密結合並列演算加速機構TCAを用いたGPU間直接通信によるCollective通信の実装と性能評価

松本 和也[1,a)]　塙 敏博[2]　児玉 祐悦[1,3,†1]　藤井 久史[3,†2]　朴 泰祐[1,3]

概要：筑波大学計算科学研究センターでは，GPU クラスタにおけるノード間に跨る GPU 間通信のレイテンシ改善を目的とした密結合並列演算加速機構 TCA (Tightly Coupled Accelerators) を独自開発している．本稿では，Scatter，Reduce，Allgather，Allreduce の 4 つの Collective 通信の TCA による実装と，その性能を TCA 実証環境の GPU クラスタである HA-PACS/TCA において評価した結果を述べる．TCA による実装は通信レイテンシが問題となる小さめなサイズの Collective 通信において，MPI による Collective 通信と比べて高速にその通信処理を行うことが可能であることを示す．また，実装した Collective 通信を利用した Conjugate Gradient 法（CG 法）の実装およびその性能について述べる．本研究で用いる CG 法の並列アルゴリズムは，Allgather と Allreduce をその通信部分に用いるものである．TCA による Collective 通信を用いた CG 法実装は，疎行列のサイズ（行数）が数千から数万の場合では MPI の Collective 通信を用いた実装よりも高い性能を達成できることを示す．

Implementation and Performance Evaluation of Collective Communication with Proprietary Interconnect TCA for GPU Direct Communication

Kazuya Matsumoto[1,a)]　Toshihiro Hanawa[2]　Yuetsu Kodama[1,3,†1]　Hisafumi Fujii[3,†2]
Taisuke Boku[1,3]

Abstract: We have been developing a proprietary interconnect technology called Tightly Coupled Accelerators (TCA) architecture to improve communication latency and bandwidth between compute nodes on a GPU cluster. This paper presents the implementation and performance evaluation results of four different collective communication operations (scatter, reduce, allgather, allreduce). The performance measurements are conducted on HA-PACS/TCA, which is a proof-of-concept GPU cluster based on the TCA architecture. The implementation using TCA is faster than an MPI collective communication implementation in case collective communications for small sizes where the communication latency decides most of its performance. This paper also describes an implementation of Conjugate Gradient (CG) method utilizing the implemented collective communication and the performance. We use the parallel algorithm of CG method that utilizes the allgather and allreduce in the data communication. The CG method implementation using TCA outperforms the implementation using MPI for sparse matrices whose matrix size is thousands to tens of thousands.

1. はじめに

近年，高い演算性能とメモリバンド幅性能を持つ GPU を汎用的な計算に活用する GPGPU（General Purpose GPU）が盛んに行われている．GPU は消費電力あたりの演算性

1　筑波大学計算科学研究センター
2　東京大学情報基盤センター
3　筑波大学大学院システム情報工学研究科
†1　現在，理化学研究所計算科学研究機構
†2　現在，富士通ソフトウェアテクノロジーズ
a)　kzmtmt@ccs.tsukuba.ac.jp

能という点においても CPU を上回り，GPU を計算加速機構として搭載した GPU クラスタも増加する一方である．しかし，GPU クラスタを高性能並列処理に利用するためには，複数ノードにまたがる GPU 間データ通信が必要であり，その通信速度は GPU の演算性能と比べて充分に速いとは云えない．この問題は，多くの GPU クラスタにおける高性能アプリケーション開発の障壁となることが少なくない．

そこで筑波大学計算科学研究センターでは，ノードを跨ぐ通信に関わるレイテンシとバンド幅の改善を目指して密結合型並列演算加速機構 TCA (Tightly Coupled Accelerators) を独自開発している [1], [2]．TCA はインターコネクト・ネットワークに関する技術であり，ノードを跨ぐ GPU などのアクセラレータ間の直接通信を可能にする．2013 年 10 月からは，GPU クラスタ HA-PACS[3] の拡張部として，TCA 実証環境である HA-PACS/TCA が稼働している．

TCA については Ping-pong 性能の評価 [2] を始めとして、いくつかの性能評価が行われている [4], [5], [6]．実際のアプリケーションに TCA を利用した場合の有効性については，姫野ベンチマーク [6]，格子 QCD のライブラリ QUDA[4]，Conjugate Gradient 法（CG 法）に対してこれまでに行なってきた．これらの性能評価により，TCA はノードを跨ぐ GPU 間通信を低レイテンシで実現できることが確かめられ，通信時間が計算時間よりボトルネックとなる強スケーリングが求められるような場合の一部においては TCA は有効であることが示されている．

Collective 通信（集団通信）とは，複数ノード/プロセスが同時に関わる通信操作のことである．MPI 標準仕様においても各種の Collective 通信が定義されており，MPI を用いて並列計算を行うほとんどのプログラムに Collective 通信は現れる．Collective 通信についての研究は多様で，ネットワークトポロジを考慮した通信アルゴリズムの研究 [7] や通信ライブラリの自動チューニングの研究 [8] なども行われてきた．

本研究では，TCA を用いて各種の Collective 通信を実装しその性能を評価する．TCA による通信は Remote Write による片方向通信を基本としており，1 対 1 通信が基本の MPI により記述された並列プログラムを，単純に TCA を用いる形に移植できるわけではない．本稿においては，4 つの Collective 通信（Scatter, Reduce, Allgather, Allreduce）の TCA を用いた実装について述べる [*1]．そして，TCA を用いた実装と MPI の Collective 通信実装との性能比較を行い，TCA がどのような条件で有効であるかを示す．

