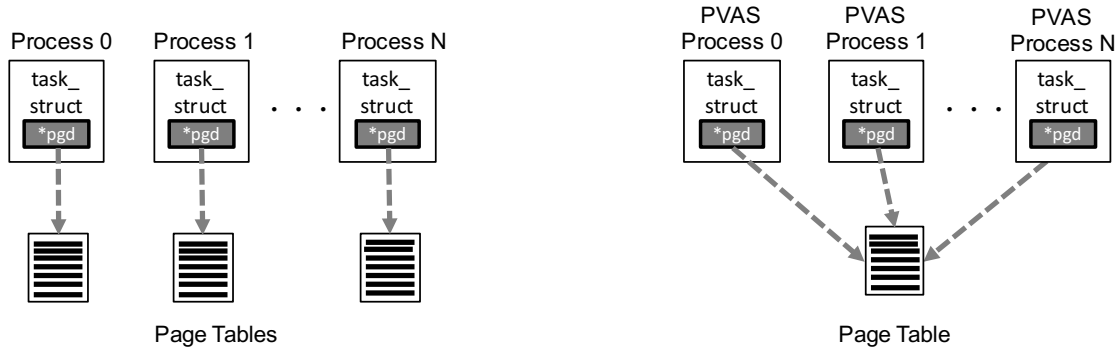
図 **4-6 PVAS** プロセスの **VMA** リスト

ティションの範囲のアドレス空間のみを管理する．PVAS プロセスは自身に割り当てられた PVAS パーティションの範囲にのみしか仮想メモリ操作を実行することができない．よって，全アドレス空間を共有するスレッドとは違い，各 PVAS プロセスは，自身に割り当てられた PVAS パーティションの範囲のみを，自身の VMA リストで管理すればよい．各 PVAS プロセスが個別の VMA リストを持つので，mmap と munmap の実行時に，PVAS プロセス同士で排他制御を行う必要はない．よって，mmap や munmap といった仮想メモリ操作を頻繁に呼び出すプログラムを PVAS プロセスが実行しても，排他制御による性能の低下が発生することはない．

4.3.2 メモリページの割り当て

4.2.3 節で述べたように，PVAS プロセスには，PVAS プロセスが動作する PVAS アドレス空間のためのページテーブルが割り当てられる．よって，同一 PVAS アドレス空間で動作する PVAS プロセス同士は，図 4-7 に示すように，同一ページテーブルを共有することになる．各 PVAS プロセスのタスク構造体の pgd メンバが同一ページテーブルを参照する．同一ページテーブルを共有するため，メモリページの割り当て時には，PVAS プロセス間で排他制御を実行する必要がある．

通常，Linux カーネルは，1 ページテーブルに対して 1 つのロックオブジェクトを割り当てる．よって，ページテーブルを更新する際には，ページテーブル全体のロックを取得し，排他制御を行うことになる．この排他制御による性能低下を低減するため，PVAS プロセスモデルでは，PVAS パーティションの単位でロックオブジェクトを作成する最適化をメモリページの割り当て処理に導入



既存のタスクモデル　　PVASタスクモデル

図 **4-7 PVAS** タスクモデルでのページテーブル

し，細粒度の排他制御を行う．

2.1.2 節で述べたように, 近年の CPU では, ページテーブルが階層構造をとる．テーブル内の各エントリが下層のテーブルをポイントするツリー構造になっており, 最終的に, 最下層のテーブルのエントリがメモリページをポイントする．このツリーを，図 4-8 で示すように，PVAS パーティションのサイズ単位でサブツリーに論理的に分解し，各サブツリーに対してロックオブジェクトを作成する．各サブツリー内のテーブルを更新するときは，当該サブツリーのロックオブジェクトによって排他制御を行う．よって，異なるサブツリーのテーブルを更新するときは，競合が発生しない．サブツリーより上位のテーブルを更新する際は，ページテーブル全体を対象にしたロックオブジェクトで排他制御を行うため，競合が発生する．

通常, PVAS プロセスは自身の PVAS パーティションのアドレス範囲にアクセスするため，ページテーブルの更新時に競合は発生しない．ページテーブル更新時の競合は，PVAS プロセスが他の PVAS プロセスに割り当てられた PVAS パーティションのアドレス範囲にアクセスしたときに発生する．しかし，PVAS

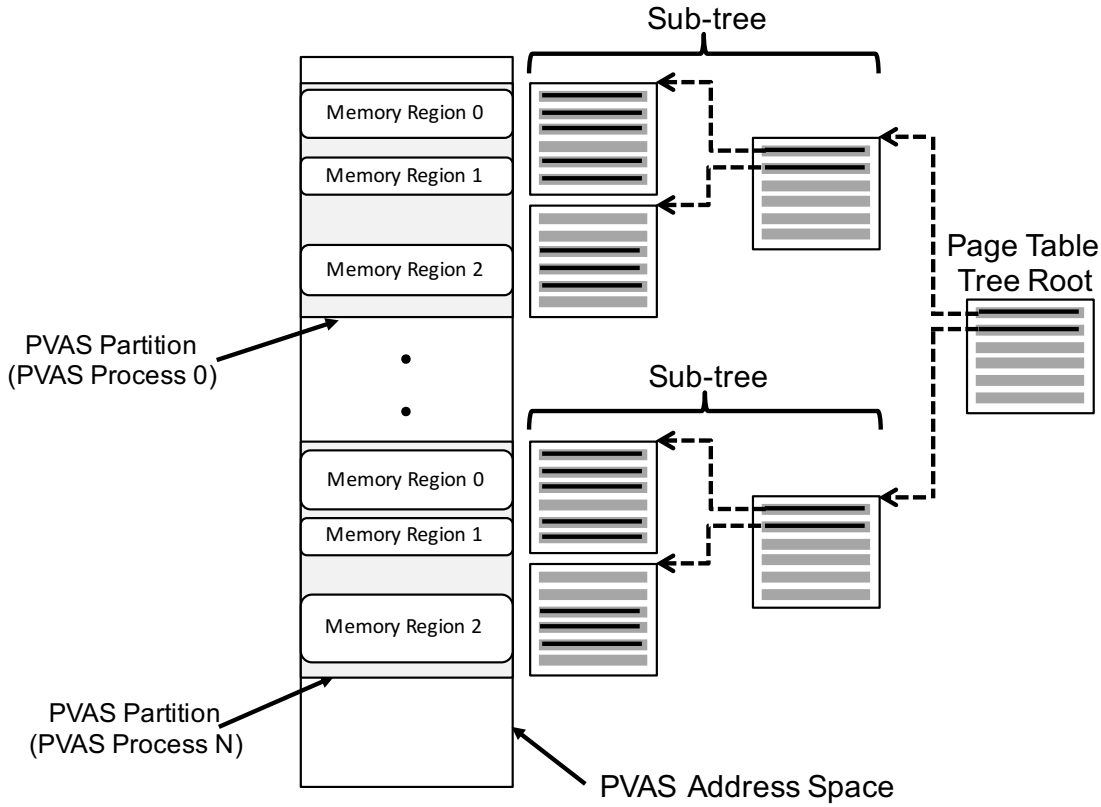


図 **4-8** ページテーブルツリーの構造

プロセスが他の PVAS プロセスに割り当てられた PVAS パーティションのアドレス範囲にアクセスするとき，そのアドレスには既にメモリページが割り当てられている可能性が高いため，ページテーブル更新時の競合が発生する頻度は低いと考えられる．

4.4 他の OS およびアーキテクチャへの適用

本研究では，PVAS タスクモデルを，Linux カーネルに実装した．しかし，他の OS カーネルにおいても，タスク管理およびメモリ管理方式は，Linux カーネルと大きな違いはないため，本章で述べた PVAS タスクモデルの実装方式は，他の OS カーネルにも適用することができる．例えば，FreeBSD[57]等の Unix 系 OS におけるカーネルのタスク管理およびメモリ管理方式の基本的な部分は Linux カーネルと大差はなく，本研究の実装を適用することができると考えられる．

PVAS タスクモデルの実装において, CPU アーキテクチャに深く依存している部分は，ページテーブルを操作する処理のみである．本研究では，x86_64 アーキテクチャをベースとする Xeon Phi コプロセッサを実装の対象としたが，他の CPU アーキテクチャ（例：Itanium[58]，sparc[59]，ARM[61]，PowerPC[60]等）においても，x86_64 アーキテクチャと同様に，多段構造のページテーブルを採用しており，本章で述べた実装方式を適用することができると考えられる．

第5章 MPI ノード内通信の実装

本研究では，PVAS タスクモデルの有用性を検証するために，MPI のノード内通信に PVAS タスクモデルを適用した．本章では，まず MPI の概要と，既存の MPI ノード内通信の実装について述べる．その後，PVAS タスクモデルを利用した MPI ノード内通信の実装について説明する．

5.1 MPI

MPI は並列計算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 のランタイムは，計算処理を実行する MPI プロセスを起動するための MPI プロセスマネージャ[38]と，MPI プロセスに通信 API を提供する MPI ライブラリから構成される．現在，C 言語や Fortran 向けの MPI ランタイムが Open Source Software (OSS)として多数公開されている．

MPI のランタイムが提供する MPI アプリケーション実行コマンド（mpiexec/mpirun 等）を実行すると，MPI プロセスマネージャが MPI プロセスを起動する．起動した MPI プロセスは，コマンドの引数で指定された MPI アプリケーションを実行する．起動された MPI プロセス同士は，計算処理に必要なデータを MPI ライブラリの提供する通信 API を用いて送受信することができる．

計算処理を実行する MPI プロセスの数は，MPI アプリケーション実行コマンドの引数によって指定する．MPI 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，利用可能な最大 CPU コア数と，起動する MPI プロセスの数を一致させるのが一般的になっている．利用可能な CPU コア数が大きくなると，より多くの MPI プロセスがシステム上で動作することになる．

MPI 通信は，異なるノード上で動作する MPI プロセス同士の通信である MPI ノード間通信と，同一ノード上で動作する MPI プロセス同士の通信である 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環境では，多数の MPI プロセスが同一ノード上で動作するため，MPI ノード内通信の発生回数が従来よりも増加する．

5.1.1 MPI 通信における基本概念

本節では，MPI 通信における基本概念について説明する．

初期化/終了処理

MPI アプリケーションは，MPI の提供する機能を用いる前に，必ず MPI_Init を実行する必要がある．また, MPI アプリケーションの終了時には, MPI_Finalize を実行する必要がある．

MPI ランク

MPI は，MPI プロセスに MPI ランクと呼ぶ識別子を付与する．通信を実行する MPI プロセスは，データを送受信するプロセスを MPI ランクによって指定することができる．MPI プロセスが，自身の MPI ランクを確認する API（MPI_Comm_rank）等がサポートされている．

コミュニケータ

MPI では，計算処理を実行する MPI プロセスをコミュニケータと呼ぶ集団にまとめることができる．MPI では，ブロードキャスト通信や全対全通信といった集団通信をサポートしているが，こういった集団通信はコミュニケータの単位で実行される．例えば，ブロードキャスト通信を実行すると，指定したコミュニケータに属する全 MPI プロセスにデータが送信される．全ての MPI プロセスを含むコミュニケータとして，MPI_COMM_WORLD が，あらかじめ定義されている．コミュニケータの作成（MPI_Comm_create），分割（MPI_Comm_split），解放（MPI_Comm_free）といった機能が MPI ではサポートされる．また，コミュニケータに属する MPI プロセスの数を返す API（MPI_Comm_size）等がある．

MPI ランクはコミュニケータごとに MPI プロセスに付与される．ある MPI プロセスが複数のコミュニケータに所属する場合，コミュニケータごとに異なる MPI ランクが, MPI プロセスに付与される．MPI プロセスが２つのコミュニケータに所属する場合，２つの MPI ランクを持つことになる．

表 5-1 基本データ型

MPI data type	C data type
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long in
MPI_FLOAT	float
MPI_DOUBLE	double

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．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 5-1 に示す．

5.1.2 1 対 1 通信

本節では，主要な 1 対 1 通信の API について説明する．説明には，C 言語向けの API を用いる．

MPI_Send

```
int MPI_Send (const void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm);
```

buf で示すアドレスから，datatype で示すデータ型のデータを count 個送信する．送信データには識別番号 tag が付与される．コミュニケータ comm における MPI ランク dest の MPI プロセスにデータが送信される．MPI_Send を実行すると，送受信処理が完了するまで，次の処理に移行することができない．

MPI_Recv

```
int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Status *status);
```

buf で示すアドレスに, datatype で示すデータ型のデータを count 個受信する．コミュニケータ comm における MPI ランク source の MPI プロセスから，識別子 tag のデータ受信する．status には送信プロセスの情報が格納される．MPI_Recv を実行すると，送受信処理が完了するまで，次の処理に移行することができない．

MPI_Isend

```
int MPI_Isend (const void *buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request);
```

基本的な動作は，MPI_Send と同じである．ただし，MPI_Isend はノンブロッキング通信であるため，送受信処理が完了しなくても，次の処理に移行することができる．request には，本通信の通信リクエストが格納される．送受信処理を完了するには，対応する通信リクエストに対して MPI_Wait を実行する必要がある．

MPI_Irecv

```
int MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request *request);
```

基本的な動作は，MPI_Recv と同じである．ただし，MPI_Irecv はノンブロッキング通信であるため，送受信処理が完了しなくても，次の処理に移行することができる．request には，本通信の通信リクエストが格納される．送受信処理を完了するには，対応する通信リクエストに対して MPI_Wait を実行する必要がある．

MPI_Wait

```
int MPI_Wait (MPI_Request *request, MPI_Status *status);
```

MPI_Wait を実行すると，request に対応する送受信処理を完了させることができる．MPI_Wait を実行すると，送受信処理が完了するまで，次の処理に移行することができない．

5.1.3 集団通信

本節では，主要な集団通信の API について説明する．説明には，C 言語向けの API を用いる．

MPI_Bcast

```
int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm );
```

MPI ランク root の MPI プロセスが指定した buffer のアドレスから，コミュニケータ comm 内の他の全ての MPI プロセスの buffer のアドレスに，datatype で示すデータ型のデータを count 個送信する．

MPI_Scatter

```
int MPI_Scatter (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                 MPI_Comm comm);
```

MPI ランク root の MPI プロセスが指定した sendbuf のアドレスから sendcount 個ずつ，sendtype で示すデータ型のデータを，comm 内の MPI プロセス 0 から順に全 MPI プロセスの recvbuf のアドレスに送信する．受信側では，recvtype で示すデータ型のデータを recvcount 個受信する．

MPI_Gather

```
int MPI_Gather (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```

comm 内の MPI プロセス 0 から順に送られてくる sendbuf 上のデータを MPI ランク root の MPI プロセスの recvbuf へ受信する．送信側は sendtype で示すデータ型のデータを sendcount 個ずつ送信する．受信側は recvtype で示すデータ型のデータを recvcout 個ずつ受信する．

MPI_Allgather

```
int MPI_Allgather (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, int recvcount, MPI_Datatype recvtype,
                   MPI_Comm comm);
```

基本的には MPI_Gather と同じであるが，各 MPI プロセスから全 MPI プロセスへギャザする点が異なる．

MPI_Alltoall

```
int MPI_Alltoall (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
```

全プロセスが全プロセスに対しスキャッタ通信を行なった場合と同じ結果を得られる．

MPI_Reduce

