Vol.2014-HPC-144 No.12
2014/5/27

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

松本 和也[1,a)]　塙 敏博[2]　児玉 祐悦[1,3]　藤井 久史[3]　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．TCA の有効性を確かめることを目的として，本研究では TCA を用いた Conjugate Gradient (CG) 法の実装・性能評価を行っている．本稿では，TCA を用いた CG 法の実装について述べるとともに、TCA 実証環境である GPU クラスタ HA-PACS/TCA における予備性能評価結果を述べる．GPU クラスタでの CG 法実装のために Allgather 集団通信と Allreduce 集団通信を，TCA による通信で実現する．TCA を用いた実装と MPI を用いた実装の比較を行い，Allgather においては比較的小さなデータサイズでは TCA が有利であること，Allreduce においては TCA を用いることにより MPI の半分ほどの通信時間で済むことを示す．そして，TCA を用いた CG 法の実装は MPI を用いた実装より最大で 20%実行時間を短縮できることを示す．

1. はじめに

現在の GPU は高い演算性能とメモリバンド幅を誇り，その利点を活かした GPGPU（General Purpose GPU）処理がさかんに行われている．GPU は消費電力あたりの演算性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，ノードをまた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]．2013 年 10 月からは，GPU クラスタ HA-PACS[2] の拡張部として，TCA を導入した HA-PACS/TCA が稼働している．

TCA の有効性を実証する研究では，pingpong 性能の評価 [1] をはじめとして、いくつかの基本的性能評価が行われている [3], [4], [5]．しかし，実アプリケーションに対する評価はまだまだ行われていない．また，TCA による通信は片方向通信で，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．

本研究では，TCA を用いた共役勾配法（CG 法: Conjugate Gradient method）の実装とその性能評価を行う．CG 法は連立一次方程式を解くための反復法の一つである [6]．複数ノードにおいて動作する CG 法として，Allgather 集団通信と Allreduce 集団通信を用いる並列アルゴリズムを実装する．それらの Allgather，Allreduce，CG 法の性能を評価し，TCA の有効性と課題について明らかにする．また，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 [7], [8] が，それらは行列サイズ（行数）が数十万以上と大きめ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，そのような並列処理性能が十分に引き出せないような小さめの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．

本稿の構成は下記の通りである．2 節では，TCA とその実証環境である HA-PACS/TCA についての説明を行う．3 節では，TCA を用いた CG 法実装について記す．この節では CG 法の並列アルゴリズムと，そのアルゴリズムに必要な集団通信の特徴も述べる．続いて 4 節では，TCA を用いた実装の性能評価結果を記述する．最後の 5 節では，本稿のまとめと今後の課題を記す．

[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
[a)] kzmtmt@ccs.tsukuba.ac.jp

2. 密結合並列演算加速機構 TCA

密結合並列演算加速機構 TCA は，アクセラレータ（計算加速機構）間の直接結合を実現する通信機構技術のことである [1]．TCA の基本的なハードウェア技術は，PCI-Express (PCIe)[9] を応用したものである．現時点で TCA が対象としているアクセラレータは，NVIDIA 社の Kepler アーキテクチャの Tesla K20 製品ファミリである．Kepler アーキテクチャでは，GPUDirect Support for RDMA[10] が利用可能になり，PCIe デバイスから GPU メモリに対する直接的なデータ読み書きが可能になっている．

PEACH2 (PCI Express Adaptive Communication Hub version 2) は，TCA 用インタフェースボードである [1]．PEACH2 ボード同士を PCIe ケーブルによりつなぐことで，TCA のクラスタシステムを構築する．PEACH2 は，ノード間のデータ通信を低レイテンシで実現する．

TCA では，ノード間の通信に PCIe を直接用いる．従来の GPU クラスタでは，ノードをまたぐ GPU 間の通信において最低 3 回のデータコピーが必要であった．例えば，ノード A の GPU A からノード B の GPU B に通信を行う際には，次のコピーが必要となる．

(1) GPU A のメモリから，PCIe 経由でノード A のメモリへコピー．
(2) ノード A のメモリから，ネットワーク経由でノード B のメモリへコピー．
(3) ノード B のメモリから，PCIe 経由で GPU B のメモリへコピー．

TCA では，ネットワーク経由のコピーを PEACH2 で置き換えることにより，PCIe プロトコルのままノード A の GPU A からノード B の GPU B へと通信することが可能となる．