また，本稿では Collective 通信実装を CG 法の実装へ利用した結果についても記す．本研究で用いる CG 法の並列アルゴリズムは，Allgather と Allreduce をその通信部分に用いるものである．CG 法の GPU クラスタへの実装に関する研究はいくつかある [10], [11] が，それらは行列サイズ（行数）が数十万以上と大規模な疎行列に対する研究である．本研究で特に注目する行列の行数は数千から数万であり，通常の GPU クラスタでは並列処理性能が十分に引き出せないような高いストロング・スケーラビリティが要求される小規模な疎行列に関する評価を行うという点も本研究の寄与の一つである．

2. 密結合並列演算加速機構 TCA

2.1 TCA と PEACH2

密結合並列演算加速機構 TCA（Tightly Coupled Accelerators) は，アクセラレータ間（演算加速機構間）の直接結合を実現する通信機構技術のことであり，その詳細は文献 [1], [2], [12] に詳しい．本節では，本稿を理解するための TCA の概要を述べる．TCA の基本的なハードウェア技術は，PCI-Express (PCIe) を応用したものである．PCIe は，シリアルバス・インタフェースであり GPU ボード，Eathernet ボード，InfiniBand ボードなどの外部機器をホストコンピュータに結合するために広く使われている．

PEACH2 (PCI Express Adaptive Communication Hub version 2) は，TCA 技術を実装した PCIe インタフェース・チップである [2]．PEACH2 同士を PCIe 外部ケーブルにより結合することにより，PC クラスタシステムを構築することができる．

PEACH2 はノードを跨ぐ GPU 間の直接データ通信を GPUDirect Support for RDMA (GDR) 技術 [13] を利用することで実現する．現在，GDR 技術は NVIDIA の Kepler アーキテクチャの GPU ファミリにおいて利用できる．GDR を用いることで，PEACH2 や InfiniBand HCA のような通信アダプタは，GPU メモリへの直接読み書きが可能となる．GDR は不必要なシステムメモリへのデータコピーを削減し，CPU のオーバーヘッドを小さくし，通信レイテンシを短くする．InfiniBand HCA も GDR による通信をサポートするが，PEACH2 は InfiniBand に比べて更にレイテンシが小さいという利点がある．それは，PEACH2 を介した通信は PCIe のプロトコルのままで行われるので，InfiniBand を介した通信では必要な PCIe と InfiniBand 間とのプロトコル変換に伴うオーバヘッドを削減できるためである．

PEACH2 ボードは，最大 4 GB/s の帯域幅でデータ通信を行う PCIe Gen2 x8 ポートを 4 つ持つ．1 ポートはホストとの接続に用い，残りの 3 ポートは他のノードの PEACH2 と接続するために用いられる．

PEACH2 は，PIO と DMA の 2 つの通信方式を備えている [2]．PIO 通信は，CPU の store 操作によりリモートノー

[*1] これら以外にも Broadcast 通信と Gather 通信が実装済みである [9]．しかし，それらの通信特性については Allgather または Scatter に類似しているため，本稿ではその結果については言及しない．

ドへデータ書き込みを行う．通信レイテンシが小さいため，小さなデータの通信に向いている．なお，PEACH2 は CPU 間の PIO 通信のみを提供する．それに対して DMA 通信機能は，PEACH2 チップに 4 チャネル搭載されている DMA コントローラにより実現される．DMA 通信は，データの読み込み元と書き込み先の PCIe アドレスおよび書き込むデータのサイズを記述したディスクリプタに沿って行われる．PEACH2 は Chaining DMA 機能を備えており，複数のディスクリプタをポインタ連結しておけば，先頭のディスクリプタに対する通信開始の命令を送ることで連続した DMA 処理が可能である．DMA 通信のレイテンシは PIO 通信のレイテンシと比べて大きいが，DMA の実測バンド幅は PIO のバンド幅より大きく，大きなデータの通信では DMA を用いる方が高速な通信が可能である．

2.2 HA-PACS/TCA

HA-PACS（Highly Accelerated Parallel Advanced System for Computational Sciences）は，筑波大学計算科学研究センターで開発・運用されている，アクセラレータ技術に基づく大規模 GPU クラスタシステムである．HA-PACS は，2012 年 2 月に運用が開始されたベースクラスタ部と，2013 年 10 月に運用が開始された TCA 部（HA-PACS/TCA）から成る．HA-PACS ベースクラスタはコモディティ製品により構成されているのに対し，HA-PACS/TCA にはコモディティ製品に TCA を通信機構として加えた構成である．HA-PACS/TCA は TCA アーキテクチャの実証環境計算機であり，PEACH2 ボードの実験環境でもある．本研究では，HA-PACS/TCA のみを用いる．