```
int MPI_Reduce (const void *sendbuf, void *recvbuf, int count, MPI_Datatype
```

datatype, MPI_Op op, int root, MPI_Comm comm);

comm 内の全プロセスの sendbuf 上のデータに op で指定した演算を実施して，演算結果を，MPI ランク root の MPI プロセスの recvbuf に送る．演算するデータの型と個数は datatype と count で指定する．

MPI_Allreduce

int MPI_Allreduce (const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

基本的には MPI_Reduce と同じであるが，演算した結果を全 MPI プロセスへ送信する点が異なる．

5.1.4 派生データ型による通信

MPI では基本データ型の他に，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ひと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．例えば，図 5-1(A)に示すような 2 次元配列のデータがあり，グレイのブロックのデータを送信する場合，送信するデータは図 5-1(B)で示すように，メモリ上に不連続に配置される．

表 5-2 派生データ型を定義するための API（C 言語）

API
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);

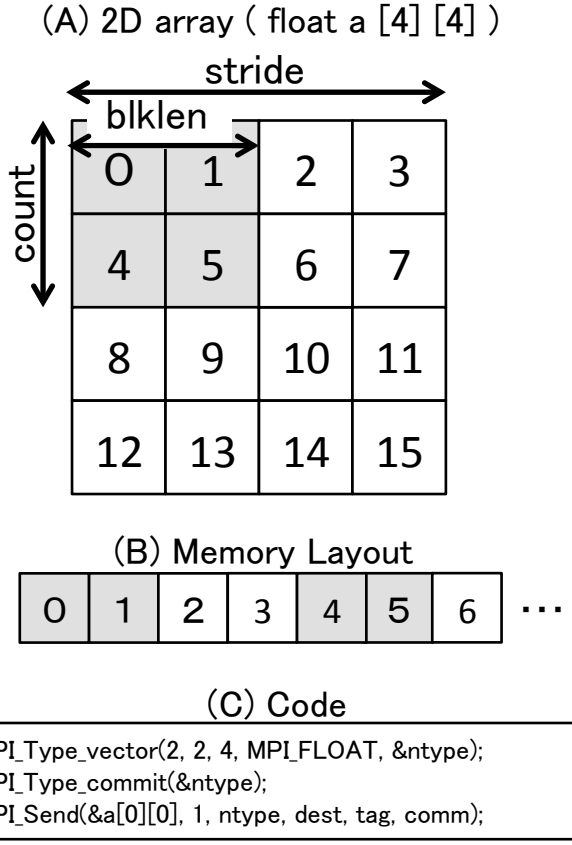


図 **5-1** 派生データ型を用いる通信

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，一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，既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 API を表 5-2 に示す．

図 5-1(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なり，図 5-1(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 といった通信 API 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 MPI 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し使用して通信を行うことができる．

5.2 MPI ノード内通信の実装

Open MPI や MPICH[21]，MVAPICH[39]といった主要な MPI 実装では，低遅延かつ高スループットな実装として，共有メモリを用いた MPI ノード内通信をサポートしている．本節では，共有メモリを用いた MPI ノード内通信の実装，およびその問題点について述べる．主要な MPI ライブラリの一つである Open MPI の実装を例にして説明するが，他の MPI ライブラリでも実装方式に大きな差はない．

5.2.1 共有メモリを用いた MPI ノード内通信の概要

Open MPI の構造はモジュール化されており，通信アルゴリズムやデータの送受信方式を柔軟に追加・変更することができる．Open MPI では，Byte Transfer Layer (BTL)において，マルチコア環境向けノード内通信モジュールである sm BTL をサポートしている. sm BTL は, 共有メモリによって, データ転送を行う．BTL レイヤは，Open MPI においてデータ転送を担当するレイヤである．

MPI 通信を実行するためには，通信データを MPI プロセス間で送受信する必要がある. sm BTL をはじめとする共有メモリを用いた MPI ノード内通信の実装では，通信を行なう MPI プロセスの双方が利用可能な共有メモリを作成し，両プロセスのアドレス空間にマッピングする．マッピングした共有メモリにデータの読み書きを行うことで，異なるアドレス空間で動作する MPI プロセス同士がデータの送受信を行なう．

5.2.2 メモリプール

sm BTL では，データの送受信を共有メモリ上に確保した通信キューとバッファを用いて行う. 送信側の MPI プロセスが, 送信データを共有メモリ上のバッファにコピーし，当該バッファを通信先のプロセスの通信キューに追加する．通信キューとバッファは共有メモリ上に存在するため，送受信側の MPI プロセ

スの双方からアクセスすることができる．

各 MPI プロセスの通信キューとバッファを格納する共有メモリは，MPI ライブラリの初期化時に作成されたメモリプールから取得する．MPI アプリケーションの実行時，最初に起動した MPI プロセスがある程度大きなサイズの共有メモリを作成する．他の MPI プロセスは，図 5-2 に示すように，その共有メモリを自身のアドレス空間にマッピングし，メモリプールとして利用する．共有メモリの作成およびマッピングは mmap を用いて行う．メモリプールから取得されたメモリは，共有メモリとして同一ノード内の全 MPI プロセスのアドレス空間にマッピングされるため，どの MPI プロセスからでもアクセスすることができる．

Linux のような仮想記憶をサポートする近年の OS カーネルはメモリをページ単位で管理している．オンデマンドページングによるページ割り当てを行う OS カーネルでは, mmap を実行した時点では, 実際にはメモリの割当は行なわない．プロセスが mmap 実行したアドレス領域に実際にアクセスした際に，当該アドレスに対してメモリページの割当を行なう．よって，ある程度大きなサイズのメモリプールを作成したとしても，作成した時点でメモリ消費量が大きく増加することはない．メモリプールの内，実際に MPI ノード内通信のために使用された領域にのみ，メモリが共有メモリとして割り当てられ，その分だけメモリ消費量が大きくなる．

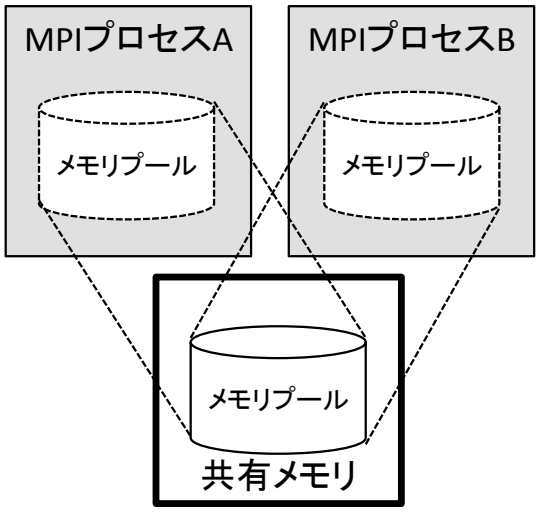


図 **5-2** 共有メモリプール

5.2.3 Eager 通信

主要な MPI ライブラリでは，データの送受信において，Eager 通信と Rendezvous 通信の 2 つの通信プロトコルをサポートしている．Eager 通信は，通信を行うプロセス間で同期なしに通信を行う方式である．一方，Rendezvous 通信は送信プロセスと受信プロセス双方の準備が完了した時点で通信を開始する方式である．まず，sm BTL の Eager 通信の実装について述べる．

Eager 通信では，送信データを Eager 通信用のバッファにバッファリングすることで，非同期にデータの送受信を行なう．sm BTL の Eager 通信では，まず送信プロセスが前節で述べたメモリプールから Eager 通信用のバッファを取得する．この Eager 通信用バッファは共有メモリ上に存在するため，送信プロセスと受信プロセスの双方がデータの読み書きを行なうことができる．次に送信プロセスは，送信するデータを自身の送信バッファから Eager 通信用バッファにコピーする．そして，データをコピーした Eager 通信用バッファを受信プロセスの通信キューに追加して送信処理を完了する．受信プロセスは受信バッファの準備が完了した時点で，当該 Eager 通信用バッファからデータを自身の受信バッファにコピーして受信処理を完了する．Eager 通信では，受信プロセスの受信準備が完了するのを待つことなく，送信側が通信を開始，完了することができるので，通信遅延が小さくなる．

一度使用された Eager 通信用バッファは，以降の Eager 通信で再利用される．sm BTL では, 再利用する Eager 通信用バッファを free list によって管理している．free list を参照する際の競合を避けるため，各 MPI プロセスは，個別の free list で，自身の利用する Eager 通信用バッファを管理する．Eager 通信の際，送信プロセスは自身の free list から使用していない Eager 通信用バッファを取得して Eager 通信を行なう．受信プロセスはデータのコピーを終えた後，当該 Eager 通信用バッファを送信プロセスの free list に返却する処理を行なう. free list に利用できるバッファがなかった場合，送信プロセスはメモリプールから新たに Eager 通信用バッファを取得して，free list に追加する．集団通信のように短期間に多数の通信を行なう処理が実行された場合, 一度に多数の Eager 通信用バッファがメモリプールから取得され，メモリ消費量が大きくなる．

データサイズが大きくなると次節で述べる理由により，Rendezvous 通信の方が Eager 通信よりも通信遅延が小さくなる. sm BTL では Eager 通信と Rendezvous 通信を切り替えるデータサイズを *eager limit* というパラメータで調整することができる. sm BTL では eager limit のデフォルト値は 4 KB である. 送信するデー

タと付随するメタデータの合計サイズが eager limit 以下の場合，そのデータは Eager 通信で送信される. Eager 通信用バッファの再利用を可能にするため, Eager 通信用バッファのサイズは eager limit に設定されたサイズとなる.

5.2.4 Rendezvous 通信

Rendezvous 通信は，送信プロセスと受信プロセス双方の準備が完了した時点でデータの送受信を行なう同期通信である．図 5-3 は，Rendezvous 通信の概要を示している．Rendezvous 通信では，まず送信プロセスが Ready-to-Send (RTS) メッセージを受信プロセスに送信する．送信処理に対応する受信処理が実行されたら，受信プロセスは Clear-to-Send(CTS)メッセージを送信プロセスに送信する．送信プロセスが CTS メッセージを受け取った時点で同期が完了し，データの送受信が実行される．なお，RTS メッセージと CTS メッセージは Eager 通信によって送受信される.

通信を行なう双方のプロセスは個別のアドレス空間で動作するため，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はできない．よって，データの送受信は，共有メモリ上に確保した中間バッファを経由して行なう．以下に sm BTL における Rendezvous 通信の実装について述べる.

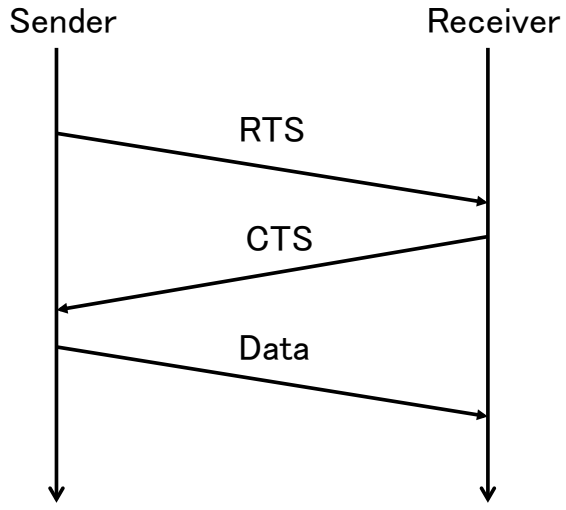


図 **5-3 Rendezvous** 通信の概要

sm BTL の Rendezvous 通信では，まず送信プロセスがメモリプールから Rendezvous 通信用の中間バッファを取得する．この中間バッファは共有メモリ上に存在するため，送信プロセスと受信プロセスの双方からデータを読み書きすることができる. 次に送信プロセスは, 送信するデータを図 5-4 に示すように, 先頭から順に中間バッファにコピーしていく．図では連続データを送受信するケースを示しているが，不連続データの場合も同様に，バッファの先頭から順にデータをコピーしていく．そして，データをコピーした中間バッファから，受信プロセスの通信キューに追加していく．受信プロセスは，送信プロセスからのコピーが完了した中間バッファから順々に，データを自身の受信バッファにコピーしていく．このように，中間バッファを経由したパイプライン転送を行なうことで, 送信プロセス側のメモリコピー (送信バッファから中間バッファ) と受信プロセス側のメモリコピー（中間バッファから受信バッファ）をオーバラップさせることが可能になる．このため，図 5-5 に示すように，単純にデータを，中間バッファを経由してコピーするよりも通信遅延が小さくなる.

