Vol.2016-OS-137 No.6
2016/5/30

In-memory MapReduceにおける最適なShuffle手法の検討

大黒 晴之[1,a)]　川島 英之[2]　建部 修見[2]

概要：大規模データをクラスタシステム上で並列分散処理する技術 MapReduce における最大のボトルネックはノード間でデータの交換を行う shuffle と呼ばれる処理であることが知られており，これまでに shuffle コストの削減を図る研究が数多くなされてきた．本研究では先端ネットワーク技術である InfiniBand と RDMA（Remote Direct Memory Access）を活用し，通信手段とアルゴリズムの 2 つの観点から最適な shuffle 手法に関して検討を行う．加えてこれら 2 要素が shuffle の性能に与える影響を純粋に測定するために独自 in-memory MapReduce 処理系を C/C++で実装し，性能評価を行う．通信手段に関しては RDMA と InfiniBand 上で IP 通信をエミュレートする IPoIB（IP over InfiniBand）の 2 種類を採用し，テキスト中の単語をその長さでグループ分けする GroupBy ワークロードでの比較では RDMA が *map + shuffle* フェーズにおいて 1.49 倍高速であることを確認した．通信アルゴリズムに関しては *Fully-Connected* と *Pairwise* の 2 手法を比較し，IPoIB での通信では map と shuffle のオーバラップを行う Fully-Connected が最大 10.8%効率的であり，加えて Pairwise の方が RDMA の適用による性能改善幅が大きいことを確認した．また独自処理系と Apache Spark の比較実験においてテキスト中の単語の出現回数を集計する WordCount ワークロードでは 3.27 倍，2 単語の共起頻度を集計する BiGram Count ワークロードでは 3.51 倍の高速化をそれぞれ達成した．

1. はじめに

今日，学術界や産業界において大規模データの収集及びそれらを活用した価値創造が活発に行われており，対象とするデータのサイズは年々増加している．このような大規模データをクラスタコンピューティング環境を用いて高速に分析処理する手法として，従来のリレーショナルデータベースと SQL の組み合わせ以外の手法が数多く考案されてきた．中でも 2004 年に提案された MapReduce [1] と呼ばれるクラスタコンピューティング向けのプログラミングモデルは，そのモデルのシンプルさやスケーラビリティによって特に注目を集めている．MapReduce を利用することで 1 ノードでは処理しきれないペタバイト級のデータを，分散処理などの詳細な部分をユーザに意識させることなく処理することが可能となる．また MapReduce フレームワーク上に Logistic Regression や K-means に代表される各種機械学習アルゴリズムを実装できることも知られており [2], [3]，より柔軟な大規模データ分析を行うことができる．MapReduce のオープンソース実装として 2011 年には Apache Hadoop [4] が，2014 年には Apache Spark [5] がそれぞれリリースされた．特に in-memory 処理系である Spark は on-disk 処理系である Hadoop と比べて圧倒的な性能を示すことから，ここ数年学術界，産業界から大きな注目を集めている．MapReduce における性能上のボトルネックはノード間でデータの交換を行う shuffle と呼ばれるフェーズであることが知られており，これまで shuffle フェーズのコスト削減を図る研究が数多くなされてきた．これらの研究は既存処理系の実装改善 [6], [7], [8], [9]，新たなスキームの提案 [10], [11], [12] の 2 種類に大きく分類することができる．

このようにソフトウェアの分野では伝統的なデータ解析手法である SQL に囚われない，より柔軟な手法を提供するフレームワークの利用が広がる一方で，ハードウェアの分野でもアーキテクチャの大幅な変革が起ころうとしている．ネットワーク分野では，InfiniBand に代表される高性能インターコネクトや RDMA（Remote Direct Memory Access）などの例が挙げられる．これらは先端技術であり，現時点では大学や研究機関の保有するスーパコンピュータやクラスタシステムでの採用事例が主であるが，一般企業