PEACH2 チップは，4 つの PCIe Gen2 x8 ポートを持つ．1 ポートはホストとの接続に用い，2 ポートは隣接ノードの PEACH2 とのリングトポロジを構成するために使われ，残りの 1 ポートはリングトポロジー間の対向ノードの PEACH2 ノード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 [1]．PIO 通信は，CPU の store 操作によりリモートノー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，少量データの通信に向いている．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，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．しかし，ディスクリプタはホスト上に保存しておく必要があるため，通信開始時にディスクリプタを PEACH2 に転送するオーバーヘッドが存在する．このオーバーヘッドを避け，より軽量な DMA 命令発行を可能とするために，PEACH2 は各 DMA チャネルごとに 16 個を限度としてディスクリプタをレジスタに記憶しておく機能を備えている．この機能をレジスタモードと呼ぶ．

2.1 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 クラスタである [2]．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．本研究では，HA-PACS/TCA のみを用いる．

表 1 に HA-PACS/TCA の構成仕様を記す．HA-PACS/TCA の各ノードの構成を図 1 に示す．現在，HA-PACS/TCA の 16 ノードは，PEACH2 により 2 重リング状につながっており（各リングは 8 ノードから成る），HA-PACS/TCA システムは 4 つのそのような 16 ノードのグループから構成される．PEACH2 は各ノードの GPU0，GPU1 への直接アクセスが可能である．ただし，GPU2/GPU3 との間の QPI をまたぐ直接アクセスに関しては，性能が低く問題があるため無効にされている．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．

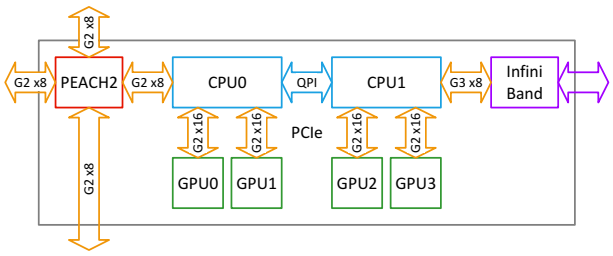


図 1 HA-PACS/TCA のノード構成

3. TCA を用いた CG 法の実装

3.1 CG 法

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を解くための反復法である．本稿において，連立一次方程式は $Ax = b$ と記す．ここで A は $N \times N$ の対称正定値

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

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)[11] 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．

CG 法の逐次アルゴリズムを図 2 に示す [6], [12]．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 2 のアルゴリズムは毎反復において ($k \geq 2$)，1 回の SpMV(図 2 の行 11)，3 回の DOT(行 4,12,15[*1])，3 回の AXPY(行 9,13,14) 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 [13] に，DOT と AXPY は cuBLAS ライブラリ [14] 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，それらの cuSPARSE と cuBLAS ルーチンを利用し，これに TCA による複数ノードの GPU 間通信を加えて並列 CG 法を実装する．

3.2　CG 法の並列化

本研究では，CG 法の並列化として一番単純な，行列 A を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

```
 1: r := b − Ax
 2: norm0 := sqrt(r^T r)
 3: for k := 1, 2, ··· do
 4:     ρ := r^T r
 5:     if k = 1 then
 6:         p := r
 7:     else
 8:         β := ρ/ρprev
 9:         p := βp + r
10:     end if
11:     q := Ap
12:     α := ρ/(p^T q)
13:     x := αp + x
14:     r := −αq + r
15:     norm := sqrt(r^T r)
16:     if norm/norm0 < ε then
17:         break
18:     end if
19:     ρprev := ρ
20: end for
```

図 2　CG 法の逐次アルゴリズム

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 3 のように記述できる．

各反復において，図 3 の並列アルゴリズムは，図 2 の逐次アルゴリズムとは以下の点で異なる．

- SpMV 計算 (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（図 3 の行 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 (AllreduceSum)．

Allgather と Allreduce (Sum) はどちらも集団通信であり，並列化により高速化を達成するためにはこの通信時間をできるだけ短くしなければならない．

本研究では，TCA を用いて Allgather と Allreduce の実装を行う．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