Eager 通信では, 送信プロセスと受信プロセスが非同期に通信を行う. よって, Rendezvous 通信のように，パイプライン転送でデータを送受信することができない．データを送受信する際に，メモリコピーをオーバラップさせることができないので，ある程度，送受信するサイズが大きくなると，Eager 通信の方が

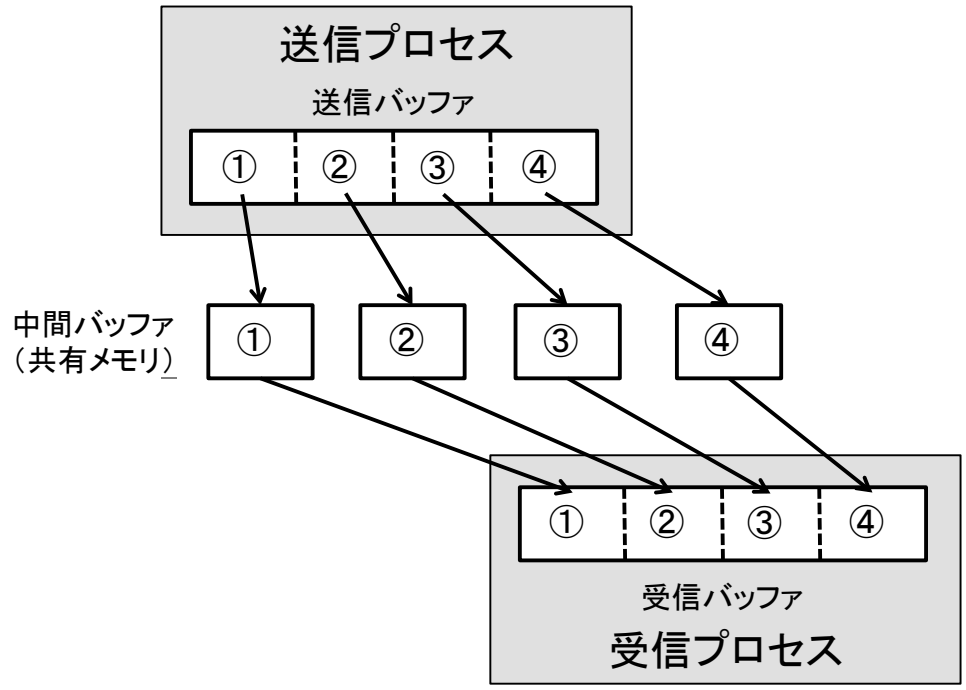


図 **5-4** 中間バッファによるデータの転送

Rendezvous 通信よりも通信遅延が大きくなる．

sm BTL では，パイプライン転送に用いる中間バッファのデフォルトサイズは 32 KB である．中間バッファは，Eager 通信用バッファと同様に再利用され，free list で管理される．パイプライン転送の途中で中間バッファに空きがなくなった場合，新たにメモリプールから共有メモリを取得し，中間バッファの数を増加させる．中間バッファを利用する際の競合を避けるため，各 MPI プロセスは個別の中間バッファをそれぞれ保持する．

sm BTL では共有メモリ上の中間バッファを経由したパイプライン転送の他に，KNEM による Rendezvous 通信もサポートしている．この方式では，まず送信プロセスが送信バッファのアドレスとサイズを，KNEM 用のシステムコールを呼び出すことによって，OS カーネルに登録する．そして，OS カーネルから受け取った送信バッファのディスクリプタを，CTS メッセージに含めて，受信プロセスに通知する．受信プロセスは KNEM 用のシステムコールを実行し，通信メッセージのコピーを OS カーネルに依頼する．この際，送信バッファのディスクリプタと受信バッファのアドレスをシステムコールの引数として OS カーネルに通知する．依頼を受けた OS カーネルは，送信バッファが格納されているメモリを OS カーネルが使用しているアドレス空間にマッピングする．そして，マッピングした領域からデータを受信プロセスの受信バッファにコピーする．データのコピー終了後，マッピングしたメモリはアンマッピングされる．この方式では，1 度のメモリコピーでデータの送受信を終えることができるが，通信のたびにシステムコールを呼び出す必要があり，OS カーネルにコンテキストスイッチへのコンテキストスイッチが発生してしまう．

KNEM による Rendezvous 通信は，基本データ型による連続データの送受信にしか用いることができない．派生データ型を用いる不連続データの送受信には，中間バッファを経由したパイプライン転送が用いられる．

MPICH においても Open MPI と同様に，KNEM を用いた Rendezvous 通信をサポートしている．

5.2.5 共有メモリによる MPI ノード内通信の問題点

本節では，共有メモリによる MPI ノード内通信の問題点について述べる．前節で説明した Open MPI の sm BTL の実装を前提として問題点を述べるが，他の MPI ライブラリの MPI ノード内通信の実装と sm BTL の実装に大きな差は無く，ここで述べる問題点は共有メモリを用いた MPI ノード内通信の実装全般に共通する問題となる．

Rendezvous 通信の通信遅延増加

Rendezvous 通信において, 共有メモリを用いた MPI ノード内通信の実装では, 共有メモリ上の中間バッファを経由したメモリコピーでデータを転送するか, OS カーネルの支援によるメモリコピーでデータを転送していた．これらの方式では，通信の際に以下で述べるオーバヘッドが生じ，通信遅延が増加してしまう．

共有メモリ上の中間バッファを経由したメモリコピーによってデータを転送する場合，データを先頭から順にパイプライン転送する．よって，データサイズが大きくなると，多数のメモリコピーを実行する必要がある．メモリコピー回数の増加に加え，中間バッファを確保する処理や，中間バッファを通信先の通信キューに追加する処理が増え，そのオーバヘッドにより通信遅延が増加してしまう．中間バッファのサイズを大きくすることでメモリコピーの回数を減らすことができるが，その場合，パイプライン転送の段数が少なくなるため，パイプライン転送の効果が小さくなってしまう．

OS カーネルの支援によるメモリコピーでデータを転送する場合は，1 度のメモリコピーで通信を完了することができる．しかし，通信のたびにシステムコー

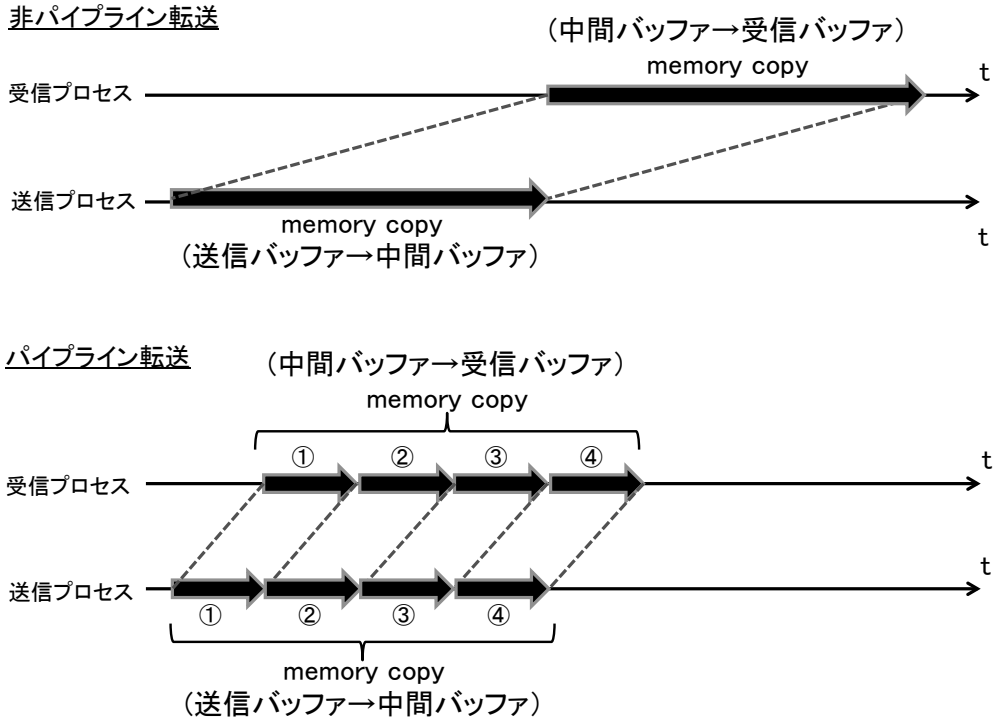


図 **5-5** パイプライン転送

ルを実行する必要があり，そのオーバヘッドにより通信遅延が増加してしまう．

ページテーブルの肥大化によるメモリ消費量増加

既に述べた通り，仮想記憶をサポートする近年の OS では，メモリをページという単位に区切り管理している．各プロセスがどのメモリページを自身のアドレス空間のどの位置にマッピングしているかという情報は，ページテーブルという OS カーネル内の管理テーブルに記録される．CPU はこのページテーブルを参照し，現在実行しているプロセスが使用しているメモリのアドレスを物理メモリ上のアドレスに変換する．

共有メモリを用いた MPI ノード内通信では，通信を行なう双方の MPI プロセスのアドレス空間に共有メモリをマッピングし，マッピングした共有メモリを経由してデータの送受信を行なう．MPI プロセスがマッピングする共有メモリが増加する場合，マッピング情報を記録するページテーブル内のエントリ（ページテーブルエントリ）の数も併せて増加するためにページテーブルが肥大化し，ページテーブルによるメモリ消費量が増加する．共有メモリを用いた MPI ノード内通信では，以下のデータを格納する共有メモリを，各 MPI プロセスが自身のアドレス空間にマッピングする必要がある．

- 通信先の通信キュー
- 通信先の Eager 通信用バッファ
- 通信先の Rendezvous 通信用の中間バッファ

同一ノード内で動作する N 個の MPI プロセスが全対全の通信を行う場合，2.1.2 節で述べた通り N^2 のオーダのメモリマッピングが生成される．よって，システム全体のページテーブルによるメモリ消費量も N^2 のオーダで増加する．1 ノードあたりのコア数および MPI プロセス数は今後も増加していくことが予想されるため，N^2 のオーダで増加するページテーブルのメモリ消費量は，メニーコア環境では大きな問題となる．

MPI ライブラリ自体のメモリ消費量を削減するための研究は既に行なわれている[53-54]．しかし，MPI ノード内通信によるページテーブルサイズの増加に着目した研究は，まだ実施されていない．

5.3 MPI ノード内通信への PVAS タスクモデルの適用

前章で述べた通り，共有メモリを用いた MPI ノード内通信には，Rendezvous 通信の通信遅延増加とページテーブルサイズの肥大化によるメモリ消費量の増加という問題がある．この通知遅延の増加とメモリ消費量の増加はアドレス空間をまたいでデータ転送するために発生している．MPI プロセスがそれぞれ個別のアドレス空間で動作するため，Rendezvous 通信において，送信プロセスの受信バッファから送信プロセスのバッファに直接メッセージをコピーすることができず，通信遅延が増加していた．また，MPI プロセスがそれぞれ個別のアドレス空間で動作するため，通信先のプロセスが使用している共有メモリを，各 MPI プロセスが自身のアドレス空間にマッピングしてデータの送受信を行なう必要があり，ページテーブルサイズの肥大化によるメモリ消費量増加を招いていた．

もし，MPI プロセスが同一アドレス空間で動作するならば，アドレス空間をまたいでデータ転送を行う必要がなくなり，上記の問題が発生するのを回避することができる．本研究では，PVAS タスクモデルを用いて MPI プロセスを同一アドレス空間で実行し，アドレス空間をまたいでデータ転送することなく，MPI ノード内通信を実行可能とする．これにより，よりメニーコア環境に適した，高速かつメモリ消費量の少ない MPI ノード内通信を実現する．

本章では，PVAS タスクモデルを利用したメニーコア環境向け MPI ノード内通信の実装について述べる．実装は Open MPI を対象として行なったが，他の MPI ランタイムに対しても，同様の方式で実装可能であると考える．

5.3.1 プロセスマネージャ

MPI アプリケーションを PVAS プロセスに実行させることで，PVAS プロセスとして動作する MPI プロセスを起動することができる．PVAS プロセスとして同一アドレス空間で動作する MPI プロセス同士は，互いのメモリにアクセスできる．

通常，Open MPI のプロセスマネージャは，MPI アプリケーションの実行コマンドが実行されたら，fork によって，ユーザが指定した数だけプロセスを生成する．生成されたプロセスは，ユーザに指定された MPI アプリケーションを execve によって実行し，MPI プロセスとなる．

対して，PVAS タスクモデルを用いた MPI ノード内通信をサポートする Open MPI のプロセスマネージャは，まず，pvas_create によって，PVAS プロセスを生

成する PVAS アドレス空間を作成する．そして，pvas_spawn を呼び出して，ユーザが指定した数だけ当該 PVAS アドレス空間に PVAS プロセスを生成する．プロセスマネージャは，ユーザに指定された MPI アプリケーションを pvas_spawn の引数とし，PVAS プロセスに実行させる．MPI アプリケーションを実行する PVAS プロセスは MPI プロセスとなる．PVAS プロセスとして動作する全 MPI プロセスの実行が終了したら，プロセスマネージャは，pvas_destroy を呼び出し，不要になった PVAS アドレス空間を削除する．

5.3.2 PVAS BTL

PVAS タスクモデルを用いた MPI ノード内通信をサポートする BTL モジュールを，sm BTL をベースに実装した．このモジュールを PVAS BTL とする．本節では，PVAS BTL の実装について述べる．

Eager 通信

まず，PVAS BTL における Eager 通信によるデータ送受信について述べる．PVAS BTL では，図 5-6 に示すように，通信キューと Eager 通信用のバッファを取得するメモリプールを，最初に起動した MPI プロセスが malloc によって作成する．

並列処理を実行する各 MPI プロセスは，このメモリプールから，自身が用いる通信キューと Eager 通信用バッファを取得する．PVAS BTL では，並列処理を行なう MPI プロセス群が同一アドレス空間内の PVAS プロセスとして動作している．MPI プロセス同士は，互いのメモリにアクセスすることができるので，どの MPI プロセスも，このメモリプールからメモリを取得することができる．

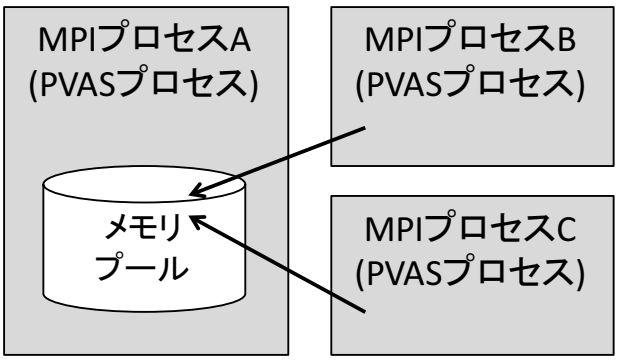


図 **5-6 PVAS BTL** のメモリプール

また，メモリプールから取得したメモリにはどの MPI プロセスからでもアクセス可能である．よって，このメモリプールから取得した通信キューと Eager 通信用バッファを用い，sm BTL と同様の方法で，Eager 通信によるデータの送受信を行なうことができる．メモリプールの作成方法以外は sm BTL の実装を流用するため，Eager 通信によるデータの送受信の性能は，sm BTL と同等となる．

Rendezvous 通信（連続データ）

次に，Rendezvous 通信による連続データの送受信について述べる．PVAS BTL では，並列処理を行なう MPI プロセス同士が互いのメモリにアクセスすることができる．よって，PVAS BTL における Rendezvous 通信では，受信プロセスが，送信プロセスの送信バッファから自身の受信バッファにデータを直接コピーすることで通信を行なう．送信プロセスが CTS メッセージと共に送信バッファのアドレスと送受信するデータのサイズを受信プロセスに通知し，受信プロセスは通知されたアドレスから，データを受信バッファにコピーする．図 5-7 に示すように，1 度のメモリコピーで通信を完了することができる．