[1] 筑波大学大学院 システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] daikoku@hpcs.cs.tsukuba.ac.jp

が保有するクラスタでもインターコネクトとして採用されるケースが徐々に増えており [13]，今後市場価格の下落に伴いより幅広く普及することが予想される．

これらの前提を踏まえ，本研究では高速インターコネクトである InfiniBand 及び RDMA を活用し，in-memory MapReduce における最適な shuffle 手法を，通信手段とアルゴリズムの 2 つの観点から検討及び評価を行う．通信手段は InfiniBand 上で IP 通信をエミュレートする IPoIB（IP over InfiniBand）と RDMA の 2 種類を，通信アルゴリズムは MPI（Message Passing Interface）における All-to-All 通信アルゴリズムの 1 つである *Pairwise* アルゴリズムに基づく方式，Hadoop/Spark 等で採用されている全ノードペア間で相互にリンクを張りデータ転送を行う *Fully-Connected* 方式の 2 手法を，それぞれ比較する．また上記の 2 要素が shuffle のパフォーマンスに与える影響を純粋に測定するために，独自の in-memory MapReduce 処理系を C/C++を用いて実装し，既存システムである Spark との比較を通して提案手法の性能評価を行う．

本論文の構成は以下の通りである．第 2 節では背景として MapReduce について，主な実装例と shuffle 性能改善に関する先行研究を交えて解説を行う．第 3 節では本研究で検討する shuffle 手法に関して，通信手段とアルゴリズムに焦点を当てて説明する．第 4 節では提案手法のプロトタイプ実装に関して，その設計と実装を詳細に説明する．第 5 節ではプロトタイプ実装の性能をクラスタ上で計測した結果を，Apache Spark との比較を交えて示す．第 6 節では本研究のまとめと今後の研究課題について考察を行う．

2. 背景

2.1 MapReduce

MapReduce [1] は 2004 年に提案されたクラスタコンピューティング向けのプログラミングモデルである．MapReduce の主な特徴はそのシンプルなモデルと，それに伴う優れたスケーラビリティ及び耐障害性である．

MapReduce ではユーザが *map* と *reduce* の 2 つの逐次処理を記述するだけで，システムがデータローカリティを考慮したうえでクラスタノードにタスクを割り当て，各ノードで処理を並列に実行する．MapReduce はユーザの立場から見ると非常にシンプルかつ明快なプログラミングモデルであるが，分散ソーティングやウェブインデックスの構築，Logistic Regression, Naive Bayes, K-means, Support Vector Machine, Neural Network に代表される各種機械学習アルゴリズムなど，非常に幅広い処理を記述できることがこれまでの研究で明らかになっている [2], [3]．

加えて map と reduce の個々のタスクは互いに独立しているため，容易にクラスタノードにスケールアウトでき，仮に実行途中に一部の処理が中断してしまってもその部分だけ再度処理することが可能である．特に数百〜数千ノード規模のシステムで処理を行う場合，数ノード規模のシステムと比べて処理の途中に一部のノードがダウンする確率が高いことから，そのような環境に適した手法であるといえる．

2.2 MapReduce の主な実装例

Google による最初の実装は C/C++ベースであった [1]．その後，有名なオープンソース実装である Java ベースの Apache Hadoop [4] や Scala ベースの Apache Spark [5] などが登場した．これらのソフトウェアはコモディティハードウェアを対象としており，リカバリ機構など耐障害性・可用性を重視した設計となっているため，産業界において広く利用されている．またこれらは SaaS（Software as a Service）として各種クラウドコンピューティングサービスでも提供されており，Spark の主なコントリビュータの 1 つである Databricks Inc. が行った調査によると，Spark ユーザの半数近くはパブリッククラウドサービス上で Spark を利用している [14]．