[*1] ベクトルの 2-ノルムは内積計算を用いて計算する．

```
 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 3 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．

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（レジスタモード）を用いて実装し，Allreduce は TCA の CPU 間 PIO 通信を用いて実装する．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．一度でも通信準備したものは，同じものを使い回すことが可能である．それゆえ Allgather と Allreduce の実装においても，初めて通信を行う前に通信準備をしてしまい，それを再利用するようにしている．

全対全の集団通信のアルゴリズムは数種類あるが，本研究では Ring 法，Neighbor Exchange 法 [15]，Recursive Doubling 法 [16]，Dissemination 法 [17], [18] の 4 種のアルゴリズムを実装した．なお，使用するプロセス数 p は 2 のべき乗数（$2, 4, 8, \cdots$）のみに限定する．図 4 にプロセス数 8 の時の各アルゴリズムの通信パターンを示す．以下にそれらのアルゴリズムの特徴を記す．

(1) Ring 法は，全プロセスがリング状に接続されていることを想定し，そのリングに沿ってデータを順に渡していく．Ring 法は通信完了までに $p-1$ の通信ステップ数が必要なアルゴリズムである．同じ方向の隣接ノードへとデータを流す方法なので，通信経路において衝突が起こらない．

(2) Neighbor Exchange 法は，隣接する 2 つのノードとのみデータをやり取りする．$p/2$ の通信ステップ数で処理が完了する．

(3) Recursive Doubling 法は，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アルゴリズムである．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通信経路における衝突を起こす可能性が高い方法である．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．

(4) Dissemination 法は，$\log_2 p$ 回の通信ステップが必要な方法で，Recursive Doubling 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．Allgather の場合は全てのデータが各ノードに集まった後にローカルシフトが必要である（Bruck のアルゴリズム [18] と呼ばれることもある）．

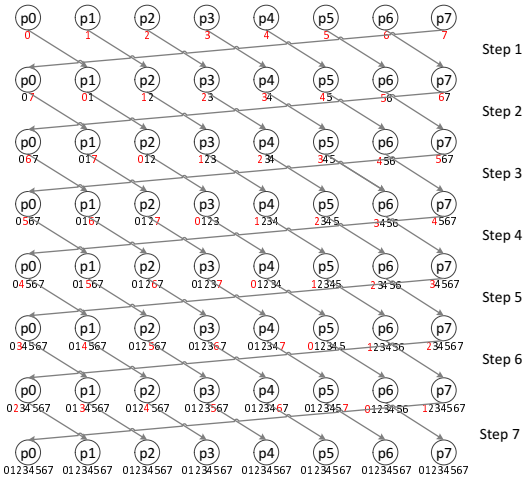
4. 性能評価

本節では，まず Allgather と Allreduce の性能について評価を行い，その後に CG 法実装の性能について評価を行う．性能の測定は HA-PACS/TCA を最大 16 ノードまで用いて行っている．HA-PACS/TCA の構成仕様は表 1 に，性能の測定条件は表 2 に記す．本研究では，1 ノードあたり 1GPU のみを用いている．これ以降の記述におけるプロセス数は利用ノード数と一致する．なお，性能評価の際には図 5 に示すようにノードを選択した．この選択の仕方も性能に影響を与えるが，すべて同じものを用いている．

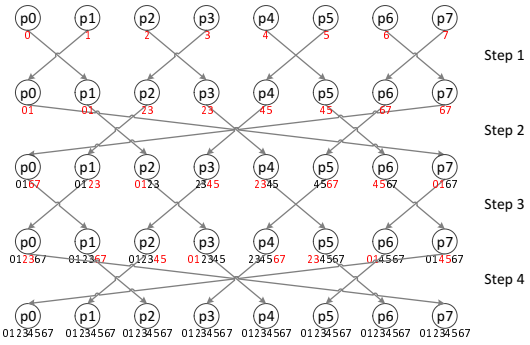
TCA を用いた実装との比較のために，本節では MPI 実装の一つである MVAPICH2-GDR 2.0b（以下 MV2GDR）[19] を用いた実装の性能も適宜示す．MV2GDR は，TCA と同様に GPU-Direct for RDMA (GDR) 技術 [10] が実装に使われている．GDR により GPU メモリと InfiniBand ボードとの間で直接アクセスが可能となり，InfiniBand を経由した小サイズデータ通信の際のレイテンシが通常の MVAPICH2 と比べて改善されている．MV2GDR は 8 KB 以下の通信までは GDR を用い，それ以上のサイズの場合は CPU メモリを介してパイプライン的にデータを送受信する．