Rendezvous 通信（不連続データ）

次に，Rendezvous 通信による不連続データの送受信について述べる．不連続データのときも，連続データのときと同様に，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．

連続データを送受信する際は，送信バッファのアドレスとデータサイズを通知すれば，受信プロセスが送信バッファから受信バッファにデータをコピーすることができた．しかし，不連続データを送受信する場合はそれに加えて，送信対象のデータが送信バッファにどのように配置されているかというレイアウト情報を受信プロセスに通知する必要がある．

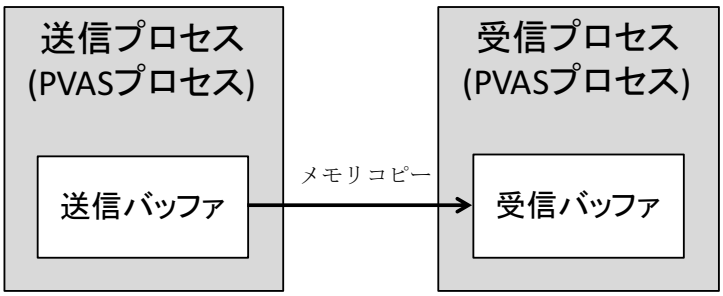


図 **5-7 PVAS BTL** の **Rendezvous** 通信

Open MPI では，通信に用いるデータ型の情報を *converter* と呼ぶオブジェクトの中に格納している．converter を参照すれば，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るかというレイアウト情報を取得することができる．PVAS BTL では，CTS メッセージに送信バッファのアドレス，送信するデータサイズ，送信するデータのデータ型に対応する converter のアドレスを含めて，受信プロセスに送信する．CTS メッセージを受信した受信プロセスは，送信側の converter と自身が保持する受信側の converter を参照して，送受信対象データの配置レイアウトを確認し，送信バッファから自身の受信バッファにデータをコピーしていく．図 5-8 は，受信プロセスが送信バッファ上の不連続データを自身の受信バッファにコピーする様子を示している．

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 は送信プロセスが使用するメモリ上に存在するが，PVAS を用いる場合は，送信側の converter のアドレスさえ受信プロセスに通知すれば，受信プロセスが送信側の convertor にアクセスすることができる．

通信性能について

既に述べたように，共有メモリを用いた MPI ノード内通信の実装では，中間バッファを経由したデータ転送によるオーバヘッド，またはシステムコールの実行によるオーバヘッドで通信遅延が増加していた．PVAS BTL の Rendezvous 通信では，これらのオーバヘッドは発生しないため，より高速なデータの送受信を実現することができる．

また，通信時に中間バッファにアクセスする必要が無くなるため，アプリケー

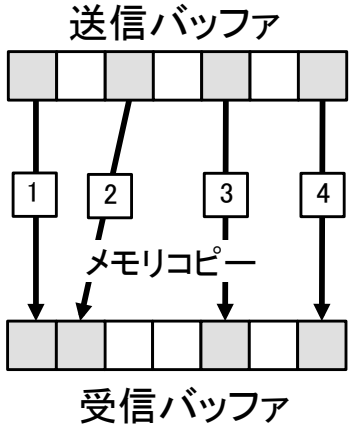


図 **5-8** 不連続データのコピー

ション実行時にアクセスするメモリの範囲が狭まり，L1/L2 キャッシュ，TLB キャッシュ等の CPU キャッシュの利用効率が改善され，アプリケーションの実効性能が向上することも期待できる．

メモリ消費量について

共有メモリを用いた MPI ノード内通信では，通信先のプロセスが使用している共有メモリを通信元のプロセスが自身のアドレス空間にマッピングして，MPI ノード内通信を実行するために必要なデータの送受信を行なっていた．よって，共有メモリのマッピングによりページテーブルが肥大化し，メモリ消費量が増加していた．PVAS BTL では，並列処理を行なう MPI プロセス同士が，互いのメモリにアクセスすることができるため，共有メモリを用いずとも，MPI ノード内通信の実行に必要なデータの送受信を行うことができる．従って，共有メモリのマッピングによるページテーブルの肥大化を回避し，その分のメモリ消費量を削減することができる．

さらなる最適化に向けて

本研究では，プロセス間のデータ転送を CPU の load/store 命令を用いたメモリコピーのみによって実装したが，DMA によるプロセス間のデータ転送を利用することも考えられる．一般的に，大きなサイズのデータ転送においては，DMA の方が CPU によるメモリコピーよりも高速であるため，DMA によるデータ転送を用いることで，より高速なノード内通信を実現できる可能性がある．Xeon Phi コプロセッサでは，DMA による CPU コア間のデータ転送をサポートしている．DMA のチャネル・コントローラの数が限られているため，数 100 の並列プロセスが同時に DMA を利用することはできない．しかし，すぐに DMA を利用できない場合は CPU のメモリコピーを利用するなど，CPU のメモリコピーと DMA を上手く併用することで，ノード内通信をさらに最適化できる可能性がある．

第6章　評価

PVAS タスクモデルを適用した MPI ノード内通信の性能とメモリ消費量を，マイクロベンチマークと MPI アプリケーションによって評価し，既存の MPI ノード内通信を用いる場合と比較した．評価環境を表 6-1 に示す．性能の評価については，連続データの送受信を行う場合と，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の双方について，評価を行なった．

6.1　性能評価（連続データの送受信）

6.1.1　マイクロベンチマーク（Intel MPI Benchmarks）

PVAS BTL と sm BTL において，連続データの送受信を行う場合の通信性能を，Intel MPI Benchmarks (IMB)[46]を用いて比較した．IMB には，MPI 通信の性能を評価するための種々のベンチマークが含まれている．本評価では，MPI_Send と MPI_Recv を用いて ping-pong 通信を行なうベンチマークによって 1 対 1 通信の性能を測定した．

また，MPI_Bcast によるブロードキャスト通信を行なうベンチマークと，MPI_Allreduce によるリダクション通信を行なうベンチマークによって，集団通信の性能を測定した．

表 6-1 評価環境

項目	内容	
プロセッサ	Intel Xeon Phi コプロセッサ 5110P	
	コア数	60
	論理コア数	240
	メインメモリ	8 GB
	L1 キャッシュ	32 KB
	L2 キャッシュ	512 KB
OS カーネル	Linux カーネル (MPSS 3.1.2) + PVAS	
MPI ランタイム	Open MPI version 1.8 + PVAS BTL	

集団通信の測定を行なう際の MPI プロセス数は 240 プロセスとした．sm BTL の Rendezvous 通信においては，共有メモリ上の中間バッファを用いてデータのパイプライン転送を行なう場合と，KNEM カーネルモジュールを用いて OS カーネルの支援によりデータの送受信を行なう場合の 2 通りを測定した．共有メモリ上の中間バッファを用いてデータのパイプライン転送を行なう際の中間バッファのサイズは，sm BTL のデフォルト値である 32 KB とした．

1 対 1 通信の通信遅延の測定結果を図 6-1 と図 6-2 のグラフに示す．図 6-1 は Eager 通信の測定結果を，図 6-2 は Rendezvous 通信の測定結果を示す．Eager 通信の測定においては，全てのデータサイズにおいて Eager 通信が実行されるように，eager limit の値を充分大きな値にした．Rendezvous 通信の測定においては，全ての通信が Rendezvous 通信で実行されるよう，eager limit の値を最小値に設定した．横軸は送受信するデータサイズを，縦軸は通信遅延を示している．

Eager 通信による 1 対 1 通信では，PVAS BTL と sm BTL は，ほぼ同等の通信性能を示している．既に述べた通り，PVAS BTL の Eager 通信は sm BTL の実装を流用している．Eager 通信用バッファを格納するメモリが共有メモリか通常の

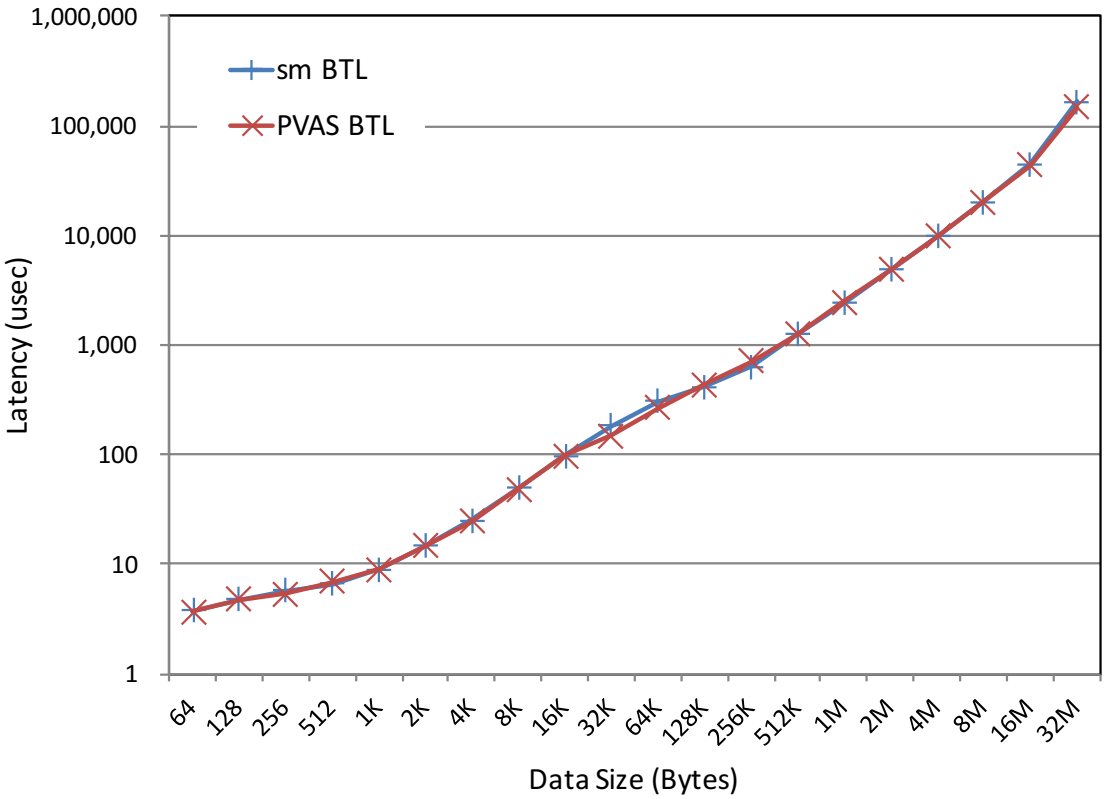


図 **6-1 1** 対 **1** 通信（**Eager** 通信）

メモリかという違いしかないため，通信性能はほぼ同等となる．

Rendezvous 通信による 1 対 1 通信では，PVAS BTL がもっとも通信遅延が小さい．sm BTL の Rendezvous 通信では，共有メモリ上の中間バッファを経由してデータのパイプライン転送を行なう．データサイズがパイプライン転送に用いる中間バッファのサイズ（32 KB）より小さい場合は，送信プロセス側のメモリコピーと受信プロセス側のメモリコピーのオーバラップを行なうことができず，1 度のメモリコピーで通信が終わる PVAS BTL よりも通信遅延が大きくなる．メッセージサイズが中間バッファのサイズより大きい場合は，パイプライン転送により，送信プロセス側のメモリコピーと受信プロセス側のメモリコピーをオーバラップすることが可能になるが，中間バッファを経由したデータ転送の実行回数自体は増加する．メモリコピーを実行する処理や，中間バッファを確保するための処理，データをコピーした中間バッファを受信プロセスの通信キューに追加する処理等の回数が増え，そのオーバヘッドにより，PVAS BTL よりも通信遅延が大きくなる．KNEM を用いて OS カーネルの支援によるデータの送受信を行う場合は，PVAS BTL と同様に 1 度のメモリコピーで通信が完了するが，通信のたびにシステムコールが実行されるため，そのオーバヘッドによ

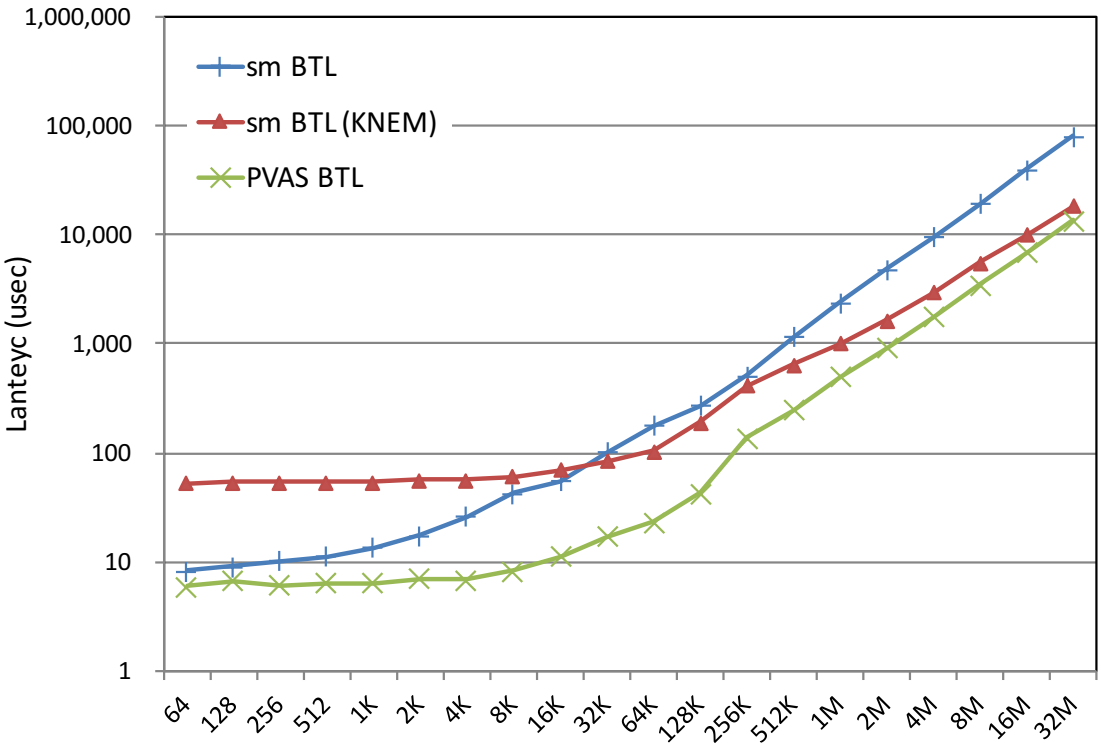


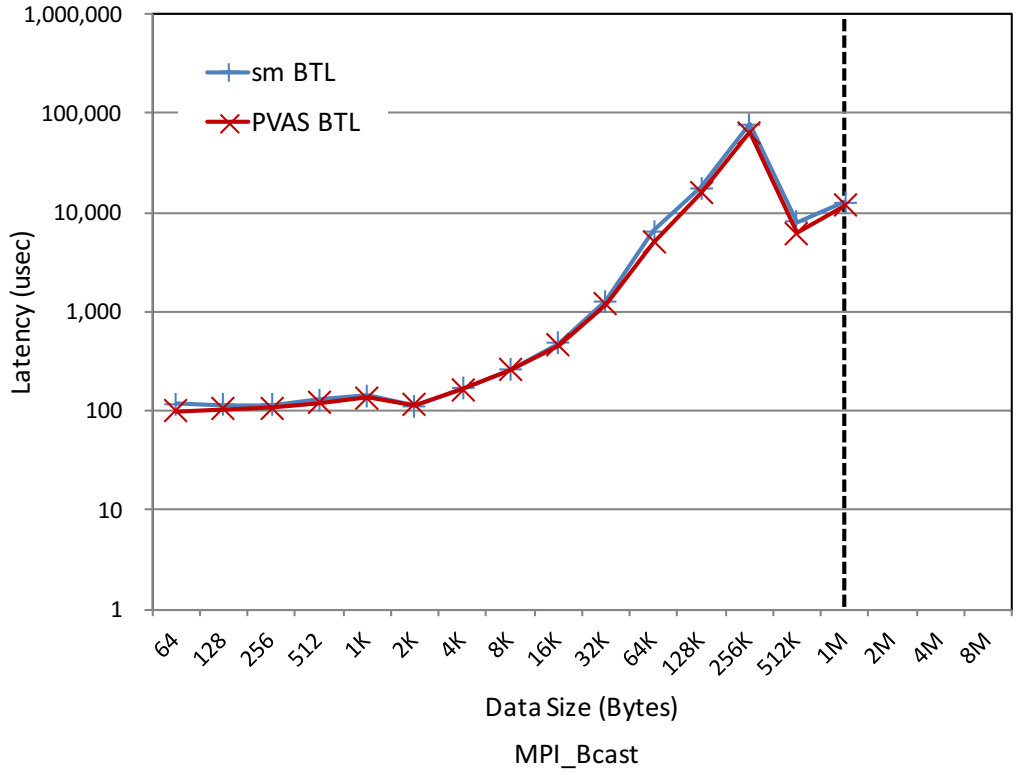
図 **6-2 1** 対 **1** 通信（**Rendezvous** 通信）

り，通信遅延が PVAS BTL よりも大きくなる．

sm BTL において，共有メモリ上の中間バッファを経由してパイプライン転送を行なう場合と KNEM を用いて OS カーネルの支援によるメッセージの送受信を行う場合を比較すると，データサイズが小さい場合はシステムコール実行のオーバヘッドにより，OS カーネルの支援によるデータの送受信の方が，通信遅延が大きくなる．データサイズが大きくなると，中間バッファを経由したデータ転送の回数が増加し，そのオーバヘッドがシステムコール実行のオーバヘッドを上回り，中間バッファを経由したパイプライン転送の方が，通信遅延が大きくなる．

図 6-3 と図 6-4 に，集団通信の通信遅延の測定結果を示す．集団通信を行なう場合はプロセス数が増加し，各 MPI プロセスが持つ通信バッファによってシステム全体のメモリの消費量が大きくなる．表 6-1 の測定環境において，ベンチマークを実行可能な最大データサイズは Rendezvous 通信では 8 MB となった．また, 集団通信を Eager 通信で行なうと 1 度の通信で多数の Eager 通信用バッファが必要となり，さらにメモリを消費する．このため，Eager 通信においてベンチマークを実行可能な最大データサイズは 1 MB となった

1 対 1 通信のときと同様の理由で，Eager 通信は，PVAS BTL と sm BTL で，ほぼ同等の通信性能となった．Rendezvous 通信では，1 対 1 通信のときと同様の理由で，PVAS BTL が最も高い通信性能を示している．集団通信では，データサイズが大きくなっても，KNEM を用いて OS カーネルの支援によるデータの送受信を行なう場合の方が，共有メモリ上の中間バッファを用いてパイプライン転送する場合よりも，通信遅延が大きくなっている．Open MPI の MPI_Bcast, MPI_Allreduce の実装では，通信のバンド幅を改善するため，図 6-5 に示すように，送受信するデータを細かいチャンクに分割して，各プロセスが協調して並列に通信を行なう通信アルゴリズムとなっている．実際に各 MPI プロセスが送受信するデータサイズは小さくなるため，KNEM を用いて OS カーネルの支援によるデータの送受信を行なう方が，通信遅延が大きくなってしまう．