一方で大学や研究所が保有するハイパフォーマンスコンピューティング環境を対象とし，次世代ハードウェアを活用するための実験的な実装も存在する．先端ハードウェアとして shared-memory multiprocessor [15], [16] や GPU [17], [18], Xeon Phi [19], InfiniBand [8], [9] を活用した事例がある．その他にも MPI を通信フレームワークとして利用した MR-MPI（MapReduce-MPI Library）[20] や，京コンピュータなどのスーパコンピュータ上で MapReduce を実行する目的で開発された KMR（K MapReduce）[21]，分散 Key-Value Store をベースとした MapReduce 処理系である SSS [22] などが存在する．

2.3 Shuffle の高速化に関する関連研究

多くの場合 MapReduce フレームワークにおける性能上の最大のボトルネックは，ノード間でデータの交換を行う shuffle フェーズであることが知られている．shuffle フェーズのコスト要因としては主に以下の 2 点が挙げられる．

1. ノード間通信に要するネットワーク I/O コスト
2. key-value データのシリアライズ/デシリアライズに要する CPU コスト

これらに加え，Apache Spark の場合はさらに以下 2 点が問題点として知られている．

3. shuffle の前後にシリアライズされたデータを一度ローカルディスクに読み書きすることに起因するランダムアクセス
4. それぞれのノードが他のすべてのノードと相互に接続を確立する Fully-Connected 通信

これらのコストを抑え shuffle フェーズの高速化を目的とした研究がこれまで多くなされてきた．これらの研究はアプローチの面から以下の 2 つに分類できる．

既存実装の高速化

[6] では Spark の shuffle フェーズにおける性能ボトルネックが前述のディスクへのランダムアクセスにあることを検証により明らかにした上で，shuffle 前にディスクに書き込まれるファイル数を削減することで性能改善を図っている．しかし書き込む先のファイルシステムの特性によって性能が左右されるという問題があり，例としてファイルシステムに ext4 を採用した場合は従来の Spark と比較して最大 2 倍の性能改善が達成されたにも関わらず，ext3 を採用した場合は逆に性能が劣化するという結果になっている．

[8] では RDMA を通信方式として用いる Shuffle Server/Client を Spark 向けのプラグインとして実装し，1/10 Gig Ethernet 及び IPoIB との性能比較を行っている．RDMA の機能は JNI（Java Native Interface）を通して Spark 内部から呼び出している．Spark のオペレータの 1 つであり，入力データを key 毎にグルーピングする *groupByKey* を用いたベンチマークでは従来の Spark と比較して 80%近い高速化を達成している．しかし通信トポロジは依然として Fully-Connected であり，加えて前述のとおり Spark における shuffle ではネットワーク I/O ではなくディスクへのランダムアクセスが性能上のボトルネックであることから，より RDMA に適した shuffle 手法の検討，及び RDMA が shuffle の性能に与える影響の詳細な調査の余地が残されていると考える．

新たなスキームの提案

MaRCO [11] では shuffle フェーズを map/reduce フェーズとオーバラップさせることで shuffle に要する時間の隠蔽を図っている．さらに Hadoop に上記の仕組みを実装し，shuffle-heavy なワークロードにおいて 23%の高速化を達成している．現在の Hadoop の実装では map/shuffle/reduce の 3 フェーズを，Spark では map/shuffle の 2 フェーズを，それぞれオーバラップして実行している．

MR-MPI [20] や KMR [21] では MapReduce フレームワークと HPC 分野で従来から利用されてきた MPI の類似点に着目し，MPI を通信フレームワークとして採用した MapReduce 処理系を実装している．MPI には *MPI_Alltoall/Allgather/Bcast* といった集団通信ルーチンが定義されており，*Bruck*, *Log-step*, *Pairwise* といった Fully-Connected よりも効率的なアルゴリズムがこれまでに複数考案されている．上記の研究ではこれらを MapReduce の shuffle 通信パターンに応用している．