表 2 性能の測定条件

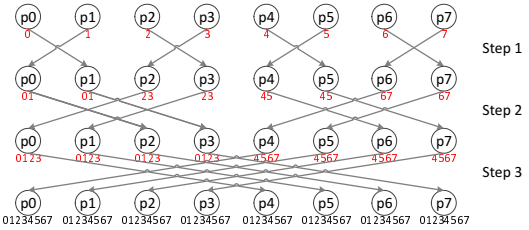
OS	CentOS Linux 6.4 Linux 2.6.32-358.el6.x86_64
GPU プログラミング環境	CUDA 5.5
C コンパイラ	Intel Compiler (Composer XE 2013.1.117)
MPI 環境	MVAPICH2 2.0b



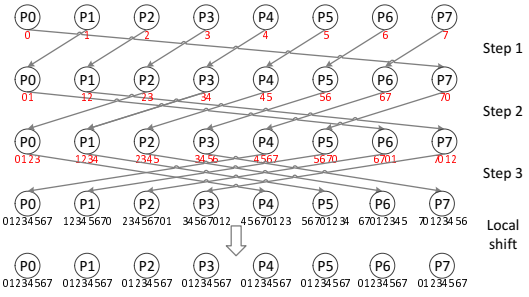
Ring 法



Neighbor Exchange 法



Recursive Doubling 法



Dissemination 法

図 4　全対全の集団通信アルゴリズムの通信パターン．図中の赤数字は，その通信ステップで送信/関係するデータを表す．

4.1　Allgather の性能

TCA のレジスタモードによる DMA 通信を用いて実装した Allgather 通信にかかる時間を表 3 に示す．この表 3 では，前節で述べた 4 種の集団通信アルゴリズムごとの

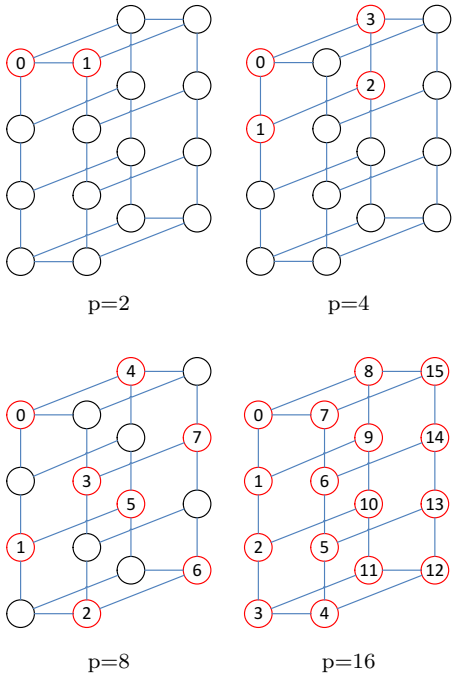


図 5　評価に用いているノード選択．赤丸は選択したノードを示し，数字はプロセスランクを表す．

μsec 単位の通信時間を，16 B, 2KB, 128 KB の 3 つのメッセージサイズ（各プロセスが通信開始前に持つデータサイズ）の場合について示している．Ring 法の 8 プロセス以上の場合と，Neighbor Exchange 法の 16 プロセスの場合は，現実装では対応していない．

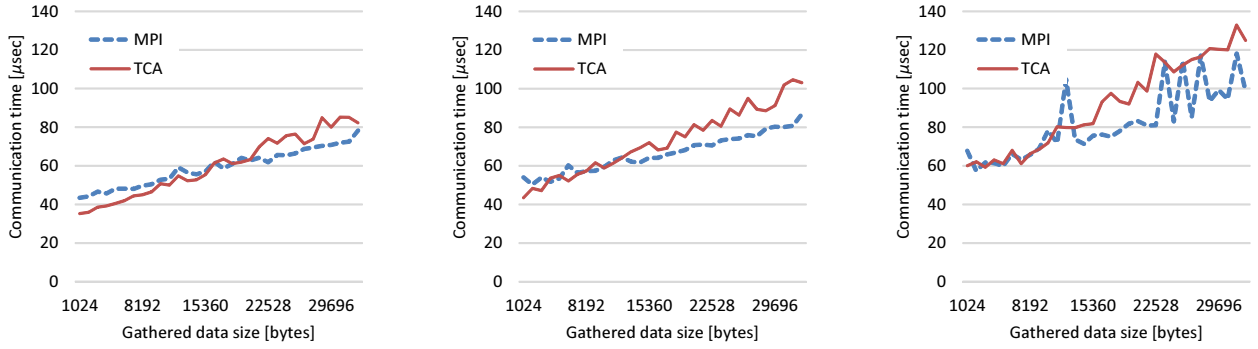
今回実装した 4 種の集団通信アルゴリズムの中では，Recursive Doubling 法が安定して高い通信速度を得られている．Recursive Doubling 法はプロセス数が 8, 16 と多くなると，メッセージサイズが 16 B と極めて小さい時以外では，最も通信時間が短い．それに対して，Dissemination 法はデータ通信の後にデータをシフトする必要があるため，そのオーバヘッドにより通信完了までにかかる時間が総じて長くなる．

TCA による GPU 間直接通信が MPI と比べてどのように有利なのかを調べるために，MV2GDR の MPI_Allgather ルーチンとの比較を行う．図 6 にプロセス数 2 の時の，図 7 にプロセス数 4, 8, 16 の時の比較結果をそれぞれ示す．なお，TCA の結果は Recursive Doubling 法によるものである．図の横軸の Gathered data size は各プロセスが Allgather をした結果として得られる配列データのバイト数を表し（各プロセスは通信開始前に (Gathered data size)/p バイトのデータを持つ），刻み幅は 1 KB である．

プロセス数 2 の時は，256 KB までは TCA を用いる方が速い（MPI の通信時間が 256 KB を境に短くなるのは，そのサイズで通信方式を切り替えているためだと思われる）．本研究の CG 法実装は倍精度実数を扱うので，このことは 32768 程度の行列サイズ（行数）までは TCA は MPI より有利であることを意味する．同様にしてプロセス数が 4, 8,

表 3　TCA の GPU 間直接通信技術を用いた各種アルゴリズム Allgather 実装の通信時間（50 回の平均で単位は μsec）

メッセージサイズ	16 B				2 KB				128 KB			
プロセス数 p	2	4	8	16	2	4	8	16	2	4	8	16
Ring	19.4	37.1	-	-	20.2	45.8	-	-	91.2	718.7	-	-
Neighbor Exchange	19.5	34.6	44.0	-	20.4	39.7	63.4	-	91.3	512.7	1368.1	-
Recursive Doubling	19.1	34.1	49.2	58.4	23.7	40.5	54.3	122.1	93.6	555.8	895.5	1147.5