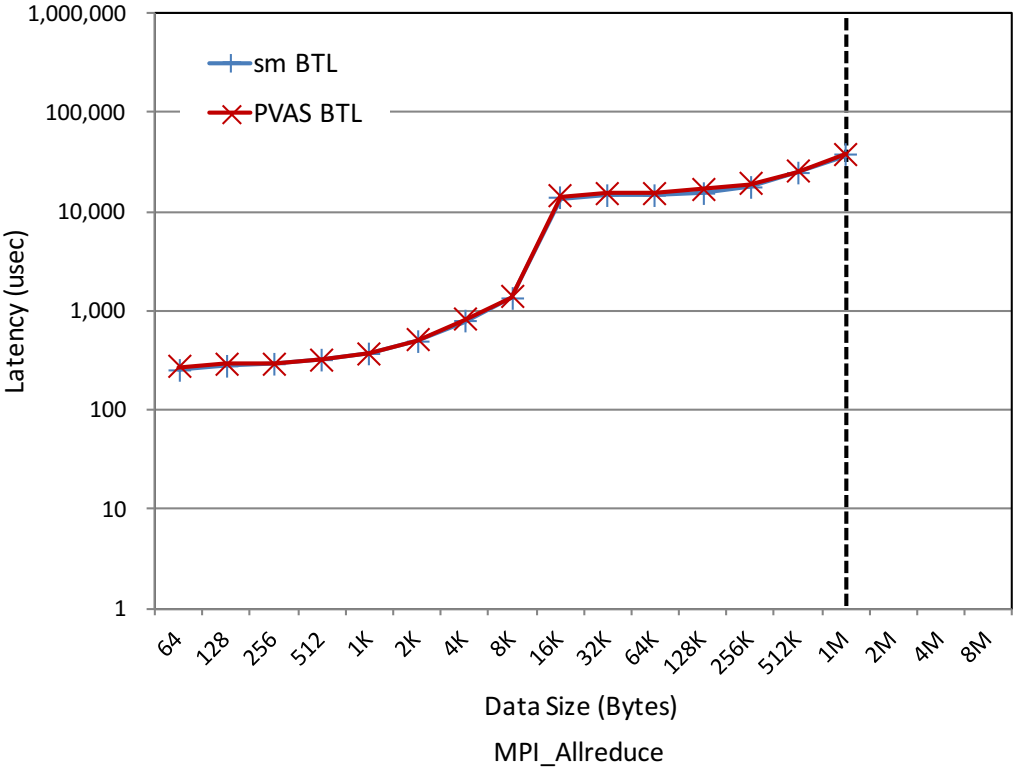
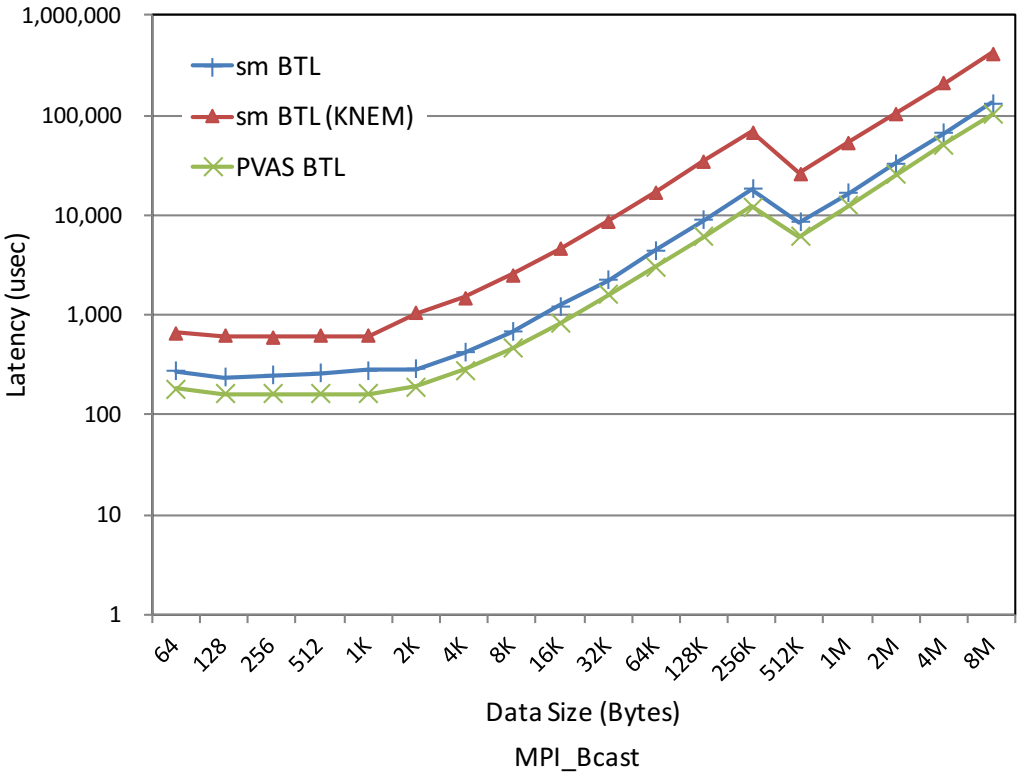


図 **6-3** 集団通信（**Eager** 通信）



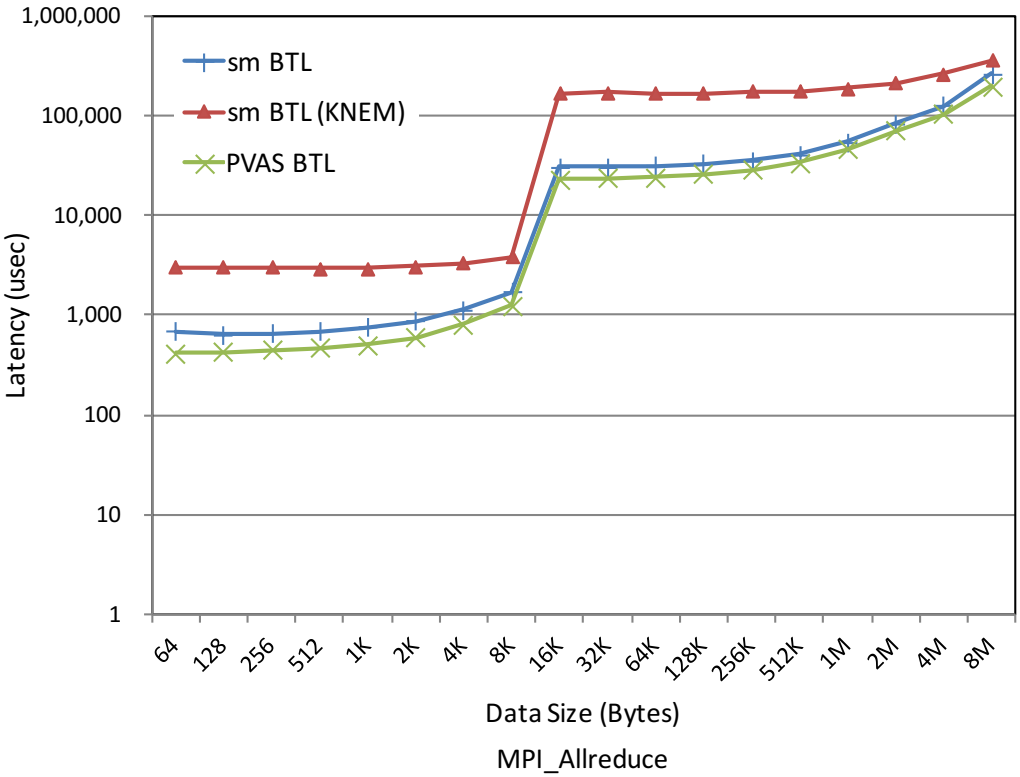


図 **6-4** 集団通信（**Rendezvous** 通信）

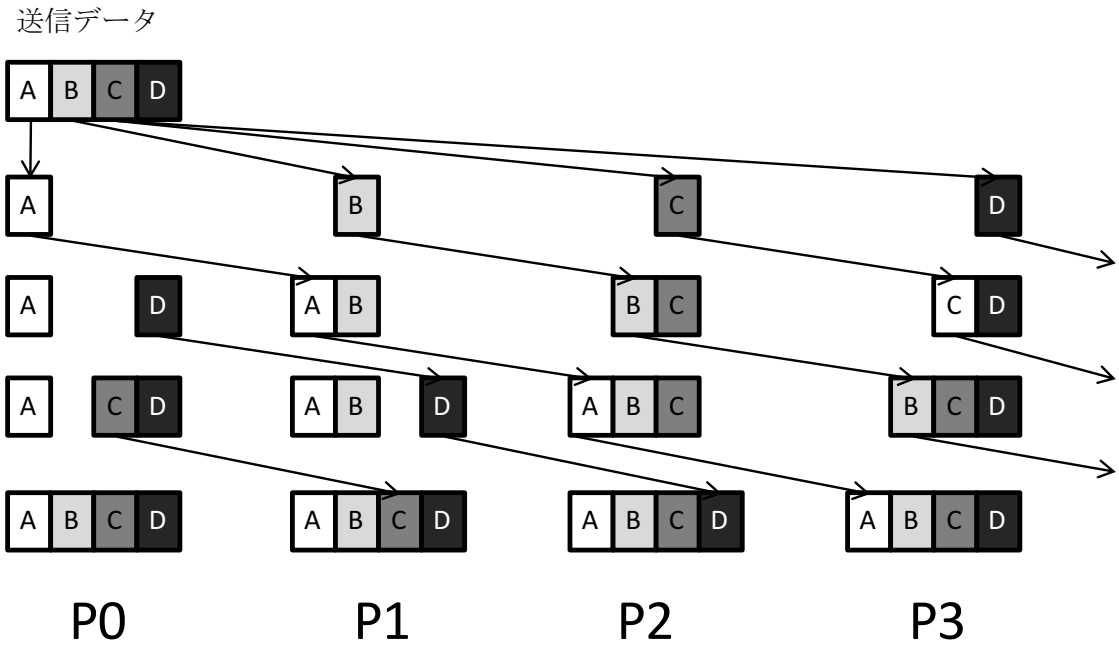


図 **6-5** 集団通信のアルゴリズム

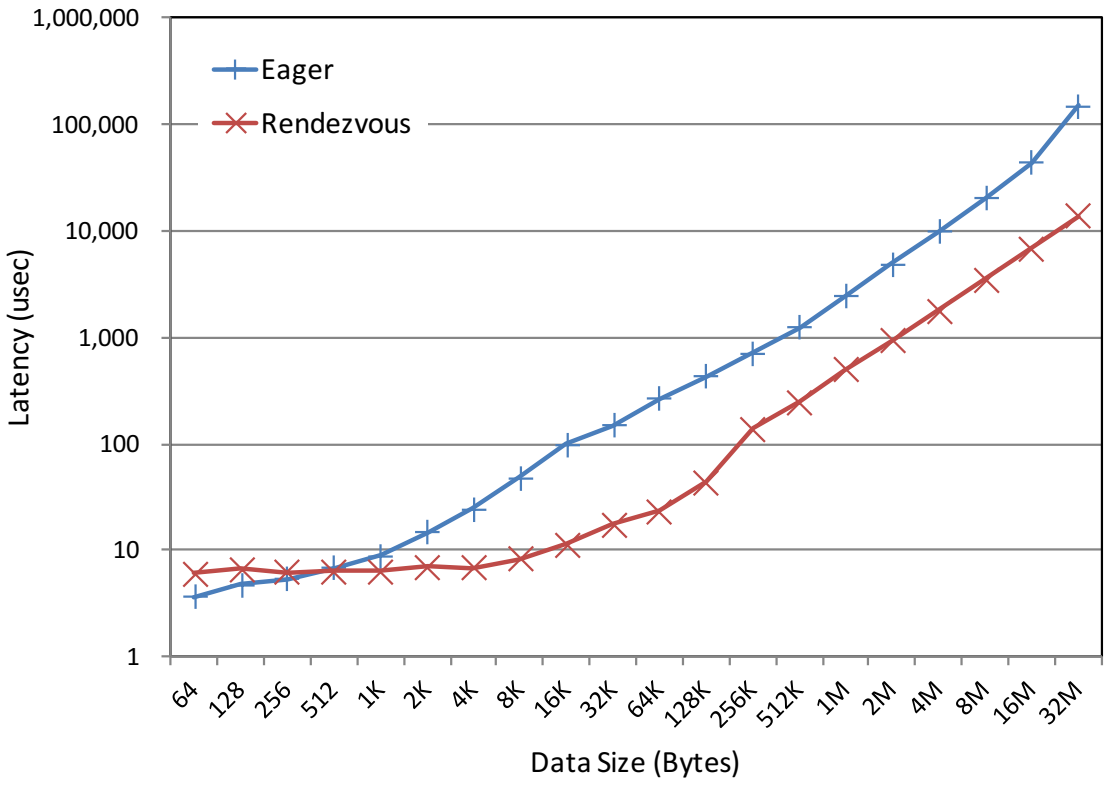


図 **6-6 Eager** 通信と **Rendezvous** 通信の比較

表 6-2 NAS Parallel Benchmarks

アプリ	実行する計算処理
EP	乗算合同法による一様乱数，正規乱数の生成
MG	簡略化されたマルチグリッド法のカーネル
CG	正値対称な大規模疎行列の最小固有値を求めるための共役勾配法
FT	高速フーリエ変換を用いた 3 次元偏微分方程式の解法
IS	大規模整数ソート
LU	Symmetric SOR iteration による CFD アプリケーション
SP	Scalar ADI iteration による CFD アプリケーション
BT	5x5 block size ADI iteration による CFD アプリケーション

図 6-6 は PVAS BTL の Eager 通信と Rendezvous 通信を比較したグラフである．グラフには，1 対 1 通信の通信遅延を示している．グラフに示す通り，送受信するデータサイズが 512 Bytes 以下の場合は，同期のコストが無いぶん，Eager 通信の方が，通信遅延が小さくなっている．しかし，データサイズが 1 KB 以上になると，Eager 通信用バッファを経由せず，送受信バッファ間で直接データをコピーできる Rendezvous 通信の方が，通信遅延が小さくなる．

6.1.2 ミニアプリケーション（NAS Parallel Benchmarks）

次に，NAS Parallel Benchmarks (NPB)[47]を用いて，PVAS BTL と sm BTL の性能を評価した. NPB には, MPI を用いて実装された科学計算のためのアプリケーションが含まれており，MPI アプリケーションを実行するシステムの評価に広く用いられている．NPB では，通信と計算処理を含めた総合的な実行性能の評価を行なうことができる．NPB に含まれるアプリケーションを表 6-2 に示す．NPB の各アプリケーションは，連続データを MPI 通信によって送受信する．

本評価では，通信が発生しない EP 除く，7 つのアプリケーションで性能測定を行なった．それぞれのアプリケーションに対し，問題サイズが異なる 7 つのクラス S，W，A，B，C，D，E (S<W<A<B<C<D<E)が用意されている．性能測定では，標準的な問題サイズとされるクラス A，B，C を用いた．ただし，FT のみ，メモリ不足により，クラス C では測定を行うことができなかった．MG，CG，FT，IS，LU においては，プロセス数が 2 の累乗でなければならないという制限のため，プロセス数を 128 としてアプリケーションを実行した．SP，BT においては，プロセス数が任意の数の累乗でなければならないという制限があ

るため，プロセス数を 225 としてアプリケーションを実行した．

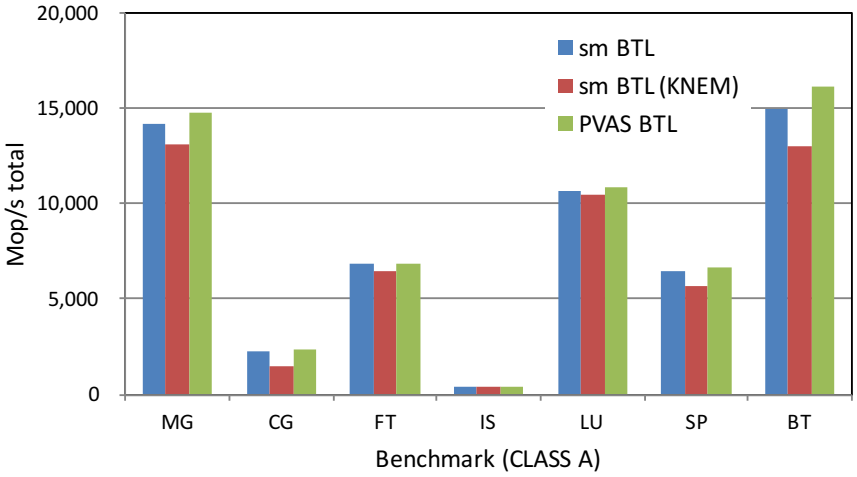
測定は，eager limit を sm BTL のデフォルト値である 4 KB に設定して行なった場合と，全てのデータの送受信を Rendezvous 通信によって行う場合について実施した．eager limit の値を最小値とすることで，全てのデータの送受信を Rendezvous 通信で行うことができる．sm BTL については，共有メモリ上の中間バッファを用いてデータのパイプライン転送を行なう場合と，KNEM を用いて OS カーネルの支援によるデータの送受信を行なう場合の 2 通りを評価した．共有メモリ上の中間バッファを用いてデータのパイプライン転送を行なう際の中間バッファのサイズは，sm BTL のデフォルト値である 32 KB とした．

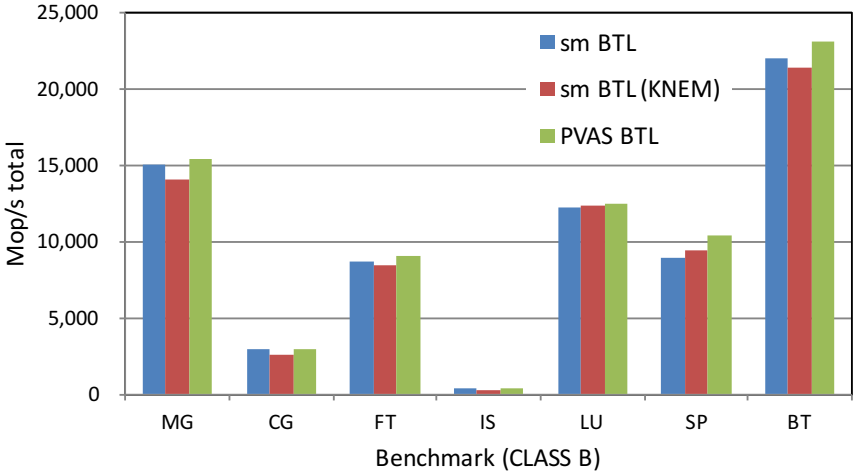
eager limit を 4 KB に設定したときの実行結果を図 6-7 に示す．また，図 6-8 に，Rendezvous 通信において，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行なう場合の sm BTL の実行性能を基準としたときの，相対的な性能差をグラフ中に示した．

ベンチマークの実行時間のうち，通信処理の時間よりも計算処理の割合が大きい場合や，通信の大部分を Eager 通信が占める場合をのぞき，PVAS BTL が他と比べて高い実行性能を示している．これは前節で述べた通り，PVAS BTL では，他と比べて高速な Rendezvous 通信を実現可能なためである．eager limit を 4 KB に設定したケースでは，最大で約 11% の性能向上を実現することができた（SP のクラス B）．

全ての通信が Rendezvous 通信で行なわれる場合の実行結果を図 6-9 に示す．また，図 6-10 に，Rendezvous 通信において，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行なう場合の sm BTL の実行性能を基準としたときの，相対的な性能差をグラフ中に示した．図 6-10 に示すように，全ての通信を Rendezvous 通信で実行した場合は，性能差はより顕著になる．この場合，最大で約 18%の性能向上を実現することができた（SP のクラス B）．

sm BTL において，KNEM を用いて OS カーネルの支援によるデータの送受信を行なう場合，共有メモリ上の中間バッファを用いてデータのパイプライン転送を行なう場合よりも全体的に実行性能が低下している．全ての通信を Rendezvous 通信で行なう場合，性能低下の比率はより大きくなる．これは，ほとんどのベンチマークで，送受信するデータのサイズが小さいためである．データサイズが小さい場合は，KNEM を用いて OS カーネルの支援によるデータの送受信を行なった方が，Rendezvous 通信の通信遅延が大きくなる．送受信するデータのサイズが大きい SP, BT クラス C では，逆に実行性能が向上している．





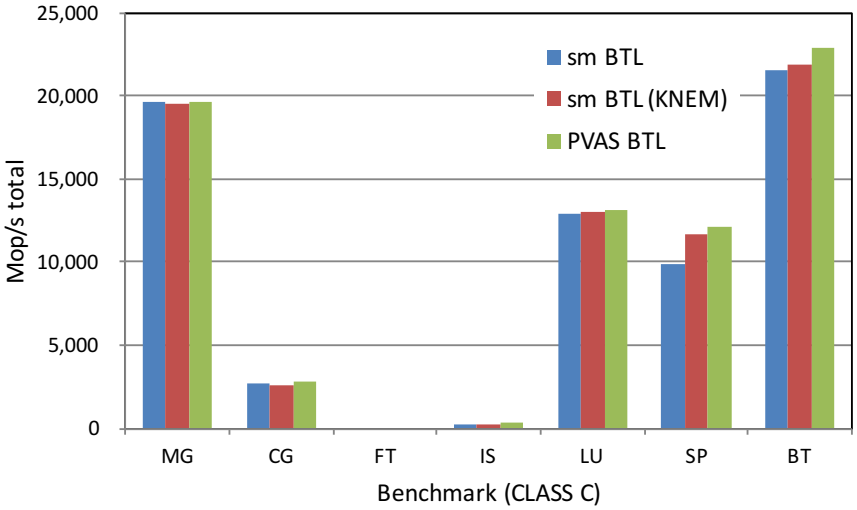


図 6-7 NPB（eager limit = 4 KB）

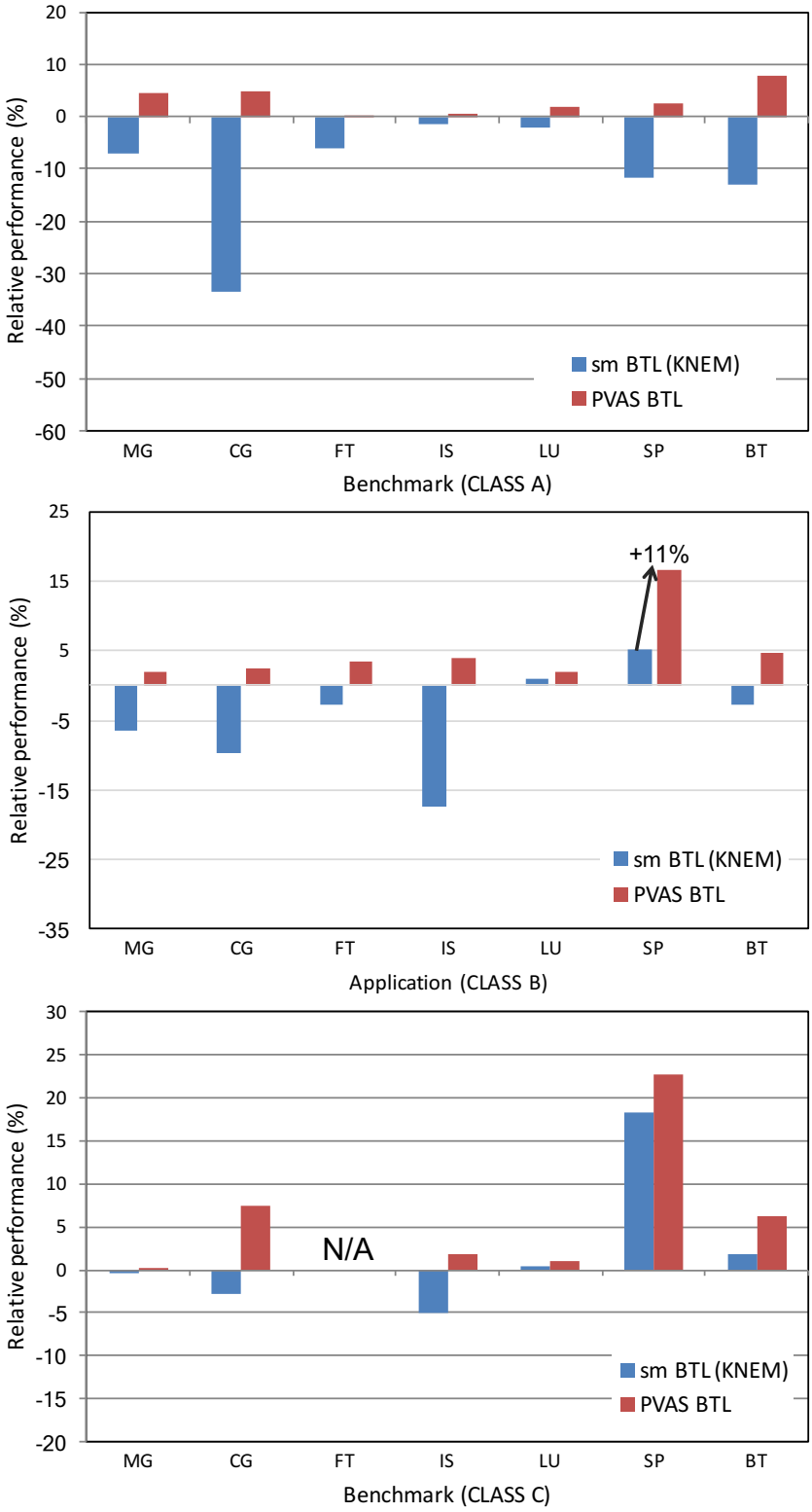
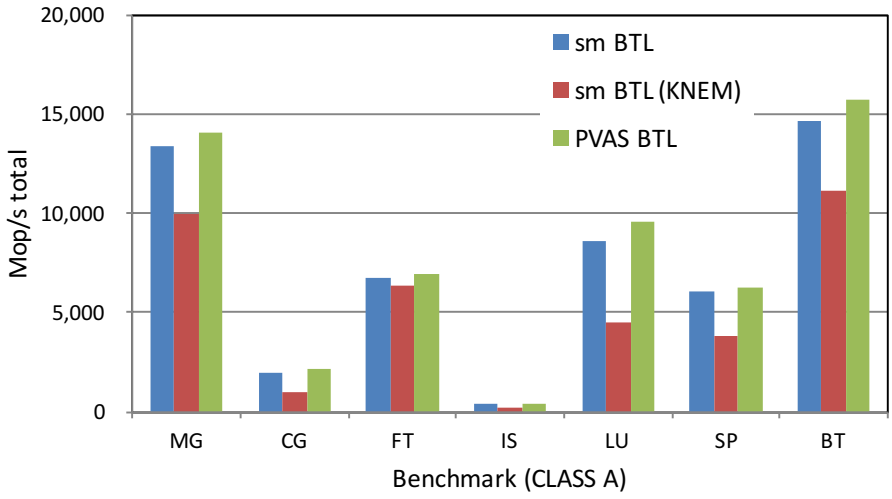
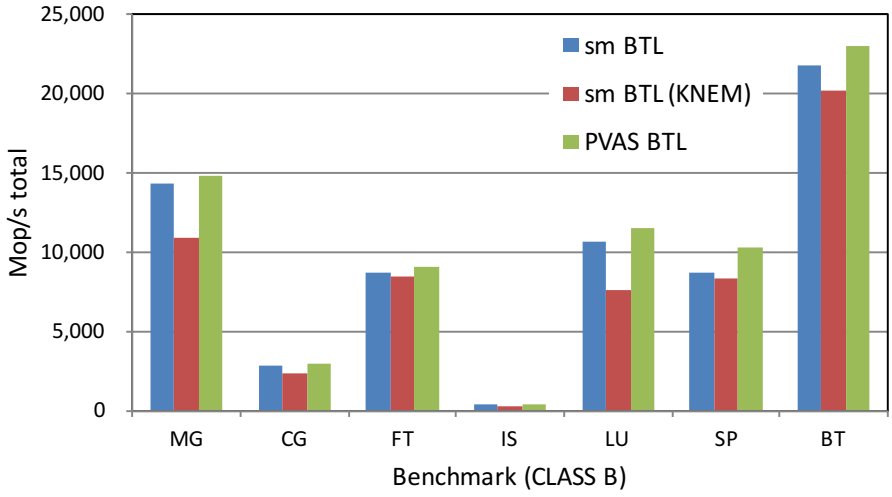


図 **6-8 NPB** における相対性能（**eager limit = 4 KB**）





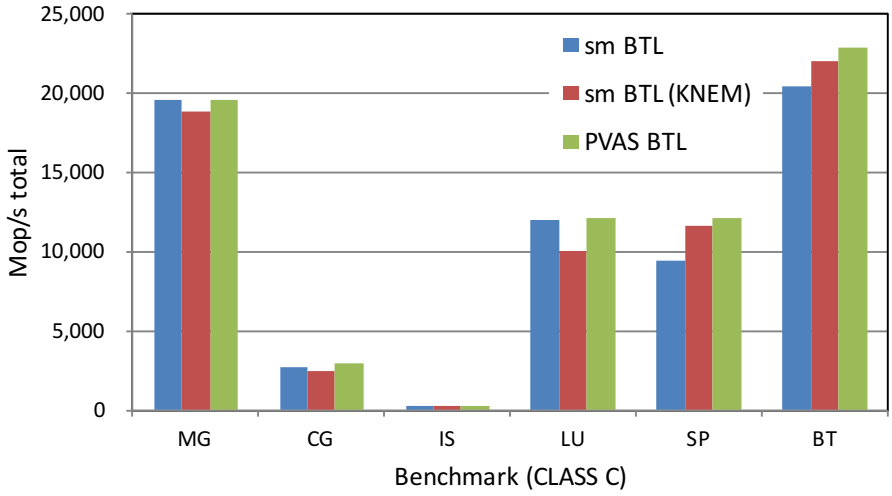


図 **6-9 NPB（eager limit = 0）**

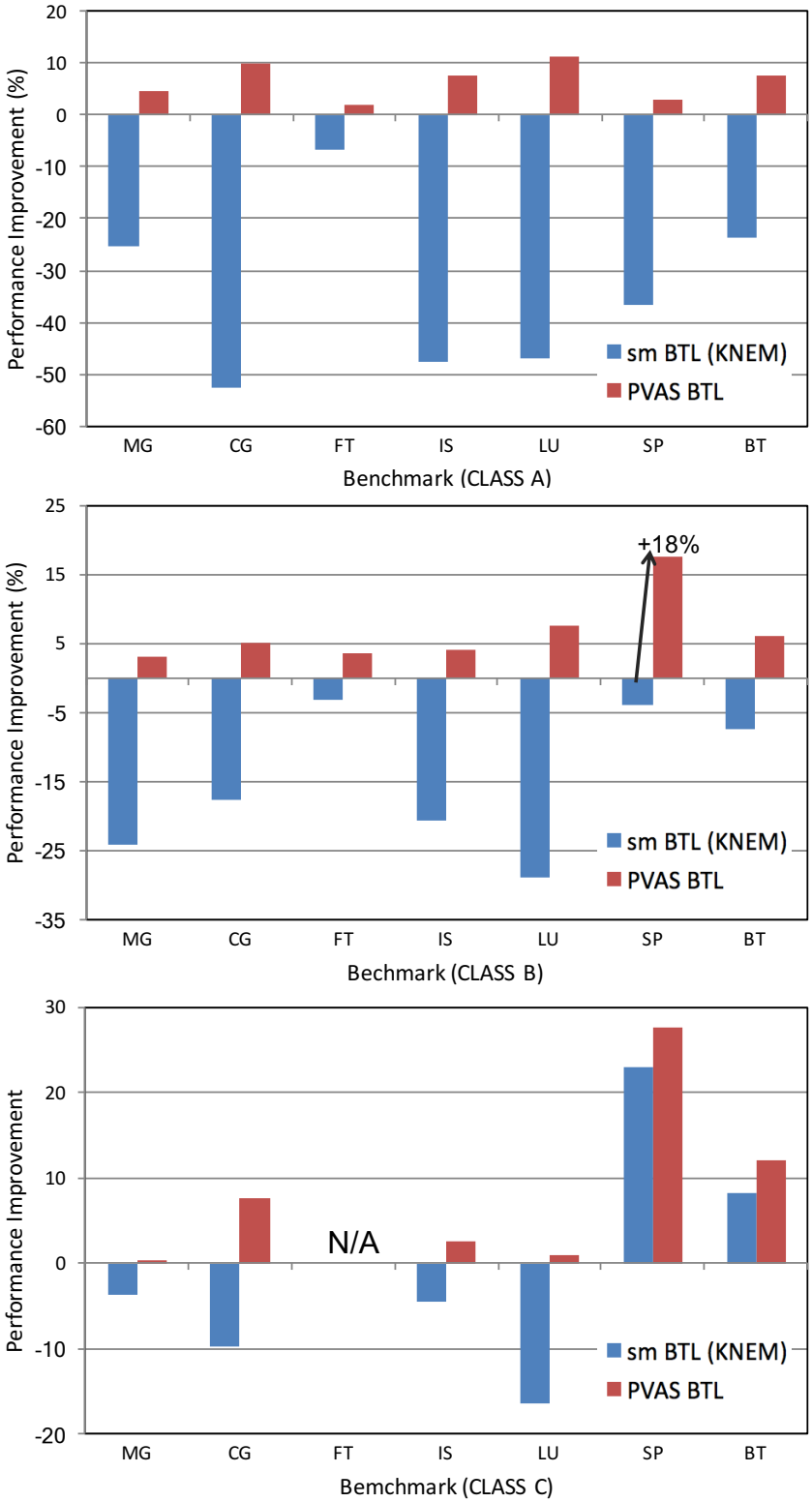


図 **6-10 NPB** における相対性能（**eager limit = 0**）

表 6-3 DDTBench の再現する通信パターン（文献[48]より抜粋）

App.	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

6.2 性能評価（不連続データの送受信）

6.2.1 マイクロベンチマーク（DDTBench）

DDTBench[48]により，派生データ型を用いて不連続なデータの送受信する場合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 アプリケーション（WRF[49]，SPECFEM3D_GLOBE[50]，MILC[51]，LAMMPS[52]，NPB の LU と MG）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．

NPB の LU と MG は，pack/unpack 処理によって不連続なデータを連続データとして送受信する. 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）を行なってから送信し，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．よって，NPB の LU と MG の通信は連続データの送受信となる．DDTBench では，派生データ型による不連続なデータの送受信によって LU と MG を実装したと仮定して，その通信パターンを再現する．

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なわない. 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 6-3 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，sm BTL と PVAS BTL の通信遅延を測定した．測定は，Rendezvous 通信を対象として行なった．また，sm BTL では不連続なデータの送受信を，KNEM を用いて実行することはできないため，中間バッファを経由したパイプライン転送を用いる実装のみ，測定を行なった．

DDTBench による通信遅延の測定結果を図 6-11 と図 6-12 に示す．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 limit の値を調整した．また，DDTBench では，送受信するデータサイズが小さいベンチマークが多いため，sm BTL 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 KB から 64 KB に設定して測定を行なった．sm BTL の結果を SM，PVAS BTL の結果を PVAS として，グラフに表示した．

PVAS BTL は，連続データの送受信と同様に，NAS_MG_x を除いて，sm BTL よりも通信遅延が小さくなった．NAS_MG_x では，PVAS BTL と sm BTL 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 KB および 32 KB のときは，sm BTL よりも PVAS BTL の方が，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 BTL の通信遅延が sm BTL よりも大きくなってしまう理由を以下に述べる．

図 6-13 は，NAS_MG_x のソースコードにおいて，データ型の定義を行な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 6-14 のようになる．

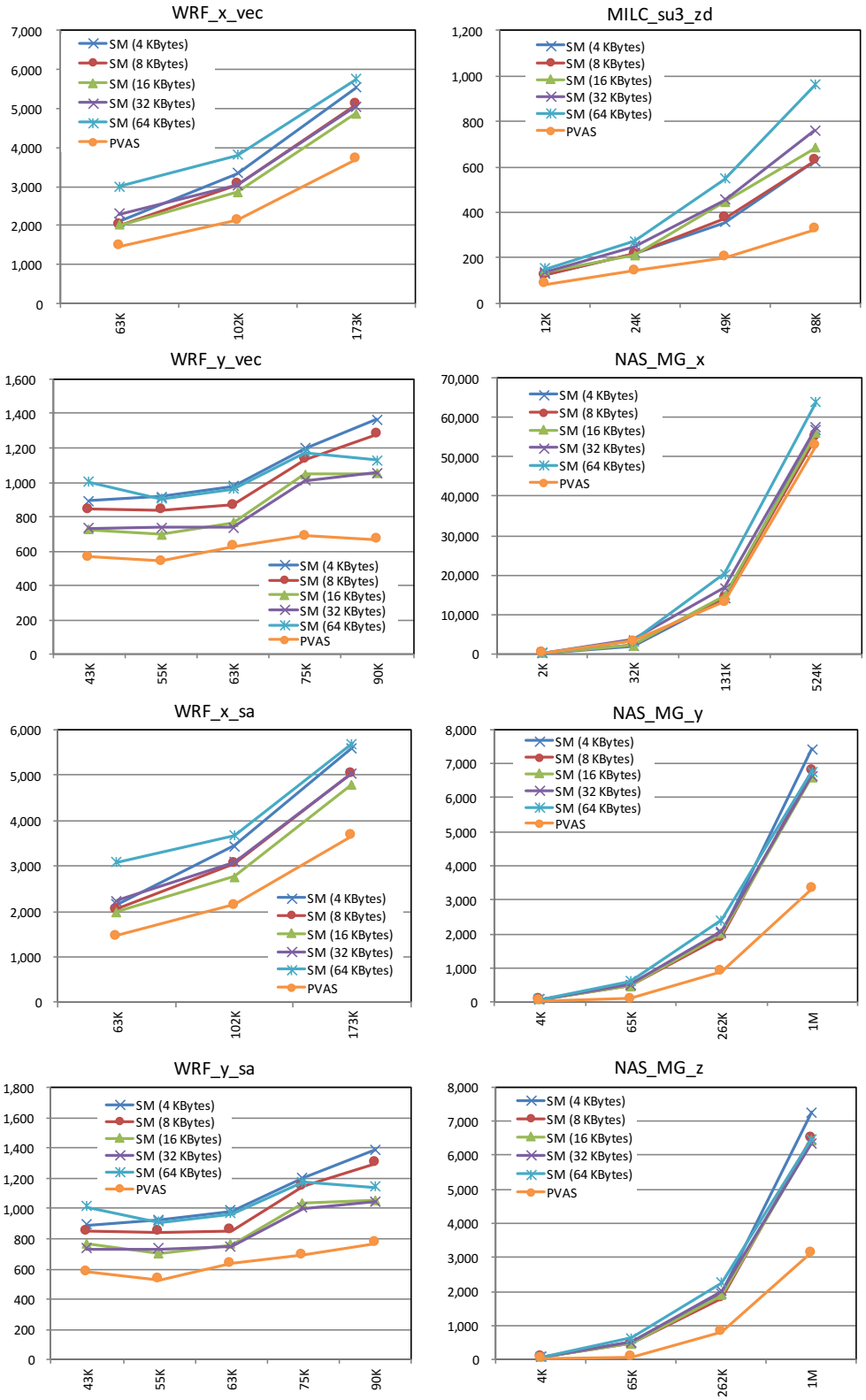


図 **6-11 DDTBench 1/2**（横軸：データサイズ **[Bytes]**，縦軸：レイテンシ**[us]**）

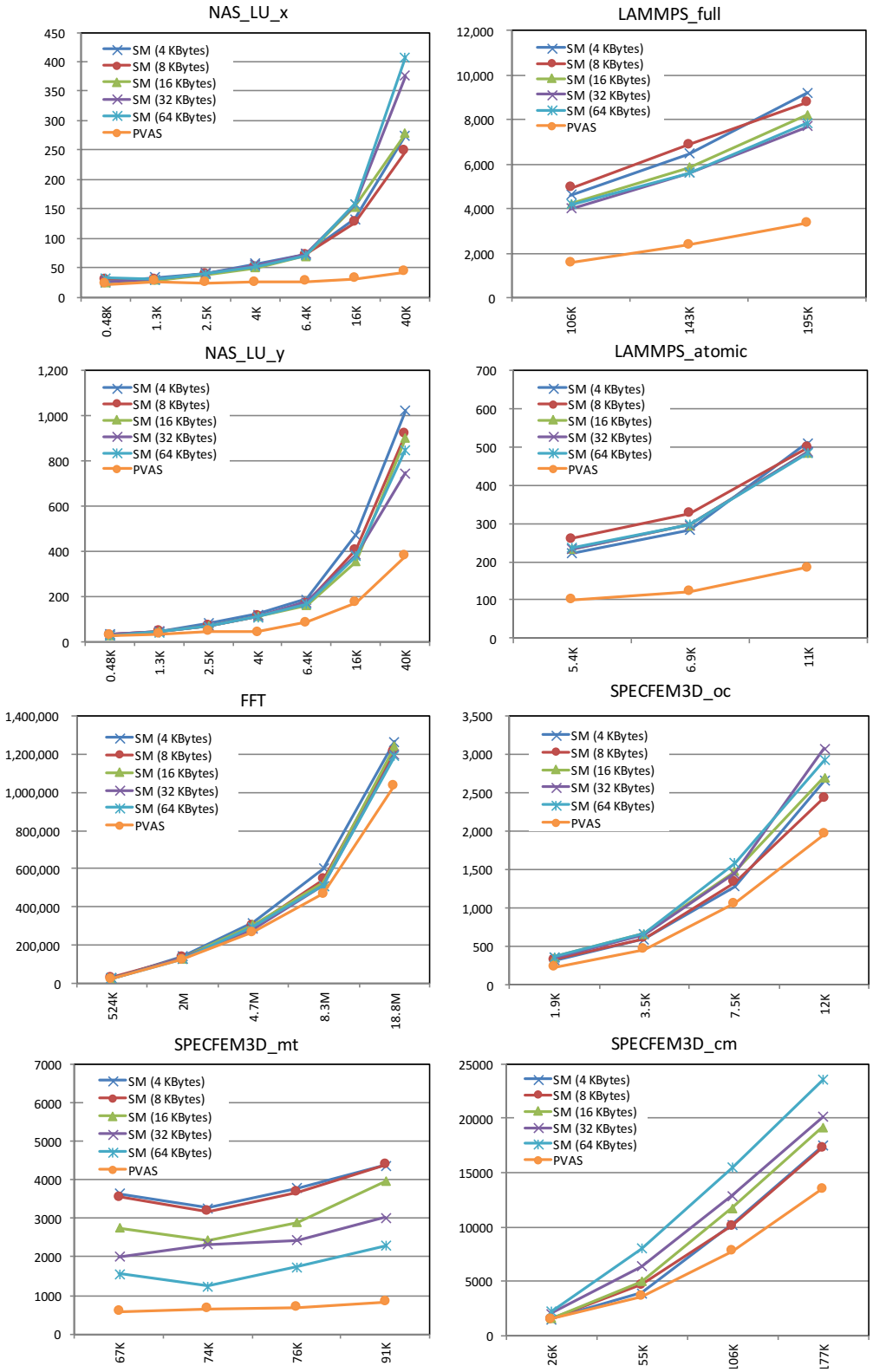


図 **6-12 DDTBench 2/2**（横軸：データサイズ **[Bytes]**，縦軸：レイテンシ**[us]**）