表 1 に HA-PACS/TCA の構成仕様を記し，HA-PACS/TCA の計算ノードの構成を図 1 に示す．それぞれの計算ノードは 2 ソケットの Intel Xeon E5-2680v2 (IvyBridge-EP) CPU と 4 つの NVIDIA K20X (Kepler GK110) GPU を演算装置として持つ．HA-PACS/TCA は 64 ノードから構成させるが，その 64 ノードは 16 ノードずつ 4 つのサブクラスタに分けられる．16 ノードのサブクラスタは 2 × 8 トーラスのトポロジで PEACH2 により接続されている．また，HA-PACS/TCA の全 64 ノードは，2 ポートの InfiniBand QDR によるフルバイセクションバンド幅の Fat Tree ネットワークによってもつながれている．

なお，HA-PACS/TCA においては，TCA は通信を行うデータ大きくなると InfiniBand に通信性能で逆転される．TCA の PEACH2 は PCIe Gen2 x8 技術を利用しているので 4 GB/s が理論ピークバンド幅であるが，dual-rail InfiniBand QDR はその倍の理論ピーク性能で MPI 通信が可能だからである [*2]．

[*2] Dual-rail InfiniBand QDR の理論ピーク性能は 8 GB/s であり，PCIe Gen3 x8 と同等の性能．

表 1 HA-PACS/TCA システムの構成仕様

ノード構成	
マザーボード	SuperMicro X9DRG-QF
CPU	Intel Xeon E5-2680 v2 2.8 GHz × 2 (IvyBridge 10 cores / CPU)
メモリ	DDR3 1866 MHz × 4 ch, 128 GB (=8 × 16 GB)
ピーク性能	224 GFlops / CPU
GPU	NVIDIA Tesla K20X 732 MHz × 4 (Kepler GK110 2688 cores / GPU)
メモリ	GDDR5 6 GB / GPU
ピーク性能	1.31 TFlops / GPU
インターコネクト	InfiniBand: Mellanox Connect-X3 Dual-port QDR TCA: PEACH2 board (Altera Stratix-IV GX 530 FPGA)
システム構成	
ノード数	64
インターコネクト	InfiniBand QDR 108 ports switch × 2 ch
ピーク性能	364 TFlops

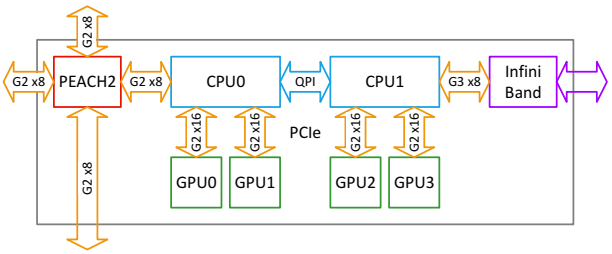


図 1 HA-PACS/TCA のノード構成

表 2 性能の測定条件

OS	CentOS Linux 6.4 Linux 2.6.32-358.el6.x86_64
GPU プログラミング環境	CUDA 6.5
C コンパイラ	Intel Compiler 14.0.3 (Composer XE 2013 sp1.4.211)
MPI 環境	MVAPICH2 GDR 2.1a

3. Collective 通信

本節では，TCA による Collective 通信の実装及びその性能評価結果を述べる．性能の測定には HA-PACS/TCA の 1 サブクラスタ（最大 16 ノードまで）を用いる．HA-PACS/TCA の構成仕様を表 1 に，性能の測定条件を表 2 に記す．本研究では，1 ノードあたり 1 GPU のみを用いており，これ以降の記述における利用プロセス数は利用ノード数と一致する．

TCA を用いた実装との比較のために，本稿では GPU クラスタを意識した先進的 MPI 実装の一つである MVAPICH2-GDR 2.1a（以下 MV2GDR）[14] による性能も適宜示す．MV2GDR は，TCA と同様に GPU-Direct for RDMA（GDR）技術 [13] が使われている．GDR により GPU メ

モリと InfiniBand ボードとの間で直接アクセスが可能となり，InfiniBand を経由した小サイズデータ通信の際のレイテンシが通常の MVAPICH2 と比べて改善されている．MV2GDR は 8 KB 以下の通信までは GDR を用い，それ以上のサイズの場合は CPU メモリを介してパイプライン的にデータを送受信するようプロトコルがスイッチされる．

本稿で述べる Collective 通信は，表 3 のように通信前と通信後のデータの状態を指定できる [15]．表において，x はサイズ m のデータであり，Collective 通信でデータはプロセス数 p によりサイズ m/p の部分データ x_i $(i = 0, 1, \ldots, p-1)$ に分割される．表 3 の Reduce/Allreduce 通信における $x^{(j)}$ は各プロセスが初期に各々持つ他プロセスと Reduction されるデータであり，$\sum_j x^{(j)}$ は Reduction 結果を表す（総和は最も頻出する Reduction 操作であり，本稿では総和操作の結果を示す）．なお本研究では，使用するプロセス数 p は 2 のべき乗数（$p = 2, 4, 8,$ or 16）のみに限定する．以降においてはプロセスランク 0 を Root プロセスとみなして実装を説明する．

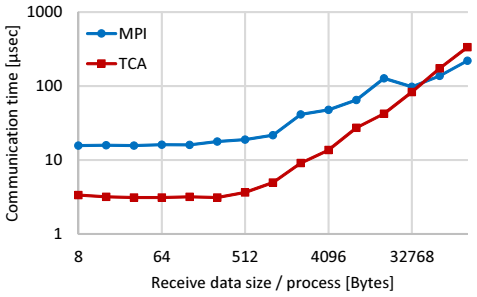
3.1 Scatter

Scatter 操作は，Root プロセスが持つデータを他のプロセスへ送る．この操作は，One-to-all personalized communication とも呼ばれる．Scatter 操作では，合計 m サイズのデータを送るが，プロセスランク 0 には始めの m/p のサイズのデータを送り，ランク 1 には次のサイズ m/p のデータを送るということを行い，最大ランクのランク $p-1$ には最後のサイズ m/p のデータを送る．