Dissemination	50.5	64.9	74.4	85.3	49.7	72.1	84.9	130.4	114.7	560.4	939.4	3435.6



p=4　　p=8　　p=16

図 7　プロセス数 4, 8, 16 の時の Allgather 実装の平均通信時間

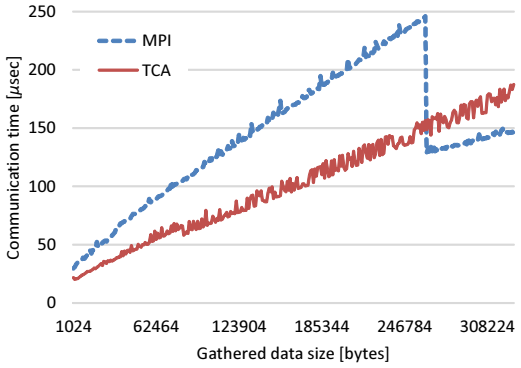


図 6　プロセス数 2 の時の Allgather 実装の平均通信時間

16 の時について比較を行うと次のことが云える．

- プロセス数が 4 の時，16 KB 付近（倍精度で 2048 要素）のサイズまでは TCA を用いる方が速い．
- プロセス数が 8 の時，7 KB 付近（倍精度で 896 要素）のサイズまでは TCA を用いる方が速い．
- プロセス数が 16 の時，3 KB 付近（倍精度で 384 要素）のサイズまでは TCA を用いる方が速い．

プロセス数が 4, 8, 16 の時は 2 の時と比べて MPI との性能差は大きくないが，それでも集めるデータサイズが小さい場合は TCA が有利となる．

プロセス数が増えると TCA と MPI の差が小さくなるが，TCA と MPI ではノード間接続に用いられているネットワークトポロジが異なるということが原因として挙げられる．TCA の方は PEACH2 により 2 重リング状に接続されているが，このトポロジでは同時にデータ通信を行うプロセス数が増えると通信経路での競合が起こり通信が遅延してしまう．それに対して MPI の方は，InfiniBand によるフルバイセクションバンド幅の Fat Tree ネットワークにより接続されているため，プロセス数が増えても通信経路での競合がほぼ起こらない．そのため，プロセス数の増加が通信時間へ与える影響は MPI の方が TCA と比べて小さくなっていると思われる．

4.2　Allreduce の性能

Allreduce (Sum) の通信時間の計測結果を表 4 に示す．表では，TCA による 4 種の集団通信アルゴリズム実装の通信時間と MV2GDR の MPI_Allreduce ルーチン実装の通信時間を比較している．本研究で行った実装の中では，Dissemination 法による Allreduce が最も速い．Allreduce では，Allgather とは違い Dissemination 法においてもデータシフトが要らないので，通信ステップ数が少ないアルゴリズムが良い結果を示している．また，Dissemination 法の TCA による実装は 16 プロセスの時でも MPI_Allreduce より通信時間が半分ほどで済んでいる．TCA の CPU 間 PIO 通信のレイテンシの短さは，Allreduce 処理において有効に働いていると云える．

4.3　CG 法の性能

CG 法の並列アルゴリズム（図 3）の TCA を用いた実装の性能測定結果を記す．前述の結果を考慮し，我々の CG 法実装では，Allgather 通信には Recurive Doubling 法を用い，Allreduce 通信には Dissemination 法を用いる．性能

表 4　Allreduce (Sum) の通信時間の比較（時間は 50 回の平均で，単位は μsec）

アルゴリズム / プロセス数	2	4	8	16
Ring	1.6	4.2	11.4	17.9
Neighbor Exchange	1.8	3.6	7.3	11.8
Recursive Doubling	1.6	3.6	7.4	9.6
Dissemination	1.6	3.6	6.4	7.5
MPI_Allreduce	5.5	8.3	12.3	16.6

評価に用いる疎行列は，The University of Florida Sparse Matrix Collection[20] 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[*2]．

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	2910	174296	59.90
s1rmq4m1	5489	281111	51.21