```
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 **6-13 NAS_MG_x** のデータ型

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同士で不連続なデータの送受信を行うと，図 6-15 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, データの転送に掛かる時間が大きくなる．共有メモリを用いる sm BTL の実装の場合，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通信と共有メモリを用いる通信の通信遅延の差が小さくなり，場合によっては，sm BTL の方が，通信遅延が小さくなってしまうと考えられる．

NAS_MG_x の測定において, 送受信するデータサイズが 2 KB 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通信の方が PVAS によっ

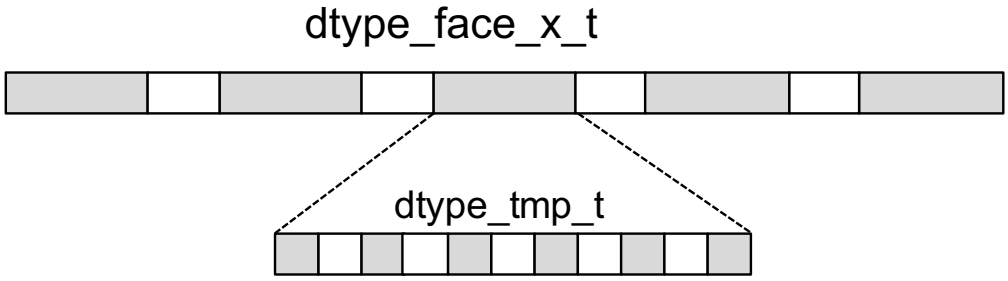


図 **6-14 NAS_MG_x** のデータレイアウト

[1] 本評価環境では，MPI_DOUBLE のサイズは 8 Bytes

て高速化した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 KB の場合，中間バッファのサイズが 4 KB から 16 KB 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通信の方が PVAS によって高速化した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 KB 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 で高速化した通信のほうが，通信遅延が小さくなる．送受信するデータサイズが 131 KB 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，そ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 KB 以上のときは，中間バッファがどのサイズでも，PVAS で高速化した通信のほうが，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 Rendezvous 通信の通信遅延の差が小さく，場合によっては共有メモリを用いる 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 通信と共有メモリを用いる 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，共有メモリを用いる 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 6-16 は，LAMMPS_full のソースコードにおいて，データ型の定義を行な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