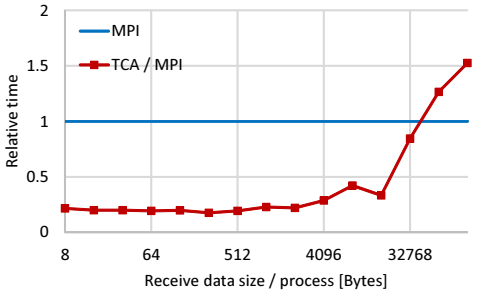
Scatter の実装としては，Root プロセスが各プロセスに対して順番にデータを送信するという単純なものを採用している．Scatter に関しては，Chaining DMA 機能を活用し，Root プロセスが一回の DMA 通信の開始命令を発行するだけで済むように実装している．図 2 に性能を示す．TCA による Scatter 実装は，小さいサイズのときに `MPI_Scatter` 実装の 20%の通信時間で済んでいる．この Scatter 実装の最大通信性能は，TCA の GPU 間通信のバンド幅により制限される．各プロセスごとに送られるデータサイズ（m/b）が 64 KB 以上になると性能が上がらなくなり，上限性能 2.8 GB/s で飽和する．その性能が上がらなくなるサイズである 64 KB で `MPI_Scatter` に性能が逆転される．

3.2 Reduce

Reduce（All-to-one reduction とも呼ばれる）は，各プロセスにあるデータ同士に加算や乗算などの結合則を満たす演算を施し，その演算結果を Root プロセスに集める Collective 通信である．Reduce の実装に関しては TCA を用いて 2 種類の実装方法を取った．その Reduce 実装の性能を図 3 に示す．図 3 の `TCA_CUDA` は，Scatter 実装と同様



(a) 通信時間



(b) MPI の時間を 1 としたときの相対時間

図 2 Scatter 実装の性能（8 プロセス使用時）

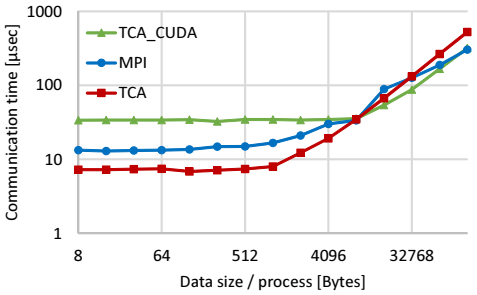


図 3 Reduce 実装の通信時間（8 プロセス使用時）

に GPU-GPU 間通信のみを利用し GPU メモリにデータを Gather し，その後に CUDA カーネルを呼ぶことで GPU で Reduction を行う Reduce 実装の結果である．この方法は 16 KB から 64 KB のサイズのデータに対する Reduce 通信に関して `MPI_Reduce` より高い性能を示すが，その CUDA カーネルを発行のためのレイテンシが 14 μsec と大きくそのレイテンシがボトルネックとなる．そこで，別の Reduce 実装として始めに TCA の GPU-CPU 間通信を利用して CPU（ホスト）メモリにデータを Gather し，CPU で Reduction 処理を行い，その Reduction 結果を GPU へコピーする方法も実装した．図 3 の `TCA` は，その CPU で Reduction を行う実装の性能であり，こちらの実装の方が `TCA_CUDA` よりも 8 KB のサイズまでは高い性能を発揮する．なお，8 Bytes サイズの Reduce における通信レイテンシは，`TCA` が 7 μsec，`TCA_CUDA` が 33 μsec，そして `MPI` が 13 μsec である．

表 3　本稿で述べる Collective 通信（4 プロセス使用時）[15]

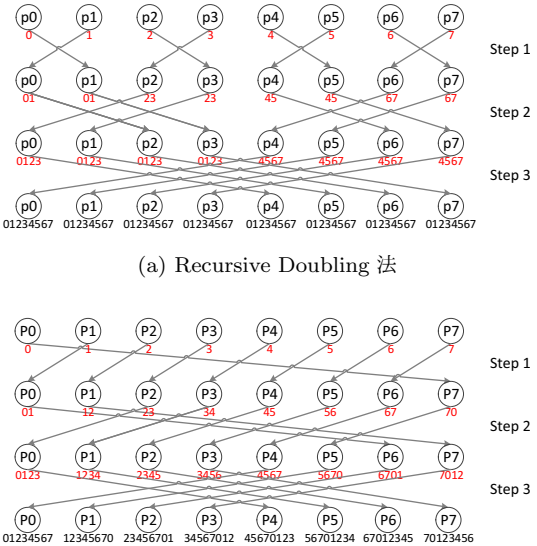
Collective 通信名	通信前				通信後			
Scatter	Rank 0	Rank 1	Rank 2	Rank 3	Rank 0	Rank 1	Rank 2	Rank 3
	x_0				x_0			
	x_1					x_1		
	x_2						x_2	
	x_3							x_3
Reduce	Rank 0	Rank 1	Rank 2	Rank 3	Rank 0	Rank 1	Rank 2	Rank 3
	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$			
Allgather	Rank 0	Rank 1	Rank 2	Rank 3	Rank 0	Rank 1	Rank 2	Rank 3
	x_0				x_0	x_0	x_0	x_0
		x_1			x_1	x_1	x_1	x_1
			x_2		x_2	x_2	x_2	x_2
				x_3	x_3	x_3	x_3	x_3
Allreduce	Rank 0	Rank 1	Rank 2	Rank 3	Rank 0	Rank 1	Rank 2	Rank 3
	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$