nd3k	9000	3279690	364.41
Pres_Poisson	14822	715804	48.29
nd6k	18000	6897316	383.18
smt	25710	3753184	145.98

図 8 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．nd6k に対しては，8 プロセスの時は 1 プロセスの時より 1.73 倍速い．nd3k や smt に対しても，程度の差はあるが計算速度の向上が見られる．しかし，nasa2910 や s1rmq4m1 のように行数が少ない問題に対しては，TCA を用いても並列化により性能を向上させることはできていない．

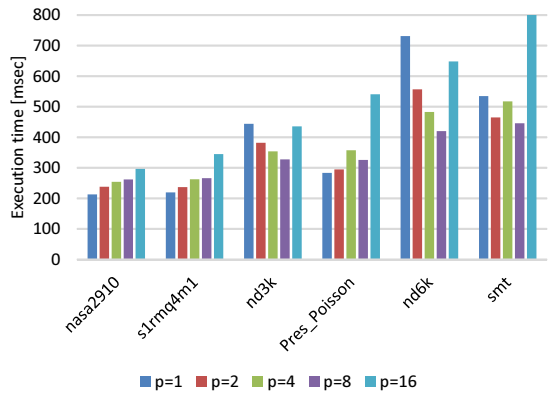
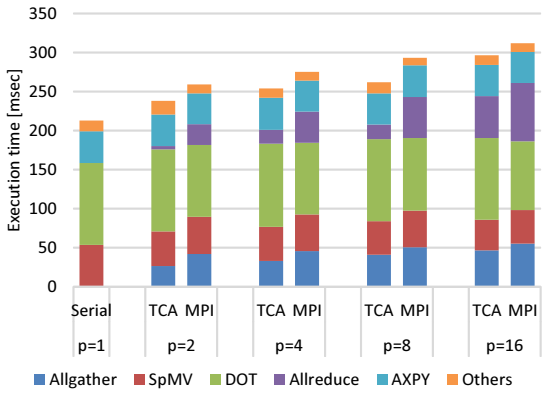


図 8　TCA を用いた CG 法の実行時間

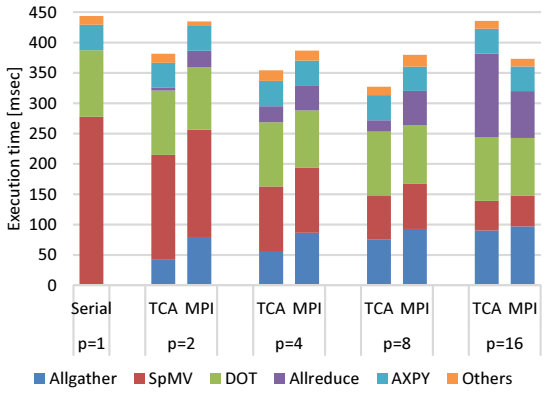
性能を更に分析するために，各処理ごとの合計実行時間の内訳を見る．図 9 に 3 種の異なるサイズの行列 nasa2910, nd3k, smt に対する各処理の内訳を示す．この内訳はプロセスランク番号 0 の結果で，比較のために MPI を用いた実装による結果も併記している．この図の結果から云えることは，nasa2910 のように行数 n（場合によっては一行あたりの非零要素数）が小さすぎると，並列化をしても計算処理（SpMV, DOT, AXPY）の実行時間がほぼ一定で変わらず，データ通信時間の分だけ遅くなってしまうということである．前述した通りこれらの計算処理には cuSPARSE と cuBLAS ライブラリのルーチンを利用しているが，本研究で利用した各ルーチンは最低でも 40 μsec ほどの時間が処理完了までかかる．TCA は小さいサイズのデータ通信では有利ではあるが，CG 法においては小さすぎると並列化する利点がなくなっている．それに対して，並列化により性能向上が達成できている場合（nd3k の p=2,4,8 の場合など）は，プロセス数を増やせば SpMV の計算時間が短くなり並列化が性能向上に寄与していることがわかる．プロセス数 2 の時には，TCA を用いることにより MPI を用いた場合と比べて，nd6k で 14%，smt で 20%実行時間を短縮できている．なお，SpMV の実行時間における TCA と MPI の差が，プロセス数の増加とともに大きくなる傾向があるが，その原因は現在調査中である．

図 9 の内訳によると，smt のプロセス数が 4 や 16 の場合などのように Allreduce の処理時間が非常に長くなっている．この原因を調べるためにランク 0 の場合だけではなく，他ランクの処理時間内訳を確認する．一例として，図 10 に smt に対するプロセス数 4 の時の各プロセスランクごとの各処理時間の内訳を示す．この図の内訳からは，TCA を用いた Allgather の処理時間にばらつきがあるために，同期完了待ちを伴う Allreduce の時間がかかっているように見えているだけということがわかる．図示はしていないが smt に対するプロセス数 16 の場合は，より Allgather のランクごとの通信時間のばらつきが大きく，特定の 1 つのプロセスの通信時間が他プロセスの通信時間より倍以上長く，そのことが著しい性能低下を招いている．これも図示してはいないがプロセス数 2 の場合でさえ，smt においてランク 0 とランク 1 の Allgather 通信の合計時間は倍以上違う．この性能低下の理由としては，前述したように TCA のトポロジが 2 重リングということも挙げられるが，現在の通信アルゴリズムでは通信のタイミングによっては特定のプロセスの通信遅延だけが大きくなってしまっていると考えられる．TCA の PEACH2 経由の通信で同じ通信経路を使う通信が同時に起こった場合は，FIFO でその通信経路が使われる．CG 法では毎反復の処理内容は同一であるので，Allgather を開始するタイミングの順番は毎反復ほぼ同じである．そのため，遅く Allgather 処理に入る特定のプロセスの通信遅延が大きくなっていると推測される．なお，この推測が正しいかどうかの確認作業は現在進めている．