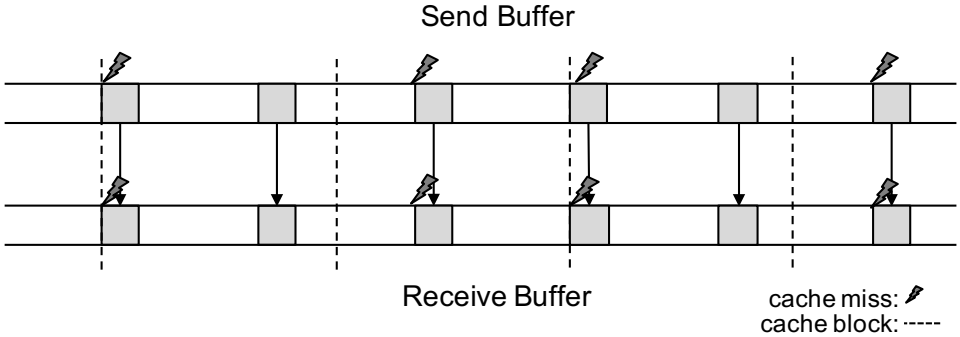


図 **6-15 NAS_MG_x** のデータコピー

```
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 6-16 LAMMPS_full のデータ型

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 6-17 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 × 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 × (3 × 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同士で通信を行う場合に比べて，データの転送を行なっている間に発生する CPU のキャッシュミスの頻

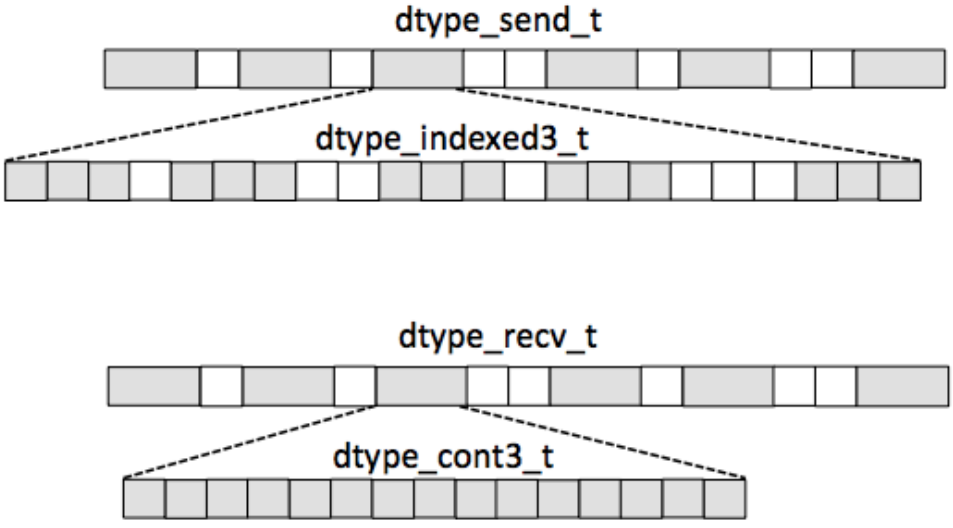


図 6-17 LAMMPS_full のデータレイアウト

度は低くなる．図 6-18 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，共有メモリ用いる Rendezvous 通信との通信遅延の差が大きくなると考えられる．

図 6-19 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[55]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 Rendezvous 通信の方が，通信遅延が小さかったデータサイズ（2 KB および 32 KB）の測定結果を示した．また，共有メモリを用いる 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 KB 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 KB にして測定した結果を示した．Xeon Phi は L1 および L2 キャッシュを持つが，Xeon Phi 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 Rendezvous 通信の方がキャッシュミスの回数が推測通り少なくなっている．対して LAMMPS_full の場合は共有メモリを用いる 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，却ってキャッシュミスの回数が増加してしまっている．本評価では測定を実施していないが，同様の理由

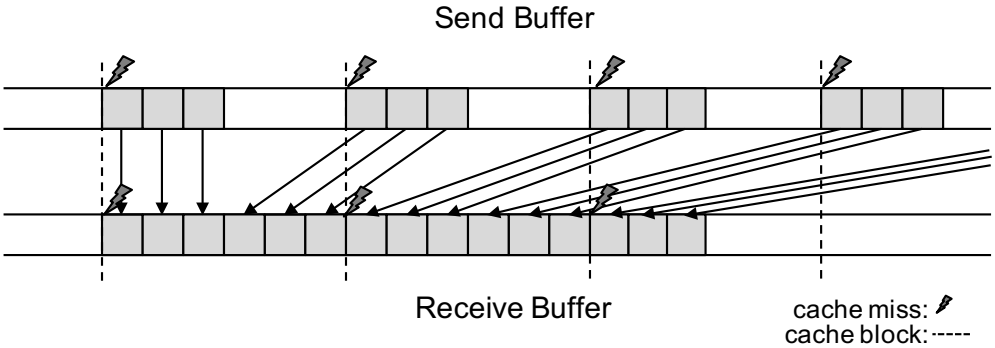


図 **6-18 LAMMPS_full** のデータコピー

で, TLB キャッシュのキャッシュミスの回数も, 共有メモリを用いる Rendezvous 通信の方が多くなると推測される.

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

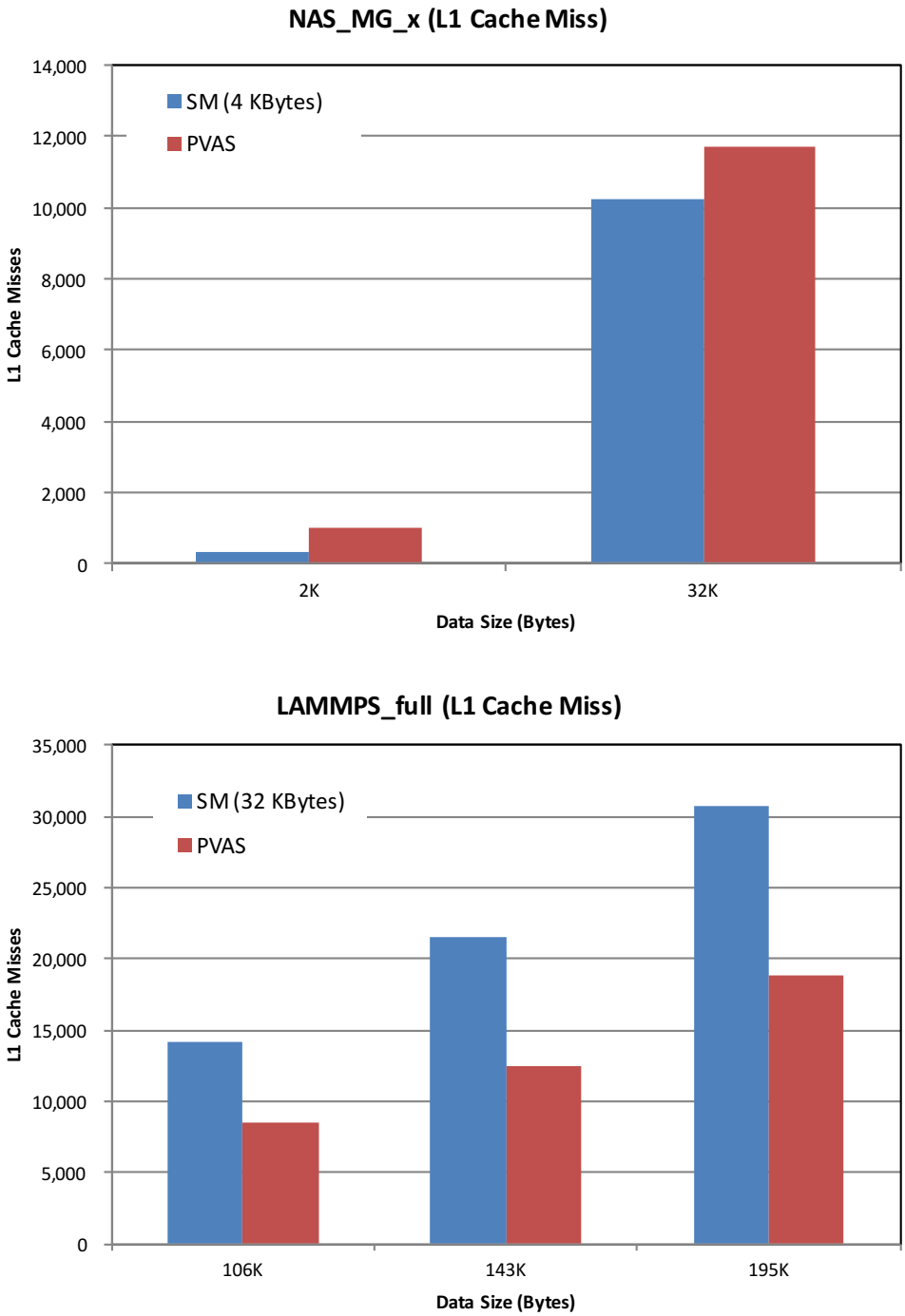


図 **6-19 L1** キャッシュミス

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, 却って通信遅延が増加してしまうケースがあることがわかった．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 BTL と sm BTL 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したい.

次に, PVAS BTL の Eager 通信と Rendezvous 通信の比較を行なった. DDTBench によって，Eager 通信の通信遅延を測定し，Rendezvous 通信の通信遅延と比較した．Eager 通信の測定の際は，常に Eager 通信が実行されるよう，eager limit を充分大きな値に設定して測定を行なった．結果を図 6-20 と図 6-21 に示す.

グラフに示す通り, 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 KB，NAS_lu_x でデータサイズが 0.48K ～ 1.3 KB，NAS_lu_y でデータサイズが 0.48 KB）を除き，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%低減した．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．中間バッファを経由するデータ転送のオーバヘッドは，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.

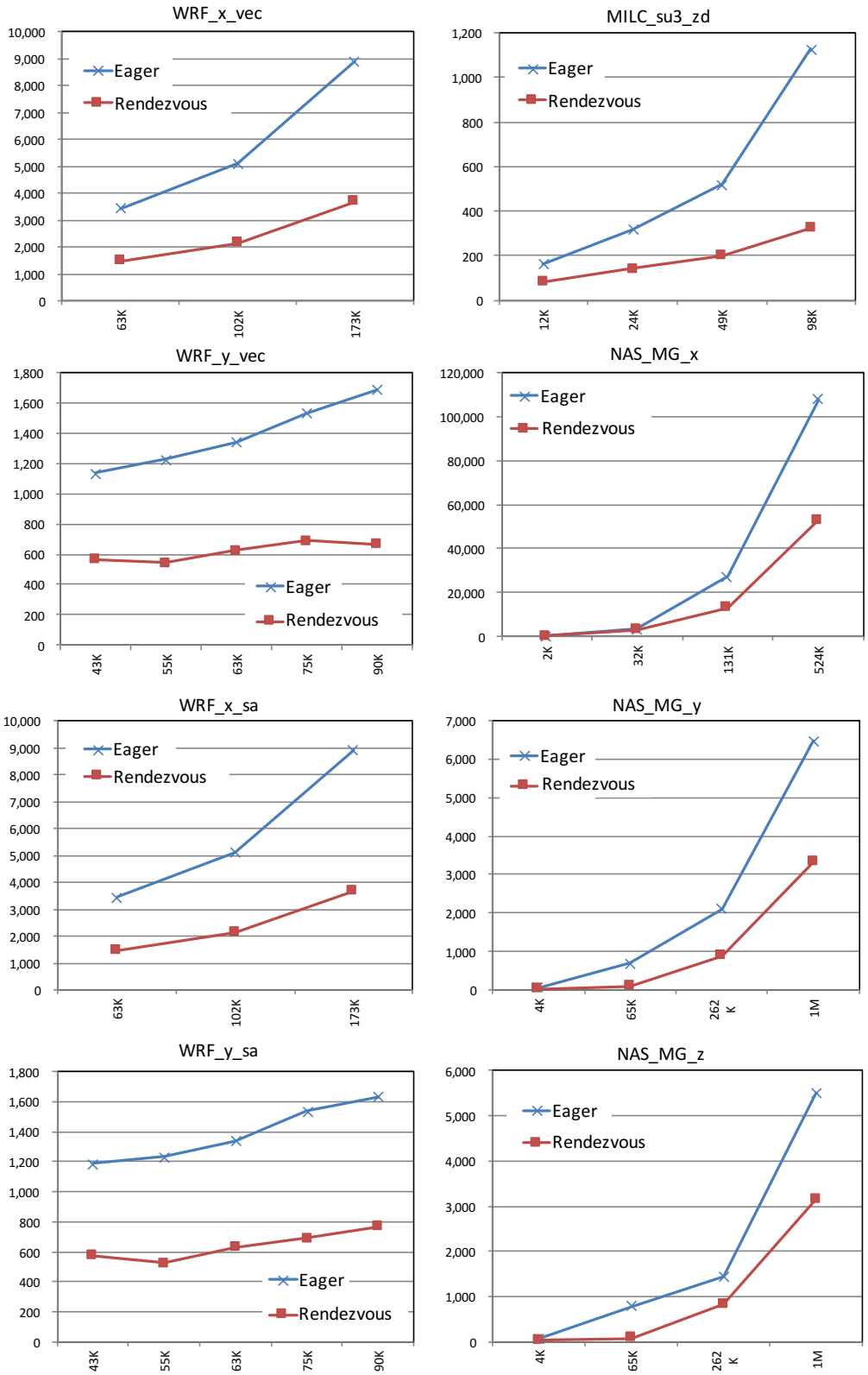


図 **6-20 Eager** 通信 **1/2**（横軸：データサイズ **[Bytes]**，縦軸：レイテンシ**[us]**）

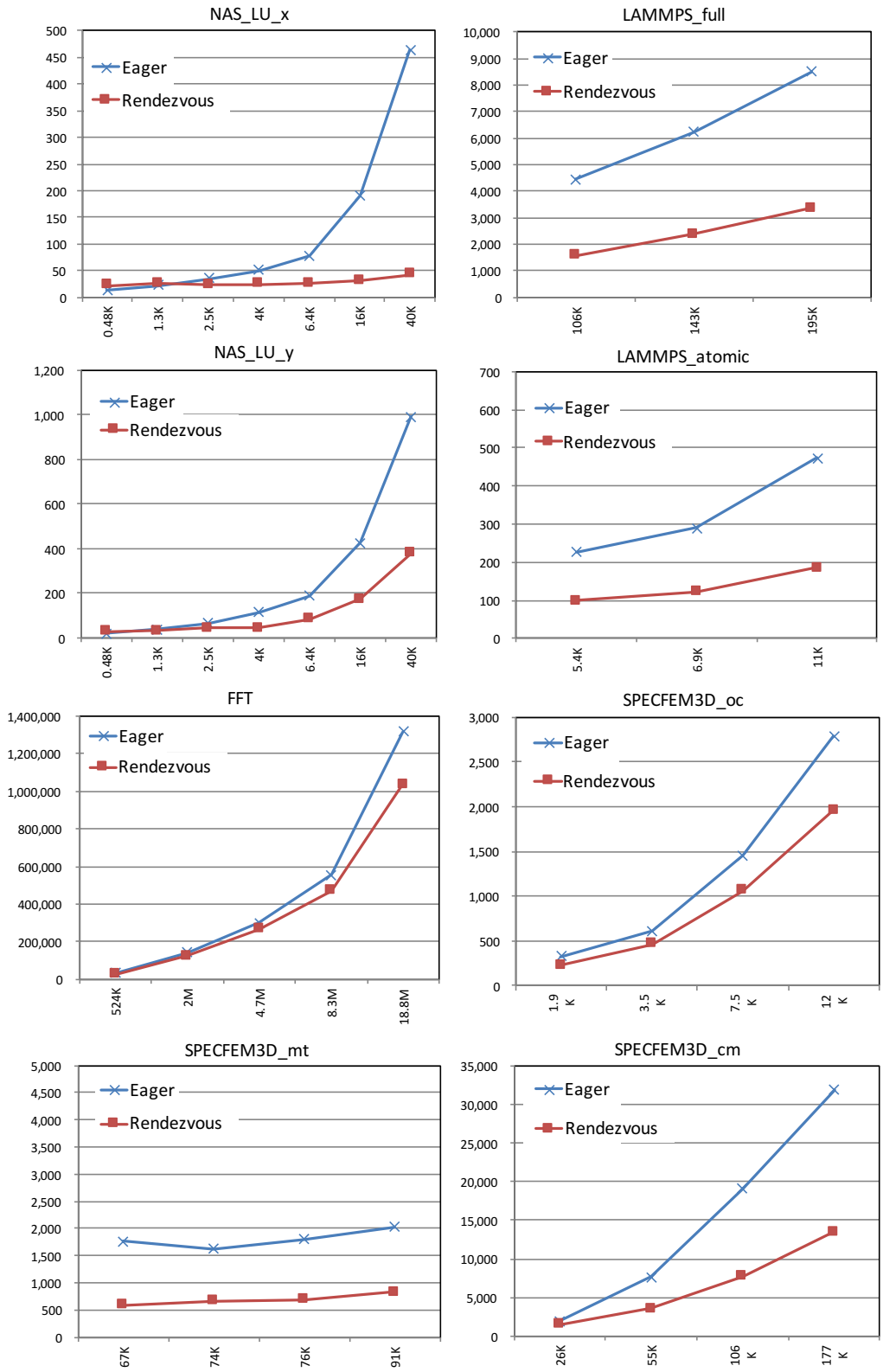


図 **6-21 Eager** 通信 **2/2**（横軸：データサイズ **[Bytes]**，縦軸：レイテンシ**[us]**）

実際，送受信するデータサイズが小さい場合（0.48 K〜2 KB）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．

sm BTL 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, ある程度大きいケースでは Eager 通信よりも有用であるといえる．Open MPI では eager limit のデフォルト値は 4 KB となっており，4 KB 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 KB を超えるケースでは，PVAS によって高速化した 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．

6.2.2 ミニアプリケーション（fft2d_datatype）

次に，fft2d_datatype[56]を用いて PVAS BTL の Rendezvous 通信と sm BTL の Rendezvous 通信を比較した．fft2d_datatype は，派生データ型を用いる MPI 通信を実行するアプリケーションで，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，送受信側共に MPI_Type_vector によって定義したデータ型を用いる．共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，4800 × 4800 と 9600 × 9600 の場合で測定を行なった．配列の要素数が 9600 × 9600 のときにメモリ消費量が約 8 GB となり，本評価で用いた Xeon Phi コプロセッサ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，常に Rendezvous 通信が実行されるように eager limit の値を最小値にして行なった. 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 KB として測定した．

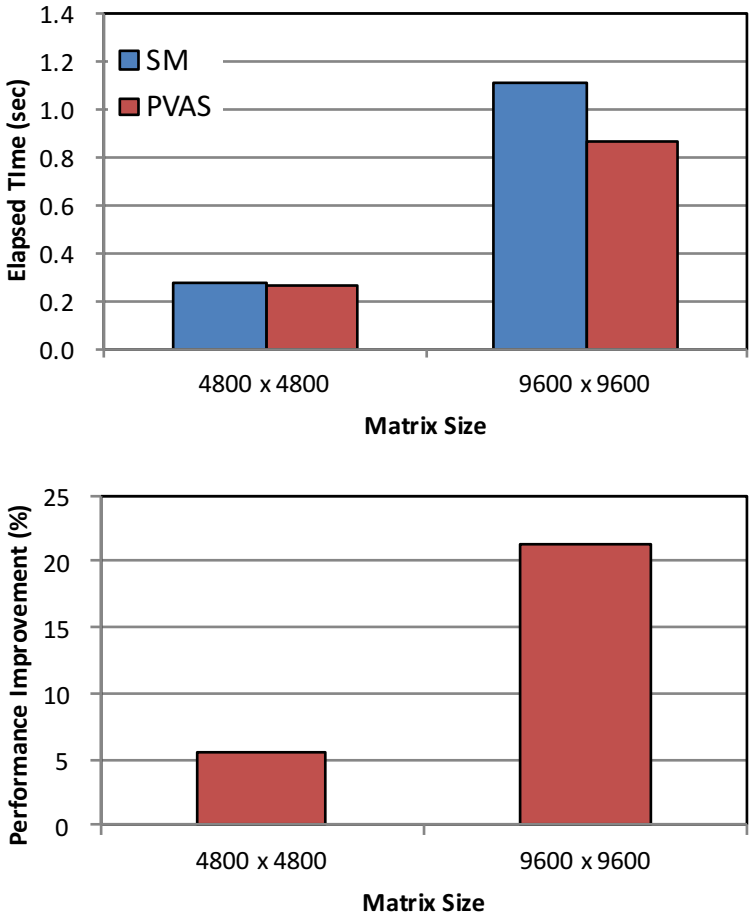


図 **6-22 fft2d_datatype**

fft2d_datatype の実行結果を図 6-22 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 9600 × 9600 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%改善した．配列サイズが 4800 × 4800 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掛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．

次に，PVAS BTL の Eager 通信と Rendezvous 通信の比較を行なった．ft2d_datatype では，プロセス数が 240 の場合，2 次元配列の要素数が 4800 × 4800 のときは 6.4 KB のデータを，要素数が 9600 × 9600 のときは 25.6 KB のデータを MPI プロセス間で送受信する．eager limit を充分大きな値に設定し，通信が全て Eager 通信で行なわれる設定で fft2d_datatype を実行したケースと，eager limit を最小値にし，通信が全て Rendezvous 通信で行なわれる設定で fft2d_datatype を実行したケースを比較した．結果を図 6-23 に示す．

上部のグラフにおいて，Eager は通信を全て Eager 通信で実行した場合の実行結果を，PVAS は通信を全て 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

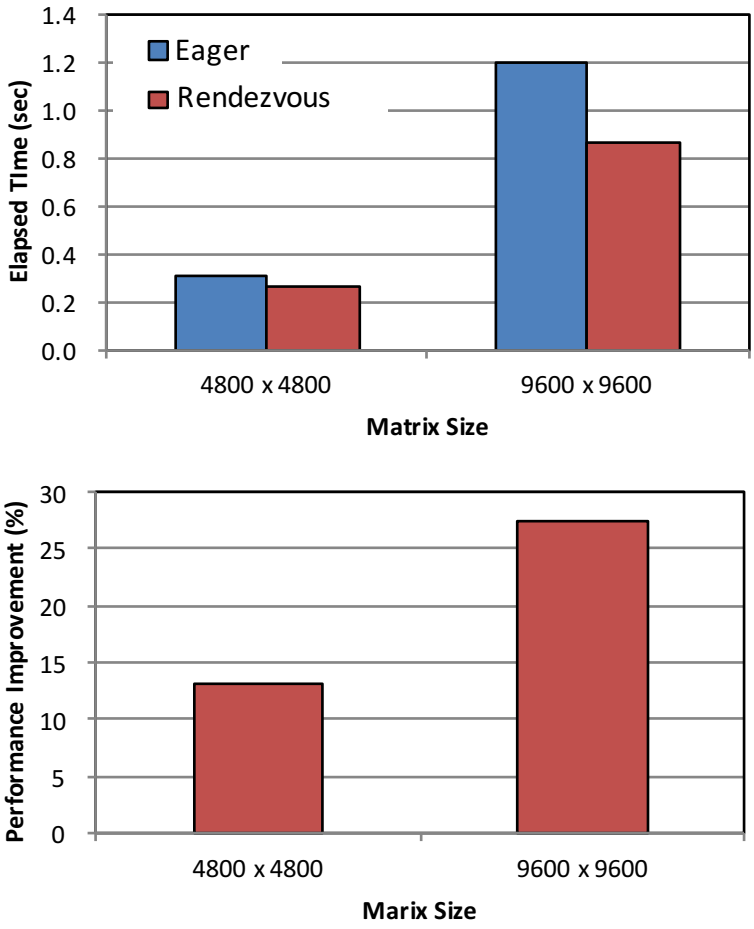


図 **6-23 fft2d_datatype（Eager vs. Rendezvous）**

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 4800 × 4800 のときは約 13%，配列サイズが 9600 × 9600 のときは約 27%, 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 による評価からもわかる．

6.3 メモリ消費量の評価

次に，MPI ノード内通信によるメモリ消費量の評価を行った．以下に評価について述べる．

6.3.1 マイクロベンチマーク（Intel MPI Benchmarks）

まず，IMB を用いて，sm BTL と PVAS BTL の MPI ノード内通信のメモリ消費量の比較を行なった．測定方法は以下の通りである．

まず，ベンチマークを実行する前のシステム全体のメモリ消費量を free コマンドで測定しておく．次に，ベンチマークを実行する全 MPI プロセスで同期をとり，全 MPI プロセスの処理が終了する直前に到達したところで，free コマンドによりシステム全体のメモリ消費量を測定する．そして，両測定値の差分をとり，ベンチマークと MPI ライブラリによるメモリ消費量を概算する．sm BTL を用いた場合のメモリ消費量と PVAS BTL を用いた場合のメモリ消費量を比較することで, 両 BTL における MPI ノード内通信のメモリ消費量の差を確認することができる．

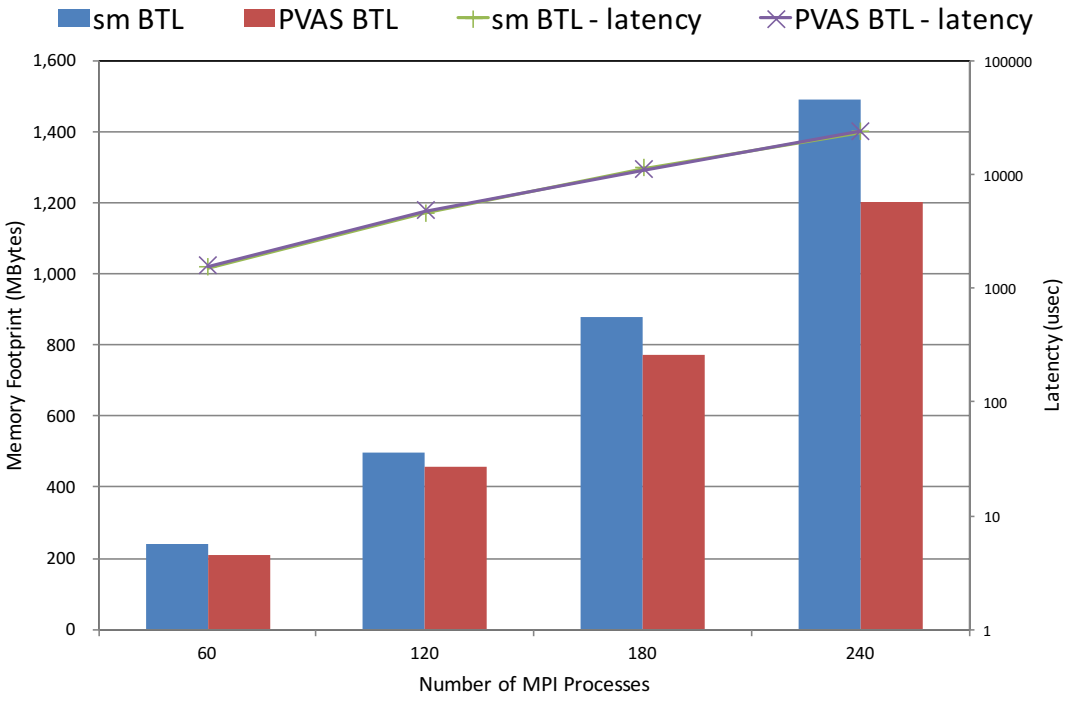
MPI_Alltoall による全対全の通信を行なうベンチマークで測定を行なった．測定は，メッセージの送受信を全て Eager 通信で行なう場合と，全て Rendezvous 通信で行なう場合の 2 通りで行なった．Eager 通信の測定では，eager limit を sm BTL のデフォルト値である 4 KB に設定し，メッセージサイズを 2 KB に固定することで，メッセージの送受信を全て Eager 通信で行なうようにした．Rendezvous 通信の測定では，メッセージサイズを eager limit のデフォルト値より大きい 16 KB に固定してベンチマークを実行した．sm BTL の Rendezvous 通信の測定では，共有メモリ上の中間バッファを経由してメッセージの送受信を行なう場合と，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行なう場合の双方で測定を行なった．共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行なう際のブロックサイズは，sm BTL のデフォルト値である 32 KB とした．ベンチマークでの通信の実行回数は 1000 とし

た．

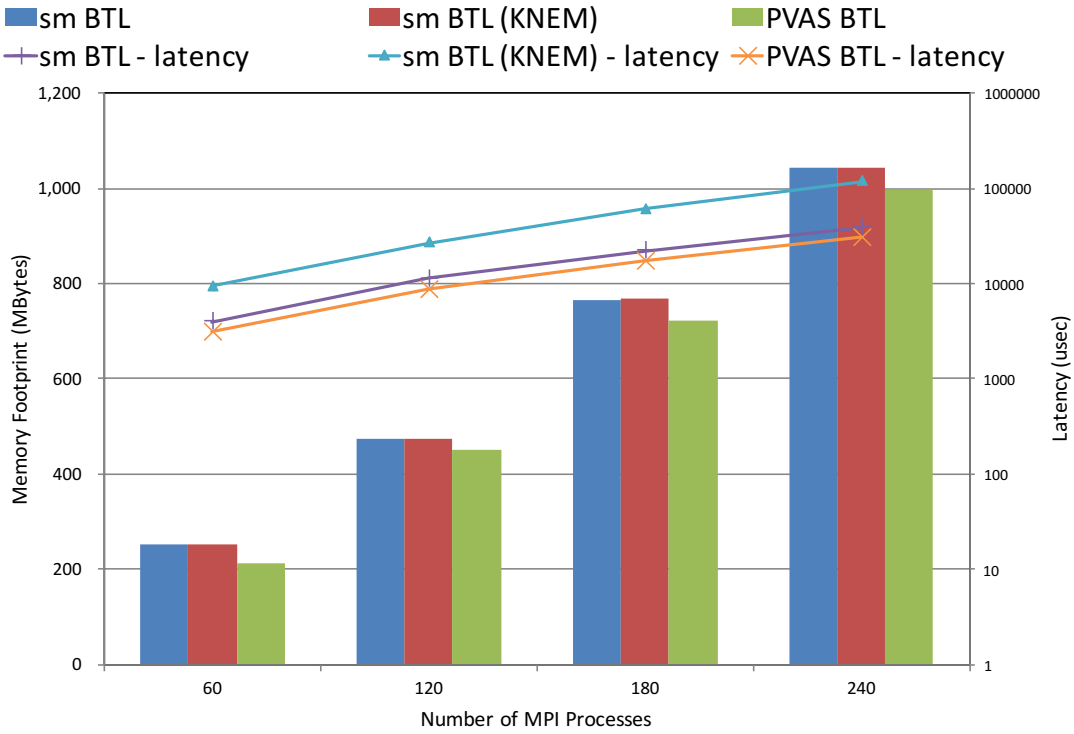
プロセス数が 60 から 240 のときにおける，ベンチマークと MPI ライブラリのメモリ消費量を図 6-24 に示す．また，グラフ中には，ベンチマークを実行したときの通信性能も示した．PVAS BTL と sm BTL を比較すると，PVAS BTL は sm BTL と同等以上の通信性能を維持しながら，メモリ消費量は sm BTL と比べて小さくなっている．240 プロセスで Eager 通信を行なう場合は約 264 MB，Rendezvous 通信を行なう場合は約 51MB，メモリ消費量が小さくなっている．Eager 通信時では，sm BTL を用いる場合と比べ，18%メモリ消費量を低減することができた．このメモリ消費量の差は，共有メモリのマッピングによるページテーブルサイズの増加に起因すると考えられる．

これを確認するため，MPI ノード内通信によるページテーブルサイズの増加量を比較した．ベンチマークを実行する前のシステム全体のページテーブルサイズと，ベンチマークを実行する全 MPI プロセスが終了する直前で停止したときのシステム全体のページテーブルサイズを，/proc/meminfo ファイルを参照して確認し，その差分を取ることで，ベンチマークの実行によって増加したシステム全体のページテーブルサイズを概算する．sm BTL を用いた場合のページテーブルサイズの増加量と PVAS BTL を用いた場合のページテーブルサイズの増加量を比較することで，MPI ノード内通信を実行したときのページテーブルサイズの増加量の差を確認することができる．

プロセス数が 60 から 240 のときにおいて，ベンチマークの実行により増加したページテーブルサイズを図 6-25 に示す. 240 プロセスで Eager 通信を行なう場合，PVAS BTL と sm BTL を比べると，ページテーブルサイズの増加量は，PVAS BTL の方が約 256 MB 少なくなっている．これは，図 6-24 のグラフで示したメモリ消費量の差とほぼ同等である. Rendezvous 通信のときも同様に，ページテーブルサイズの増加量の差が，図 6-24 のグラフで示したメモリ消費量の差とほぼ同等になっている．以上の結果から，PVAS BTL では，共有メモリのマッピングによるページテーブルサイズの増加が抑制され，MPI ノード内通信によるメモリ消費量が削減されたことがわかる．2.1.2 節で述べた通り，共有メモリによる MPI ノード内通信を用いて全対全の通信を行なう場合, プロセス数の 2 乗のオーダに従って，ページテーブルによるメモリ消費量が増加する．これは，図 6-25 のグラフ中に示した累乗近似曲線からも確認することができる．今後も 1 ノードあたりのコア数と MPI プロセス数は増加していくことが予想されるため，ページテーブルによるメモリ消費量の増加は，メニーコア環境では，より大きな問題になると考えられる．

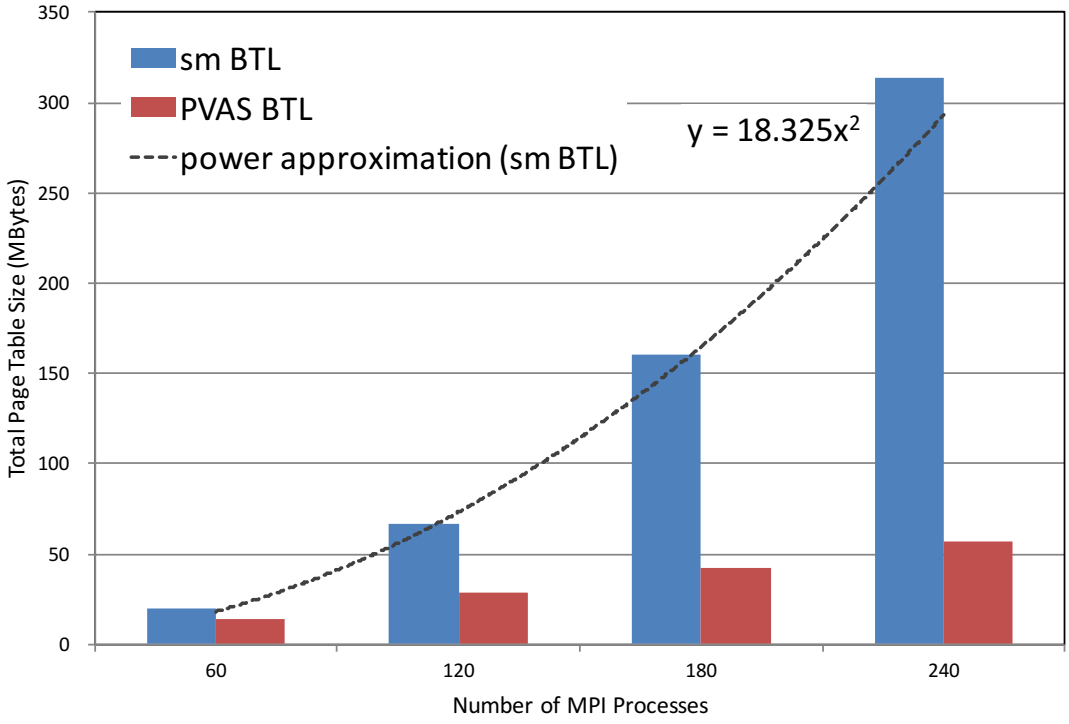


Eager通信

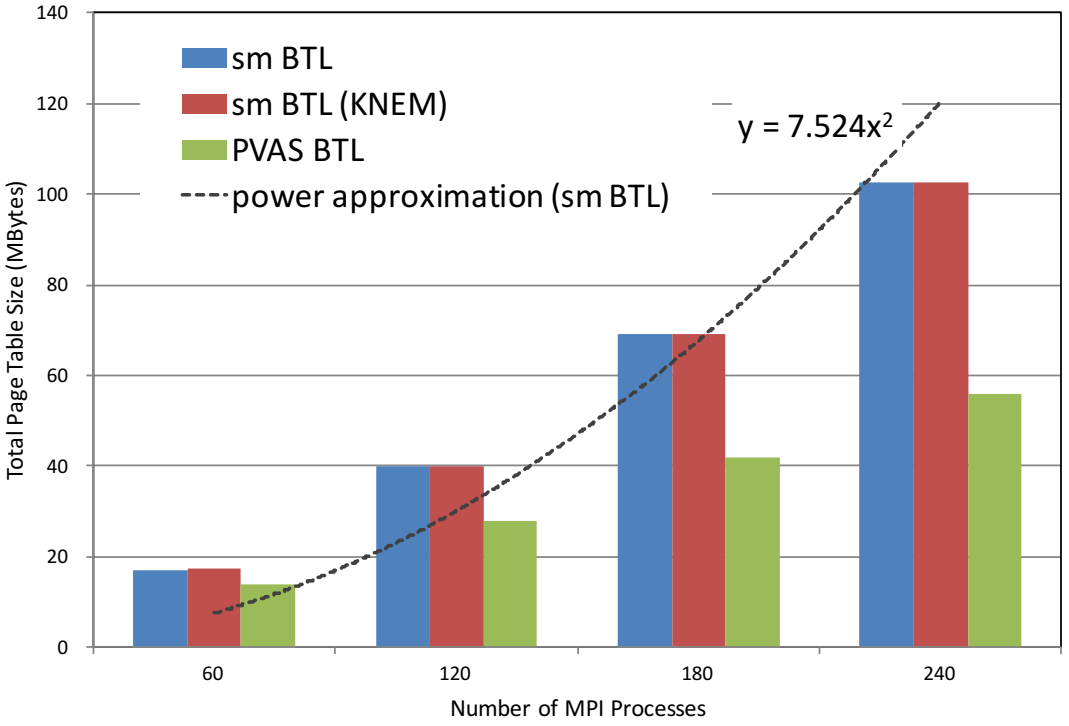


Rendezvous通信

図 **6-24** ノード内通信のメモリ消費量（**IMB**）



Eager通信



Rendezvous通信

図 **6-25** ページテーブルによるメモリ消費量（**IMB**）

集団通信を Eager 通信で行なう場合，1 度に多数の Eager 通信用バッファが使用されるため，Rendezvous 通信の場合よりもメモリ消費量が大きくなっている．また，sm BTL では，通信先の Eager 通信用バッファが格納されている共有メモリを各 MPI プロセスが, 自身のアドレス空間にマッピングするため, ページテーブルサイズの増加量も，Rendezvous 通信の場合と比べて大きくなっている．

sm BTL の Rendezvous 通信において，共有メモリ上の中間バッファを経由したパイプライン転送を行なう場合と KNEM を用いて OS カーネルの支援によるメッセージの送受信行なう場合では，メモリ消費量に大きな差はない．sm BTL は，共有メモリ上の中間バッファを経由したパイプライン転送を行うか否かに関わらず，MPI ライブラリの初期化時に，パイプライン転送用の中間バッファを作成してしまう．よって，この 2 つの方式で，メモリ消費量に大きな差が出ることは無いと考えられる．

6.3.2 アプリケーション（NAS Parallel Benchmarks）

次に，NPB を用いて MPI ノード内通信のメモリ消費量の比較を行なった．測定方法については，IMB によるメモリ消費量の測定方法と同じである．

ベンチマークには，MPI_Alltoall を実行する IS ベンチマークのクラス A を用いた．プロセス数は 2 の累乗である必要があるため，128 プロセスとした．測定は，eager limit を sm BTL のデフォルト値である 4 KB に設定した場合と，全通信を Rendezvous 通信で行なう場合の 2 通りで行なった．sm BTL の Rendezvous 通信において，共有メモリ上の中間バッファを用いてメッセージのパイプライン転送を行なう際のブロックサイズは, デフォルト値である 32 KB とした. eager limit を 4KB に設定したときのベンチマークのメモリ消費量（MPI ライブラリのメモリ消費量を含む）とベンチマークの実行によって増加したページテーブルサイズを表 6-4 に示す．また，全ての通信を Rendezvous 通信で行なった場合の IS のメモリ消費量（MPI ライブラリのメモリ消費量を含む）と, IS の実行によって増加したページテーブルサイズを表 6-5 に示す. 参考に, 6.1.2 節で測定した，ベンチマークの実行性能も併記した．なお，sm BTL の Rendezvous 通信において，KNEM を用いて OS カーネルの支援によるメッセージの送受信を行なう場合については，共有メモリ上の中間バッファを用いてパイプライン転送を行う場合と実行結果に有意な差が見られなかったため，記述を省略した．

eager limit を 4 KB に設定した場合，Eager 通信が行なわれ，Eager 通信用バッファが使用される．よって，全ての通信を Rendezvous 通信で行なう場合に比べて，メモリ消費量が大きくなっている．また，通信先の Eager 通信用バッファが

格納されている共有メモリを各 MPI プロセスが，自身のアドレス空間にマッピングするため，ページテーブルサイズの増加量も，Rendezvous 通信の場合と比べて大きくなっている．

sm BTL と PVAS BTL を比較すると，eager limit を 4 KB に設定したときのベンチマークのメモリ消費量の差は 30 MB である．このうち 20MB が，ページテーブルサイズの増加量の差である．残りの 10MB は，パイプライン転送用の中間バッファによるものと考えられる．PVAS BTL では，共有メモリのマッピングによるページテーブルサイズの増加が抑制され，MPI ノード内通信のメモリ消費量が削減されたことがわかる．既に述べた通り，共有メモリを用いた MPI ノード内通信では，プロセス数の 2 乗のオーダで，ページテーブルによるメモリ消費量が増加する．本測定では，IS の仕様により，128 プロセスでしかベンチマークを実行することができなかったが，より多数のプロセスで動作可能なアプリケーションを実行した場合は，ページテーブルサイズの増加量の差はより顕著になる．

IMB では，120 プロセスで Eager 通信の MPI_Alltoall を実行したときに，sm BTL と PVAS BTL のページテーブルサイズの増加量の差は約 38 MB であった．IS では，ほぼ同数のプロセスで Eager 通信の MPI_Alltoall を実行しているにも関わらず，ページテーブルサイズの増加量の差は 20 MB にとどまっている．これは各ベンチマークが実行する通信の回数の差に起因すると考えられる．各 MPI プロセスは，Eager 通信に用いる Eager 通信用バッファを free list で管理しており，同じ通信先に毎回同じ Eager 通信用バッファが使用されるとは限らない．よって，通信回数が増えると，通信先の MPI プロセスがアクセスする Eager 通信用バッファの数が増加する．sm BTL の場合は，通信先の MPI プロセスが自身のアドレス空間にマッピングしなければならない共有メモリが増え，ページテーブルサイズが増加してしまう．IMB のベンチマークでは，1000 回通信を実行していたが，IS ベンチマークでは，10 回しか通信を実行しないため，ページテーブルサイズの増加量が IMB のときと比べて少なくなっていると考えられる．

表 6-4 IS のメモリ消費量（eager limit = 4KB）

	sm BTL	PVAS BTL
IS のメモリ消費量（MB）	547	517
増加したページテーブルサイズ（MB）	51	31
実行性能（Mop/s total）	432.6	434.02

表 6-5 IS のメモリ消費量（Rendezvous 通信）

	sm BTL	PVAS BTL
IS のメモリ消費量（MB）	524	503
増加したページテーブルサイズ（MB）	41	31
実行性能（Mop/s total）	377.18	406.54

第7章　考察

7.1　スレッドによる並列化

本研究では，ノード間の並列化とノード内の並列化を，ともに MPI のようなプロセスレベルの並列化で行なうことを前提とし，PVAS タスクモデルを提案した．しかし，ノード間の並列化はプロセスレベルの並列化で行い，ノード内の並列化は OpenMP[32]や POSIX thread (Pthread) のようなスレッドレベルの並列化で行なうアプリケーションも存在する．ノード内の並列化をスレッドによって行なう場合は，ノード内通信は発生せず，ノード内通信を PVAS タスクモデルによって高速化する意義は失われる．

OpenMP で実装された NPB と MPI で実装された NPB の実行性能を比較した結果を図 7-1 に示す．OpenMP の実行性能を基準とし，相対的な実行性能の差をグラフ中に示した．評価環境は第 6 章と同様のものを用い，問題サイズはクラス A，B，C を用いた．ただし，FT ベンチマークについては，メモリ不足によりクラス C を実行することができなかった．MG，CG，FT，IS，LU ベンチマークにおいては，プロセスおよびスレッド数を 128 とし，SP，BT ベンチマークにおいては，プロセス数およびスレッド数を 225 とした．MPI については，eager limit を 4 KB に設定した．実行結果が示す通り，ノード内の並列化をプロセスレベルで行なうべきかスレッドレベルで行なうべきかは，実行する処理や問題サイズ等に依存する．先行研究においても，MPI による並列化とスレッドによる並列化を併用すると，MPI のみによる並列化よりも性能が低下してしまうケースがあることが報告されている[41-43]．プロセスレベルの並列化でノード内の並列化を行なうべきケースでは，本研究で提案した PVAS タスクモデルが効果を示す．

7.2　Huge page

メモリマッピングによるページテーブルサイズの肥大化を回避する方法として，Huge page を用いることが考えられる．Huge page を用いると，1 ページテーブルエントリで，より大きなサイズのメモリをマッピングすることができるため，ページテーブルサイズが小さくなる．Linux カーネルでは，Huge page を用いることが可能になっている[67]．PVAS タスクモデルと Huge page の双方を用

いてノード内通信を実装することで，ページテーブルサイズの増加によるメモリ消費量をより抑制することができる．

しかし，Huge page を用いるとメモリの管理単位が大きくなるため，却ってメモリの使用効率が低下してしまう可能性がある．また，Huge page を用いるとメモリページのマイグレーションが発生する．メモリページのマイグレーションは，システムの性能低下を招いてしまう懸念があるうえ，Remote-DMA(RDMA)[68]との相性が悪いという問題がある．MPI をはじめとする多くの通信ライブラリは infiniband の RDMA 機能を用いたノード間通信をサポートしている．RDMA により転送されるメモリは，メモリの仮想アドレスと物理アドレスのマッピングが変更されないように，mlock によってピンダウンしておく必要があるが，ピンダウンされるメモリに対しては，マイグレーションを実行することができなくなってしまう．

元来 Huge page は，巨大なメモリを効率よく管理するために用いられる機能であるため，コアあたりのメモリサイズが小さくなるメニーコア環境に適しているとはいえない．メニーコア環境で Huge page を用いる際は，前述した問題が発生しないよう，注意深く実装を行なう必要がある．

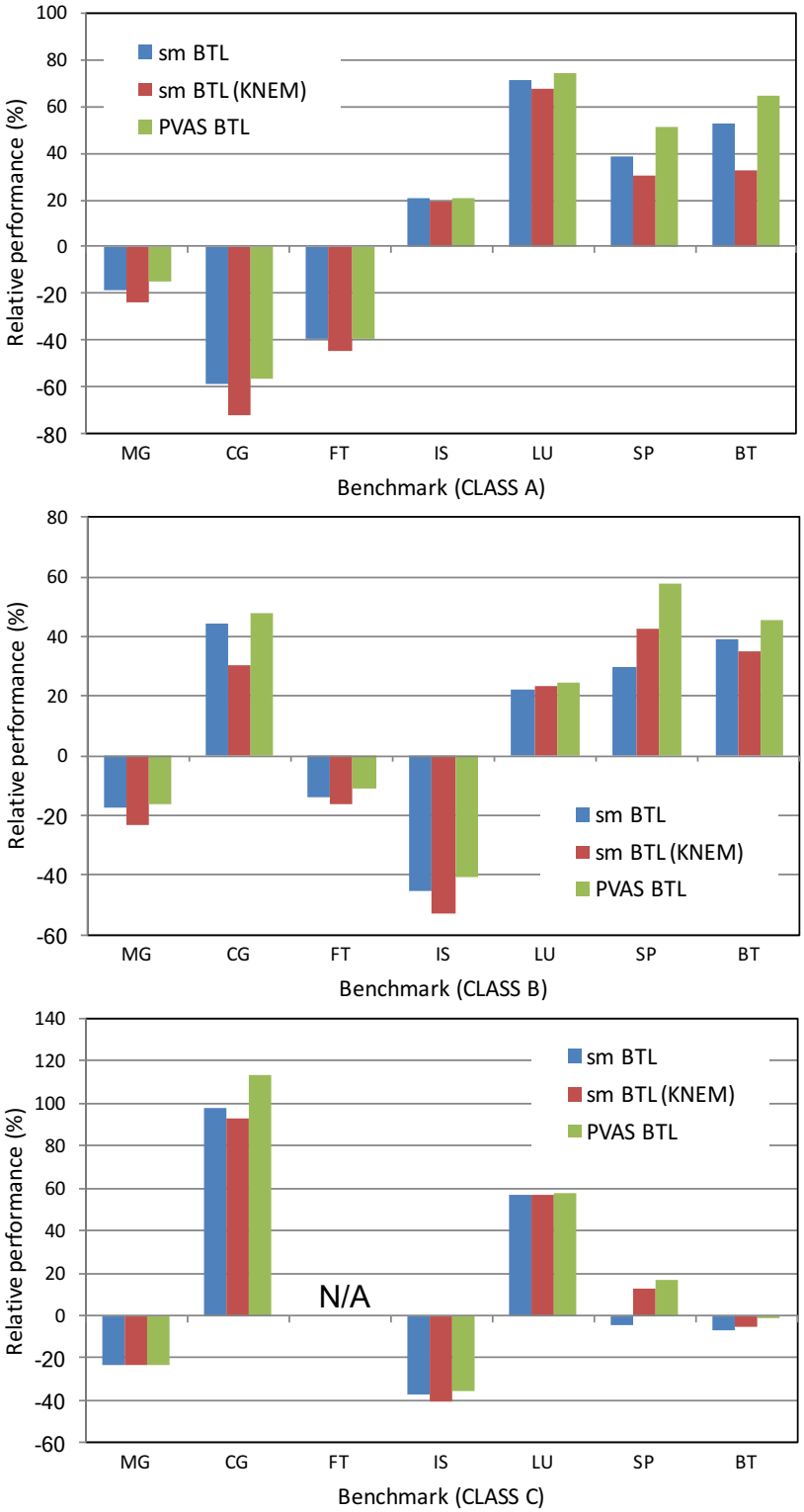


図 **7-1 OpenMP** との比較

第8章　結論

8.1　まとめ

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．一方で，1 コアあたりのメモリ量は減少する傾向にある．PVAS タスクモデルを用いると，このようなメニーコア環境で効率的に並列処理を実行することができる．本研究の提案する PVAS タスクモデルは，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を可能にする．同一アドレス空間で動作するプロセス同士が，アドレス空間をまたいでデータ転送する必要がなくなり，ノード内通信を実行する際に，通信遅延の増加やメモリ消費量の増加と問題が発生するのを，回避することができる．

本研究では，PVAS タスクモデルを MPI の通信に適用し，高速かつメモリ消費量の少ない，よりメニーコア環境に適した MPI ノード内通信を実現し，PVAS タスクモデルの検証を行なった．PVAS タスクモデルを用いた MPI ノード内通信を評価したところ，Rendezvous 通信を高速化し，連続データを送受信するミニアプリケーションの実行性能を最大で約 18%改善することができた．不連続データを送受信するミニアプリケーションについては，実行性能を約 21%改善することができた．また，ベンチマークソフトにより，MPI ノード内通信のメモリ消費量を測定したところ，共有メモリのマッピングによるページテーブルサイズの増加を抑制し，最大で約 18%，MPI ノード内通信によるメモリ消費量を低減することができた．