3.3 Allgather

Allgather（All-to-all broadcast とも呼ばれる）は，全プロセスが一斉に各プロセスへ Gather 操作を行う Collective 通信である．Allgather のアルゴリズムにも様々なものがあるが [16], [17], [18] が，先行研究で我々は Allgather アルゴリズムごとの性能の違いを調べた [5]．本稿ではその中でもっとも高い性能を示した Recursive Doubling 法 [17] による Allgather について述べる．図 4(a) に Recursive Doubling 法の通信パターンを示す．このアルゴリズムは，自ノードとデータを交換する通信相手ノードとの距離（ホップ数）を毎通信ステップごとに倍にしていく．通信ステップ数は $\log_2 p$ で済むが毎回通信相手が異なり，TCA のような mesh/torus 系のトポロジでは通信経路において衝突を起こしてしまう．Allgather の場合にはデータ通信量も毎ステップごとに倍になる．そこで，通信の衝突の影響を最小化するため，衝突はメッセージ長の短い最初のステップでできるだけ起きるようにしている．また，この Allgather 実装に関しては，直前のステップで送られていきたデータを他のプロセスに送る必要があるため Chaining DMA は使えず，ステップ数の分だけ待ち合わせをし通信開始命令を発行しなければならない．Allgather 実装の性能測定結果を図 5 に示す．128 Bytes サイズの Allgather における通信レイテンシは，`TCA` が 11 μsec であり `MPI` が 17 μsec である．TCA による Allgather は 128 KB より大きいサイズにおいて `MPI_Allgather` に性能逆転される．

3.4 Allreduce

Allreduce（All-to-all reduction とも呼ばれる）は，全プロセスが一斉に各プロセスへ Reduce 操作を行う Collective 通信である．現時点において，実装できているのは TCA の PIO 通信を利用した CPU メモリ間の Allreduce だけである．将来的に GPU 間の Allreduce は，一旦データを



(a) Recursive Doubling 法

(b) Dissemination 法

図 4　通信パターン．図中の赤数字は，その通信ステップで送信するデータを表す．

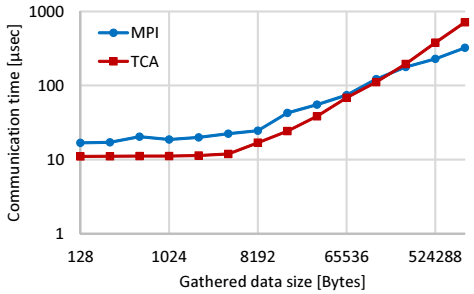


図 5　Allgather 実装の通信時間（8 プロセス使用時）

CPU に送った後で CPU 間の Allreduce 通信を行い，最後に結果を GPU に戻すことで実装する．Allreduce のアルゴリズムとしては Dissemination 法 [18], [19] を用いてい

表 4　Allreduce の通信時間（単位は μsec）

プロセス数	2	4	8	16
TCA	2.1	3.4	5.4	7.2
MPI	4.9	7.9	11.1	15.4

る [*3]．Dissemination 法の通信パターンを図 4(b) に示す．この Allreduce のアルゴリズムは，$\log_2 p$ 回の通信ステップが必要な方法で，Recursive Doubling 法と同様に通信先ノードとの距離を毎通信ステップごとに倍にしていく方法である．ただし Dissemination 法は，データを交換するのではなく全ノードの通信方向が同じになるようにデータを流していく．その CPU 間 Allreduce 実装の通信時間の測定結果を表 4 に示す．これは，倍精度スカラー変数（8 バイトのデータ）を Allreduce した場合の結果であり，通信のレイテンシにより性能が決まっている．表に示した結果からもわかるように，TCA による実装は `MPI_Allreduce` の半分以下の通信時間で Allreduce を実現する．TCA の CPU 間 PIO 通信のレイテンシの短さは，Allreduce 通信において有効に働いている．

4. CG 法実装への Collective 通信実装の適用

CG 法は，対称正定値行列を係数行列とする連立一次方程式 $Ax = b$ を解くための反復法である．ここで A は $N \times N$ の対称正定値行列であり，x および b は N 次元ベクトルである．本研究では，行列データの格納形式は，Compressed Row Storage (CRS) 形式 (CSR: Compressed Sparse Row 形式とも呼ばれる)[20] を用いる．浮動小数点演算は倍精度の実数に対して行う．なお，CG 法は前処理を行うことで収束性能を高められる可能性があるが，本研究の実装では前処理は行っていない．