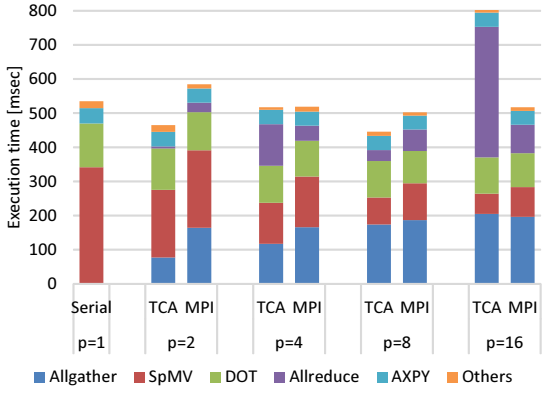
[*2] 本研究は CG 法における 1 反復当たりの処理時間の評価を目的としており，収束するか否かは問題としない．性能評価における反復当たりのバラ付きによる誤差をなくすための十分な反復回数として 1000 回を選んだ．



nasa2910 に対する内訳



nd3k に対する内訳



smt に対する内訳

図 9 CG 法実装の各処理実行時間の内訳 (ランク 0)

また，図 10 の TCA による結果と MPI による結果を見比べてみると，同程度の実行時間であるが内訳はだいぶ異なる．TCA を用いた実装は，Allgather の速さという点では MPI と比べると劣っている部分が大きいが，Allreduce 処理の速さという点では勝っている．そのため，図 7 の結果からは，あまり性能向上が見込めなさそうな問題サイズ (nd6k, smt の行数はそれぞれ 18000, 25710）においても，その Allreduce の速さにより MPI に勝る性能が得られる．ただし，この測定結果は単純に各部分の実行時間をランク毎に測定しただけであり，例えば集団通信において，その直前の状態で既にランク間に処理時間のバラ付きがあった場合，処理が進んでいるランクが遅いランクが追いつくのを待つ時間が考慮されていない．これを適切に調べるために，「時間」ではなく「時刻」を基準にした各プロセスの振る舞いを調べる必要があり，これについては調査中である．

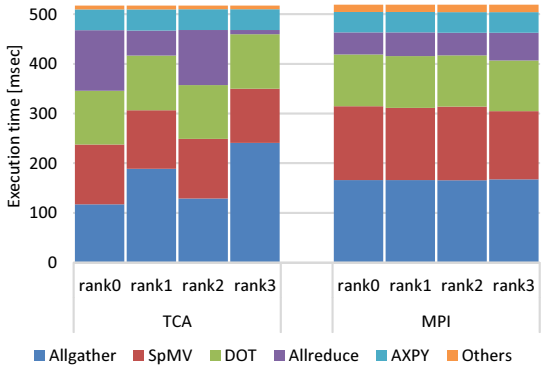


図 10 smt に対するプロセス数 4 の時の異なるプロセスランクごとの各処理実行時間の内訳

5. おわりに

本稿では，TCA による CG 法実装の性能評価結果を示した．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た．

TCA の GPU 間 DMA 通信を用いた Allgather は集めるデータサイズがそれほど大きくない場合には，MPI を用いるより高速であることを示した．Allreduce に関しては，TCA の CPU 間 PIO 通信により実現した．その通信にかかる時間は MPI の半分ほどと，レイテンシの小さいという TCA の利点が充分に発揮された．CG 法の実装の評価は，行数が 3000 程度から 30000 程度の疎行列に対して行った．行数が 25710 と Allgather の性能では MPI に劣る部分がある行列に対しても，その Allreduce の性能の高さにより性能を向上させることができた．

現在の TCA を用いた実装の問題点は，プロセス数が増えると，通信時間が急激に増えてしまうことである．Allgather 通信においてその傾向は顕著に見られ，CG 法実装の性能向上を阻む一因となっている．この問題を完璧に解消することは難しいが，通信時間をできるだけ短くするためにノードの割り当て方の最適化やよりトポロジーを考慮した通信アルゴリズムを検討することは今後の課題である．本稿で述べたソフトウェア的な集団通信の実装だけではなく，PEACH2 改良によるハードウェア的な集団通信の実装も進めていくことも予定している．

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．

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] 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).

[3] 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).

[4] 藤井久史，塙　敏博，児玉祐悦，朴　泰祐，佐藤三久，蔵増嘉伸，Clark, M.：GPU 向け QCD ライブラリ QUDA の TCA アーキテクチャによる実装，情報処理学会研究報告，Vol. HPC-143, No. 35, pp. 1–7 (2014).