評価の結果から，ノード内通信が頻繁に発生するような並列アプリケーションでは，PVAS タスクモデルによる性能向上が期待できるといえる．例えば，隣接通信を行う並列アプリケーションにおいて，隣接する並列プロセス同士を同一ノード内で動作させればノード内通信が頻繁に発生し，PVAS タスクモデルによる性能向上が見込める．また，全体全の通信を実行する並列アプリケーションでは，共有メモリによるノード内通信を用いるとプロセス数の２乗のオーダでメモリマッピングが発生し，ページテーブルサイズが増加するが，PVAS タスクモデルによるノード内通信を用いると，ページテーブルサイズの増加を抑制し，ノード内通信に掛かるメモリ消費量の低減が見込める．

以上のように，本研究で提案する PVAS タスクモデルを用いることで，高速か

つ効率的な，メニーコア環境向けのノード内通信を実現可能であると示すことができた．多コア化が進む HPC システムにおいて，ノード内通信の高速化と効率化は大きな課題であり，本研究の成果の活用が期待できる．

8.2 今後の課題

更なるノード内通信の高速化，メモリ消費量の低減が今後の課題となる．加えて，PVAS タスクモデルを用いたノード内通信を他の並列化モデルに適用し，評価することも今後行なってく予定である．また，近年では，HPC システムが大量の電力を消費するようになっており，電力消費の削減を目指す研究も盛んになっている．PVAS タスクモデルを用いると CPU を利用するメモリコピーの回数を削減することが可能になり，電力削減の効果も見込める．今後はこういった観点での評価も行う．本研究では，ノード内通信に着目したが，PVAS タスクモデルを用いて，ノード間通信を高速化，効率化する研究にも着手したい．PVAS タスクモデルを用いれば，同一アドレス空間で動作する並列プロセスの通信を集約し，ノード間通信の発生回数を削減する，といった効率化も可能であると考える．このように，本研究で提案した PVAS タスクモデルの有用性の検証，応用を継続して行ない，研究を発展させていきたい．

謝辞

本論文をまとめるにあたり，多くの方からご協力とご指導を賜りました．お世話になった全ての方々に心より感謝致します．

本論文は筆者が慶應義塾大学大学院理工学研究科後期博士課程に在籍中の研究成果をまとめたものです．同研究科教授の河野 健二先生には，著者の指導教官として，研究の遂行および本論文の執筆において，厳しくも暖かいご指導をいただきました．また，研究者としての心構えをご教示いただきました．

慶應義塾大学大学院理工学研究科教授の寺岡 文男先生，山崎 信行先生，九州大学大学院システム情報科学研究院教授の井上 弘士先生からは，本論文の副査として数多くの助言をいただくと共に，本論文の細部に渡ってご指導をいただきました．

本論文は，著者が国立研究開発法人理化学研究所に出向して行った研究をもとにしています．理化学研究所システムソフトウェア研究チームチームリーダの石川 裕氏，同システムソフトウェア研究チーム上級研究員の堀 敦史氏からは，研究を行う機会と共に，研究に対する有益な助言を数多くいただきました．

東京農工大学教授の並木 美太郎先生には OS およびシステムソフトェアに関する知見をご教示いただくと共に，多くの助言をいただきました．

日立製作所における著者の上長であるストレージ研究部部長の志賀 賢太氏，同部ユニットリーダの須藤 敦之氏，早坂 光雄氏には，博士課程への入学を支援していただくと共に，本論文の執筆を暖かく見守っていただきました．また，同部主管研究員の藤本 和久氏には，著者のメンターとしてご指導いただきました．同部主任研究員の中村 隆喜氏，研究員の亀井 仁志氏からは，本論文の執筆に際し，有益な助言と励ましの言葉をいただきました．

著者の妻である島田 真央は，著者の博士課程への入学に理解を示し，著者の研究生活を支援してくれました．また，愛猫たちは，研究生活に癒しを与えてくれました．

参考文献

[1] TOP 500 Lists: TOP 500 Supercomputer Sites, http://www.top500.org.

[2] Dongarra, J., Choudhary, A., Kale, S. et al.: The International Exascale Software Project Roadmap, White paper, Argonne National Laboratory, 2010.

[3] Hybrid Memory Cube Consortium: Hybrid Memory Cube Specification 1.1, 2014.

[4] MPI: A Message-Passing Interface Standard Version 3.1, http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[5] Berkeley UPC - Unified Parallel C: The UPC Language, http://upc.lbl.gov.

[6] 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.

[7] Jinpil, L. and Mitsuhisa, S.: Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems, In *Proc. of the 39th international Conference on Parallel Processing Workshops*, pp. 413-420, 2010.

[8] Kemal, E., Vijay, S. and Vivek, S.: X10: an object-oriented approach to non-uniform cluster computing, In *Proc. of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 519-538, 2005.

[9] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: PGAS Intra-node Communication towards Many-Core Architecture, In *Proc. of the 6th Conference on Partitioned Global Address Space Programming Model*, 2012.

[10] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: Proposing A New Task Model towards Many-Core Architecture, In *Proc. of the ACM 1st international workshop on manycore embedded systems*, pp. 45-48, 2013.

[11] 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.

[12] 島田 明男, 須藤 敦之, 堀 敦史, 石川裕, 河野 健二: 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, 情報処理学会論文誌コンピューティングシステム (ACS), Vol. 9, No. 2, pp. 46–63, 2016.

[13] Wickramasinghe, U. S., Bronevetsky, G., Lumsdaine, A. and Friedley, A.: Hybrid MPI: A Case Study on the Xeon Phi Platform, In *Proc. of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ACM, Article No. 6, 2014.

[14] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org.

[15] 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, 2013.

[16] Jin, H.-W., Sur, S., Chai, L. and Panda, D. K.: LiMIC: Support for High-Performance MPI Intra-node Communication on Linux Cluster, In *Proc of the 2005 International Conference on Parallel Processing*, pp. 184–191 (2005).

[17] Vienne, J.: Benefits of Cross Memory Attach for MPI libraries on HPC Clusters, In *Proc. of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, Article No. 33.

[18] GCC, the GNU Compiler Collection, https://gcc.gnu.org.

[19] User and Reference Guide for the Intel C++ Compiler 15.0, https://software.intel.com/en-us/compiler_15.0_ug_c.

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

[21] MPICH: High-Performance Portable MPI, http://www.mpich.org.

[22] Huang, C., Lawlor, O. and Kal ´e, L. V.: Adaptive MPI, In *Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing*, pp. 306–322, 2003.

[23] Prache, M., Carribault, P. and Jourdren, H.: MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption, In *Proc. of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 94–103, 2009.

[24] J. C. D ´ıaz Mart ´ın, J. A. Rico Gallego, J. M. A ´lvarez Llorente, and J. F. Perogil Duque.: An mpi-1 compliant thread-based implementation. In *Proc of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 327–328, 2009.

[25] Woodacre, M., Robb, D., Roe, D. and Feind, K.: The SGI AltixTM 3000 Global Shared-Memory Architecture.

[26] Brightwell, R., Pedretti, K. and Hudson, T.: SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor, In *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, Article No. 25, 2008.

[27] Friedley, A., Bronevetsky, G., Hoefler, T. and Lumsdaine, A.: Hybrid MPI: Efficient Message Passing for Multi-core Systems, In *Proc. of the 2013 ACM/IEEE conference on Supercomputing*, Article No. 18, 2013.

[28] Mellanox: RDMA Aware Networks Programming User Manual, http://www.mellanox.com/related-docs/ prod software/RDMA Aware Programming user manual.pdf.

[29] 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, In *Proc. of the 2015 IEEE Cluster Conference*, 2015.

[30] Gillett, R. B.: Memory Channel Network for PCI, In *Proc. of IEEE Micro*, Vol. 16, No. 1, pp. 12–18, 1996.

[31] Kanoh, Y., Nakamura, M., Hirose, T., HOSOMI, T. and NAKATA, T.: User-level Network Interface for a Parallel Computer Cenju-4(Special Issue on Parallel Processing), *IPSJ Journal*, Vol. 41, No. 5, pp. 1379–1389, 2000.

[32] OpenMP: The OpenMP API specification for parallel programming, http://openmp.org.

[33] T. Wilkinson, K. Murray, S. Russel, G. Heiser, and J. Liedtke. Single Address Space Operating Systems. Technical Report UNSW-CSE-TR-9504, School of Computer Science and Engineering, 1995.

[34] Chase, J., Levy, H., Baker-Harvey, M. and Lazowska, E.: Opal: A Single Address Space System for 64-Bit Architectures, In *Proc. IEEE Workshop on Workstation Operating Systems*, pp. 80-85, 1992.

[35] Gernot, H., Kevin, E., Stephen, R. and Jerry, V.: Mungi: A distributed single address-space operating system, Technical Report UNSW-CSE-TR-9314, School of Computer Science and Engineering, 1993.

[36] Tang, H. and Yang, T.: Optimizing Threaded MPI Execution on SMP Clusters, In *Proc. of the 15th International Conference on Supercomputing*, ACM, pp. 381–392, 2001.

[37] E. Demaine.: A threads-only mpi implementation for the development of parallel programs. In *Proc. of the 11th International Symposium on High Performance Computing Systems*, pp. 153–163, 1997.

[38] Hydra Process Management Framework, https://wiki.mpich.org/mpich/index.php/Hydra Process Management Framework.

[39] MVAPICH: MPI over Infiniband, Omni-Path, Ethernet/iWARP, and RoCE, http://mvapich.cse.ohio-state.edu.

[40] P, Mathias.: Too much PIE is bad for performance, Technical report / ETH Zurich, Department of Computer Science, 2012.

[41] F. Cappello and D. Etiemble.: MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks, In *Proc. of the ACM/IEEE International Conference on High-Performance Computing and Networking*, Aricle No. 12, 2000.

[42] D. S. Henty.: Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling, In *Proc. of the ACM/IEEE International Conference on High-Performance Computing and Networkin*g, Article No. 10, 2000.

[43] S. Dong and G. E. Karniadakis.: Dual-level parallelism for deterministic and stochastic CFD problems, In *Proc. of the ACM/IEEE International Conference on High-Performance Computing and Networking*, pp. 1-17, 2002.

[44] Sandia National Laboratory: Kitten Lightwaight Kernel, https://software.sandia.gov/trac/kitten.

[45] Sandia National Laboratory: Open Catamount, http://www.cs.sandia.gov/rbbrigh/OpenCatamount.

[46] Intel Corporation: Intel MPI Benchmarks 4.0, http://software.intel.com/en-us/articles/intel-mpi- benchmarks.

[47] Davide H.Bailey, e. a.: The NAS Parallel Benchmarks, *International Journal of Supercomputer Applications*, Vol. 5, No. 3, pp. 63-73, 1991.

[48] Schneider, T., Gerstenberger, R. and Hoefler, T.: Micro-Applications for Communication Data Access Patterns and MPI Datatypes, In *Proc. of the 19th European Conference on Recent Advances in the Message Passing Interface*, pp. 121– 131, 2012.

[49] Skamarock, W. C. and Klemp, J. B.: A Time- split Nonhydrostatic Atmospheric Model for Weather Research and Forecasting Applications, *J. Comput. Phys.*, Vol. 227, No. 7, pp. 3465–3485, 2008.

[50] Komatitsch, D. and Tromp, J.: Modeling of Seismic Wave Propagation at the Scale of the Earth on a Large Beowulf, In *Proc. of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 33-33, 2001.

[51] Bernard, C., Ogilvie, M. C., Degrand, T. A., De- tar, 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, 1991.

[52] Plimpton, S.: Fast Parallel Algorithms for Short-range Molecular Dynamics, *J. Comput. Phys.*, Vol. 117, No. 1, pp. 1–19. 1995.

[53] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R. and Tra ¨ff, J. L.: MPI on a Million Processors, In *Proc. of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, pp. 20–30, 2009.

[54] Goodell, D., Gropp, W., Zhao, X. and Thakur, R.: Scalable Memory Use in MPI: A Case Study with MPICH2, In *Proc. of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, pp. 140–149, 2011.

[55] The University of Tennessee Innovative Computing Laboratory: Performance Application Programming Interface, http://icl.cs.utk.edu/papi/.

[56] Scalable Parallel Computing Laboratory: MPI Derived Datatype (Benchmark) Page, http://spcl.inf.ethz.ch/ Research/Parallel Programming/MPI Datatypes/.

[57] The Free BSD Project, https://www.freebsd.org.

[58] Intel Corporation: Intel Itanium Processor, http://www.intel.com/content/www/us/en/products/processors/itanium.html.

[59] SPARC, http://sparc.org.

[60] IBM: PowerPC User Instruction Set Architecture, http://public.dhe.ibm.com/software/dw/library/es-ppcbook1.zip.

[61] ARM: ARM Processor Architecture, https://www.arm.com/ja/products/processors/instruction-set-architectures/index.php.

[62] Roscoe, T.: Linkage in the Nemesis single address space operating system, *ACM SIGOPS Operating Systems Review*, Volume 28, Issue 4, pp. 48-55, 1994.

[63] IBM: IBM i, https://www-03.ibm.com/systems/power/software/i.

[64] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska.: Sharing and protection in a single address space operating system, *ACM Transactions on Computer Systems*, Vol. 13, Issue 4, pp. 271–307, 1994.

[65] JerryVochteloo, Stephen Russell, and Gernot Heiser.: Capability-based protection in the Mungi operating system. In *Proc. of the 3rd International Workshop on Object Orientation in Operating Systems*, pp. 108–115, IEEE, 1993.

[66] Kevin Murray, Ashley Saulsbury, Tom Stiemerling, Tim Wilkinson, Paul Kelly, and Peter Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proc. of the 2nd USENIX Symposium on Microkernels and other Kernel Architectures*, pp. 31–43, 1993.

[67] The Linux Kernel Archive: Huge pages, https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[68] Mellanox Interconnect Community: What is RDMA?, https://community.mellanox.com/docs/DOC-1963.

[69] Hoefler, T., Dinan, J., Buntinas, D., Balaze, P., Barret, B., Brightwell, R., Gropp, W., Kale, V., Thankur, R: MPI+MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory, Published in *Journal Computing*, Vol. 95, Issue 12, pp. 1121-1136, 2013.

[70] Zhu, X., Zhang, J., Yoshii, K., Li, S., Zhang, U., Balaji, P.: Analyzing MPI-3.0 process-level shared memory: A case study with stencil computations, In *Proc. of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.