本研究では，比較的小サイズの行列による強スケーリングを想定し，CG 法の並列化として行列 A を単純に一次元分割する手法を用いる．CG 法を並列化するために，疎行列 A を行方向にほぼ均等にプロセス数でデータ分割し，かつベクトル x, b も同割合で均等に分割し各プロセスに初期データとして持たせる．つまり，プロセス数を p と記述し $n = \lfloor N/p \rfloor$ とするとき，各プロセスは $n \times N$ の A の部分行列および n 次元の b と x の部分ベクトルを持つ (ただし最大ランクのプロセスは $(N - (p-1)n) \times N$ 行列および $(N - (p-1)n)$ 次元ベクトルを持つ)．このようにデータ分割を行うことにより，CG 法の並列アルゴリズムは図 6 のように記述できる．

CG 法の主な演算は，疎行列ベクトル積計算 (SpMV: Sparse Matrix-Vector multiply)，内積計算 (DOT product)，ベクトル加算 (AXPY) である．図 6 のアルゴリズムは毎反復において ($k \geq 2$)，1 回の SpMV(図 6 の行 15)，3 回の DOT(行 6, 16, 20[*4])，3 回の AXPY(行 12, 18, 19) を行う．これらの行列とベクトルに対する 3 つの演算は基本的な演算であり，CUDA による NVIDIA 社の数値計算ライブラリでも提供されている．SpMV は cuSPARSE ライブラリ [21] に，DOT と AXPY は cuBLAS ライブラリ [22] にそれぞれ cusparseDcsrmv, cublasDdot, cublasDaxpy ルーチンとして実装されている．本研究では，各 GPU 内の処理ではそれらの cuSPARSE と cuBLAS ルーチンを利用する．

```
 1: x := Allgather(x_l)
 2: r_l := b_l − A_l x
 3: d_t := r_l^T r_l
 4: norm0 := sqrt(AllreduceSum(d_t))
 5: for k := 1, 2, ··· do
 6:    ρ_t := r_l^T r_l
 7:    ρ := AllreduceSum(ρ_t)
 8:    if k = 1 then
 9:       p_l := r_l
10:    else
11:       β := ρ/ρprev
12:       p_l := βp_l + r_l
13:    end if
14:    p := Allgather(p_l)
15:    q_l := A_l p
16:    α_t := ρ/(p_l^T q_l)
17:    α := AllreduceSum(α_t)
18:    x_l := αp_l + x_l
19:    r_l := −αq_l + r_l
20:    d_t := r_l^T r_l
21:    norm := sqrt(AllreduceSum(d_t))
22:    if norm/norm0 < ε then
23:       break
24:    end if
25:    ρprev := ρ
26: end for
```

図 6　CG 法の並列アルゴリズム．各変数の下付き文字 “l” および “t” は，各プロセスごとにローカルに持つ部分データおよび一時データであることをそれぞれ表す．

各反復において，図 6 の並列アルゴリズムは，以下のようなデータ通信が必要である．

- SpMV 計算 (図 6 の行 15) を行う前に，全プロセスが各プロセスに均等に分散されているベクトルデータ p_l を集める必要がある (Allgather)．
- 各プロセスごとの DOT 計算（行 6, 16, 20）の後に，そのローカルなベクトル内積の総和を計算し，全プロセスがその総和を持つ必要がある (Allreduce)．

この Collective 通信に TCA により実装した Allgather と Allreduce を利用する．Allgather は各プロセスが持っているデータブロックを他のプロセスとやりとりし，Allreduce は倍精度の場合は 8 バイトという非常に少量のデータを他プロセスとやりとりする．以上の特徴から，Allgather は TCA の GPU 間 DMA 通信による実装を利用し，Allre-

[*3] Allreduce アルゴリズムごとの性能の違いについても以前に調べ，その中で Dissemination 法が最も良い性能を示している [5].

[*4] ベクトルの 2-ノルムは内積計算を用いて計算する．

表 5　性能評価に用いた疎行列の特性

行列名	行数（N）	非零要素数（nnz）	nnz/N
nasa2910	2,910	174,296	59.9
s1rmq4m1	5,489	281,111	51.2
smt	25,710	3,753,184	146.0
nd3k	9,000	3,279,690	364.4
nd6k	18,000	6,897,316	383.2
nd12k	36,000	14,220,946	395.0
nd24k	72,000	28,715,634	398.8

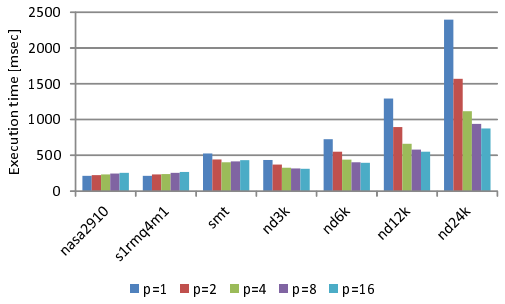


図 7　TCA を用いた CG 法の実行時間

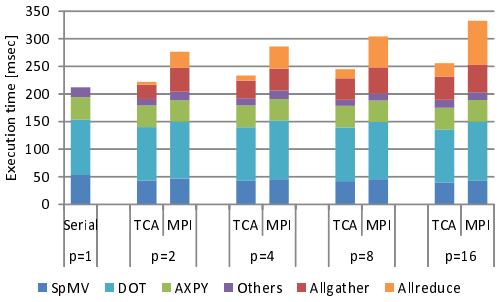
duce は TCA の CPU 間 PIO 通信による実装を利用する [*5]．TCA による通信を行うためには，DMA 通信の場合は DMA ディスクリプタを作成する必要があり，PIO 通信の場合は PIO 領域の準備が必要である．一度通信準備をしたら同じ通信を何度でも行えるので，CG 法実装の Allgather と Allreduce 通信においては，初めて通信を行う前に一度だけ通信準備をするようにしている．