[5] 中尾昌広，村井　均，下坂健則，田淵晶大，塙　敏博，児玉祐悦，朴　泰祐，佐藤三久：Tightly Coupled Accelerators アーキテクチャに向けた XcalableMP 拡張，情報処理学会研究報告，Vol. HPC-143, No. 34, pp. 1–8 (2014).

[6] Golub, G. H. and Van Loan, C. F.: *Matrix Computations*, The John Hopkins University Press, 4th edition (2013).

[7] 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).

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

[9] PCI-SIG: PCI Express Base Specification Revision 3.0 (2010).

[10] NVIDIA Corp.: NVIDIA GPUDirect, (online), available from ⟨https://developer.nvidia.com/gpudirect⟩ (accessed April 27, 2014).

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

[12] Naumov, M.: Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, Technical report, NVIDIA White Paper (2011).

[13] NVIDIA Corp.: cuSPARSE Library, (online), available from ⟨http://docs.nvidia.com/cuda/cusparse/index.html⟩ (accessed April 27, 2014).

[14] NVIDIA Corp.: cuBLAS Library, (online), available from ⟨http://docs.nvidia.com/cuda/cublas/index.html⟩ (accessed April 27, 2014).

[15] Chen, J., Zhang, L., Zhang, Y. and Yuan, W.: Performance Evaluation of Allgather Algorithms on Terascale Linux Cluster with Fast Ethernet, *Proceedings of the 8th International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA '05)*, IEEE, pp. 437–442 (2005).

[16] 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).

[17] 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).

[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 (online), DOI: 10.1109/71.642949 (1997).

[19] Panda, D. K.: MVAPICH2-GDR (MVAPICH2 with GPUDirect RDMA), The Ohio State University (online), available from ⟨http://mvapich.cse.ohio-state.edu/overview/mvapich2gdr/⟩ (accessed April 27, 2014).

[20] Davis, T. A. and Hu, Y.: The University of Florida Sparse Matrix Collection, *ACM Transactions on Mathematical Software*, Vol. 38, No. 1, pp. 1:1–1:25 (online), available from ⟨http://www.cise.ufl.edu/research/sparse/matrices⟩ (2011).