4.1　CG 法実装の性能

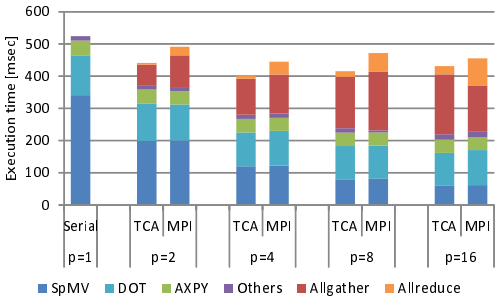
性能評価に用いる疎行列は，The University of Florida Sparse Matrix Collection[23] から取得した表 5 に記す実数の対称正定値行列である．なお，実際の使用において，CG 法は解が収束するまで反復する必要があるが，本研究では性能評価のために反復回数を 1000 回に固定している [*6]．

図 7 に，各疎行列に対する TCA を用いた CG 法の実行時間を示す．この図では，プロセス数を 1, 2, 4, 8, 16 と増やしたときの実行時間を示している．行列 nd3k, nd6k, nd12, nd24 に関しては，プロセス数を増やすことにより性能向上を達成できている．行列 smt に関しては，4 プロセス用いるときが最も高い性能を示す．残り 2 つの小さめの行列（nasa2910 と s1rmq4m1）に関しては， プロセス数を増やすと性能は悪化する．

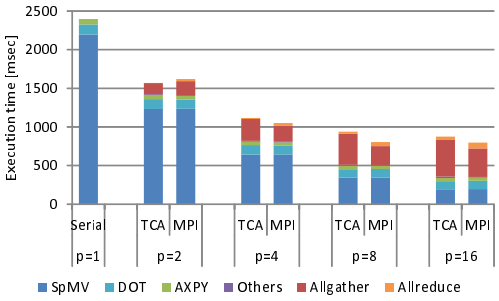
性能を更に分析するために，各処理ごとの合計実行時間の内訳を見る．図 8 に 3 種の異なるサイズの行列（nasa2910, smt, nd24k）に対する各処理の内訳を示す．この内訳はプロセスランク番号 0 の結果で，比較のために MPI を用いた実装による結果も併記している．この図の結果から云えることは，nasa2910 のように行数 n（および行あたりの非零要素数）が小さすぎると，並列化をしても計算処理（SpMV, DOT, AXPY）の実行時間がほぼ一定で変わらず，データ通信時間の分だけ遅くなってしまうということである．それに対して nd24k のように行列サイズが大きすぎると，並列化により性能は向上するが，TCA による Allgather の通信時間が MPI のものより長くなってしまい，結果として TCA による実装の方が MPI による実装より性能が劣るものになってしまう．それらの中間ぐらいの行列サイズ（smt のような 15,000 行から 35,000 行ほどのサイズ）の行列に対しては，TCA を用いることで MPI を用いるよりも良い性能を実現できている．



(a) nasa2910



(b) smt



(c) nd24k

図 8　CG 法実装の各処理実行時間の内訳 (ランク 0)

[*5] cublasDdot は計算したローカルな内積を CPU 側にも返すことができるので，それにより返されたスカラー値を CPU 間 Allreduce を利用して内積を計算する．

[*6] 本研究は CG 法における 1 反復当たりの処理時間の評価を目的としており，収束するか否かは問題としない．性能評価における反復当たりのバラ付きによる誤差をなくすための十分な反復回数として 1000 回を選んだ．

5. おわりに

本研究では，TCA による Collective 通信の実装を行い，HA-PACS/TCA GPU クラスタにおいてその実装の性能評価を行なった．本稿では Scatter, Reduce, Allgather, Allreduce 通信の実装とその性能を述べた．TCA はノードを跨ぐ通信を低レイテンシで実現するが，その特徴により小さいサイズの Collective 通信については MPI による Collective 通信と比べて高速にその通信処理を行うことが可能であった．

また，実装した Collective 通信を利用した CG 法の実装を行い，その性能について評価を行った．CG 法の並列アルゴリズムとしては，SpMV 計算に必要なデータを Allgather で集め，ベクトル内積を Allreduce を利用して実現するものを用いている．TCA を用いた実装は疎行列のサイズ（行数）が数千から数万の場合においては MPI を用いた実装よりも高い性能を示した．しかし，GPU を利用する際に必要な CUDA カーネルの発行のためのレイテンシといった通信以外の処理にかかる時間がボトルネックとなり，TCA を利用しても必ずしもストロング・スケーラビリティを改善できるというわけではないことがわかった．

今後は Collective 通信実装の更なる改善を行うと共にその性能についてより詳細な解析を行う．また，どのような通信と計算を含んだアプリケーションが TCA を用いるのに適しているのかを明らかにしていくことも今後の課題である．

謝辞 本研究の一部は JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」による．また，HA-PACS/TCA システムの利用は筑波大学計算科学研究センター学際共同利用プログラム（課題名「密結合演算加速機構アーキテクチャに向けた GPGPU アプリケーション」）による．

参考文献

[1] 塙 敏博，児玉祐悦，朴 泰祐，佐藤三久：Tightly Coupled Accelerators アーキテクチャに基づく GPU クラスタの構築と性能予備評価，情報処理学会論文誌. コンピューティングシステム， Vol. 6, No. 3, pp. 14–25 (2013).

[2] Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators, *Proc. IPDPSW 2013*, IEEE, pp. 1030–1039 (2013).

[3] 朴 泰祐，佐藤三久，塙 敏博，児玉祐悦，高橋大介，建部修見，多田野寛人，蔵増嘉伸，吉川耕司，庄司光男：演算加速装置に基づく超並列クラスタ HA-PACS による大規模計算科学，情報処理学会研究報告，Vol. 2011-HPC-130, No. 21, pp. 1–7 (2011).

[4] 藤井久史，藤田典久，塙 敏博，児玉祐悦，朴 泰祐，佐藤三久，藏増嘉伸，Clark, M.：GPU 向け QCD ライブラリ QUDA の TCA アーキテクチャ実装の性能評価，情報処理学会研究報告，Vol. 2014, No. 43, pp. 1–9 (2014).

[5] 松本和也，塙 敏博，児玉祐悦，藤井久史，朴 泰祐：密結合並列演算加速機構 TCA を用いた GPU 間直接通信による CG 法の実装と予備評価，情報処理学会研究報告，Vol. HPC-144, No. 12, 情報処理学会，pp. 1–9 (2014).

[6] 藤井久史，塙 敏博，児玉祐悦，朴 泰祐，佐藤三久：TCA アーキテクチャによる並列 GPU アプリケーションの性能評価，情報処理学会研究報告，Vol. HPC-140, No. 37, pp. 1–6 (2013).

[7] Sack, P. and Gropp, W.: Faster topology-aware collective algorithms through non-minimal communication, *ACM SIGPLAN Notices*, Vol. 47, No. 8, pp. 45–55 (2012).

[8] Barnett, M., Shuler, L., van de Geijn, R., Gupta, S., Payne, D. G. and Watts, J.: Interprocessor collective communication library (InterCom), *Proc. IEEE Scalable High Performance Computing Conference 1994*, IEEE Computer Society, pp. 357–364 (1994).

[9] 松本和也，塙 敏博，児玉祐悦，藤井久史，朴 泰祐：密結合並列演算加速機構 TCA を用いた GPU 間直接通信による Collective 通信の実装と予備評価，情報処理学会研究報告，Vol. HPC-147, No. 23, 情報処理学会，pp. 1–10 (2014).

[10] Cevahir, A., Nukada, A. and Matsuoka, S.: High Performance Conjugate Gradient Solver on Multi-GPU Clusters using Hypergraph Partitioning, *Computer Science - Research and Development*, Vol. 25, No. 1-2, pp. 83–91 (2010).

[11] Chen, C. and Taha, T. M.: A Communication Reduction Approach to Iteratively Solve Large Sparse Linear Systems on a GPGPU Cluster, *Cluster Computing* (2013).

[12] Kodama, Y., Hanawa, T., Boku, T. and Sato, M.: PEACH2: An FPGA-based PCIe Network Device for Tightly Coupled Accelerators, *HEART 2014* (2014).

[13] NVIDIA: NVIDIA GPUDirect, (online), available from ⟨https://developer.nvidia.com/gpudirect⟩ (accessed Jan 15, 2015).

[14] Panda, D. K.: MVAPICH2-GDR (MVAPICH2 with GPUDirect RDMA), The Ohio State University (online), available from ⟨http://mvapich.cse.ohio-state.edu/overview/⟩ (accessed Jan 15, 2015).

[15] Chan, E., Heimlich, M., Purkayastha, A. and van De Geijn, R.: Collective communication: theory, practice, and experience, *Concurrency and Computation: Practice and Experience*, No. July, pp. 1749–1783 (2007).

[16] Grama, A., Karypis, G., Kumar, V. and Gupta, A.: *Introduction to Parallel Computing*, Addison-Wesley, 2nd edition (2003).

[17] Kogge, P. M. and Stone, H. S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, *IEEE Transactions on Computers*, Vol. C-22, No. 8, pp. 786–793 (1973).

[18] Bruck, J. and Ho, C.-T.: Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 11, pp. 1143–1156 (1997).

[19] Hensgen, D., Finkel, R. and Manber, U.: Two Algorithms for Barrier Synchronization, *International Journal of Parallel Programming*, Vol. 17, No. 1, pp. 1–17 (1988).

[20] Saad, Y.: *Iterative Methods for Sparse Linear Systems*, SIAM, 2nd edition (2003).

[21] NVIDIA: cuSPARSE Library, (online), available from ⟨http://docs.nvidia.com/cuda/cusparse/index.html⟩

(accessed Jan 15, 2015).
[22] NVIDIA: cuBLAS Library, (online), available from ⟨http://docs.nvidia.com/cuda/cublas/index.html⟩ (accessed Jan 15, 2015).
[23] Davis, T. A. and Hu, Y.: The University of Florida Sparse Matrix Collection, *ACM Transactions on Mathematical Software*, Vol. 38, No. 1 (online), available from ⟨http://www.cise.ufl.edu/research/sparse/matrices⟩ (2011).