3. RDMA を利用した Shuffle 手法

本研究では InfiniBand や RDMA といった HPC 分野で広く使われている先端技術を活用し，in-memory MapReduce 処理系における最適な shuffle 手法を，通信手法とアルゴリズムの 2 つの観点から検討を行う．通信方式としては IPoIB と RDMA の 2 種類の比較を行う．通信アルゴリズムとしては Hadoop/Spark などの既存のシステムで広く採用されている方式である *Fully-Connected* 方式，MPI の All-to-All 通信，すなわち全プロセス間でのデータ交換で用いられる方式である *Pairwise* 方式の 2 種類の比較を行う．

3.1 通信方式

InfiniBand は従来の Ethernet と比べて高スループット，低レイテンシを実現する先端ネットワーク規格である．InfiniBand は HPC 分野でのデファクトスタンダードとなっており，スーパコンピュータの演算性能を評価するランキングである TOP500 にランクインするシステムの約半数がインターコネクトとして採用している [23]．InfiniBand は複数の通信モードをサポートしており，そのうちの 1 つである RDMA では CPU の介在なしにあるノードから別のノードのローカルメモリ上のデータを読み書きすることができる．さらにアプリケーションバッファ上のデータを OS バッファにコピーすることなく送信するため，データのコピーやコンテキストスイッチに要するコストを削減することができる．InfiniBand ではさらに IPoIB (IP over InfiniBand) と呼ばれる，InfiniBand 上で IP 通信をエミュレートするモードを提供している．ユーザは既存のコードを改変することなく，InfiniBand カードに割り当てられたアドレスを指定して通信を行うだけで IPoIB を利用することができる．

図 1 は第 5 節で示す評価実験の際に使用したクラスタ環境の 2 ノード間で Ping-Pong 通信を行い，1 Gig Ethernet/IPoIB/RDMA（rsocket）のネットワーク帯域幅を測定した結果である．メッセージサイズが大きくなるに連れて RDMA とその他の差が広がっており，メッセージサイズ 1 MB では RDMA は Ethernet の 46 倍高速であるという結果になった．IPoIB も Ethernet と比較すると高い性能を示しているが，InfiniBand の性能を最大限引き出すためには RDMA API を使用が望ましいといえる．

3.2 通信アルゴリズム

Fully-Connected 方式

この方式では 1 つのノードから他のすべてのノードにリンクを張り，データの転送を行うことで shuffle 時の通信を実現する．ノード数 n に対してリンクが全部で $(n-1)\times n$ 本必要になるというデメリットがある一方，各ノードで独立して通信を行えるため map/shuffle フェーズのオーバラップが容易に実現できるというメリットがある．Hadoop や Spark ではこの方式で shuffle 時の通信を実装している．

Pairwise 方式

Pairwise アルゴリズムは MPI における All-to-All 通信を実現するためのアルゴリズムの 1 つである．Pairwise アルゴリズムでは各ステップ毎にプロセス内でペアを作り，

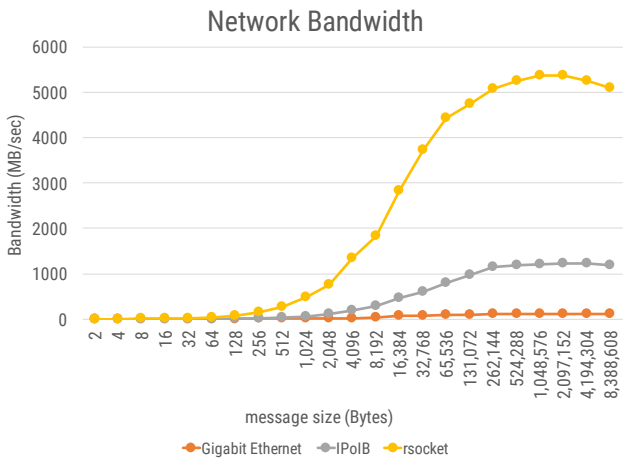

図 1 ネットワーク帯域幅測定結果

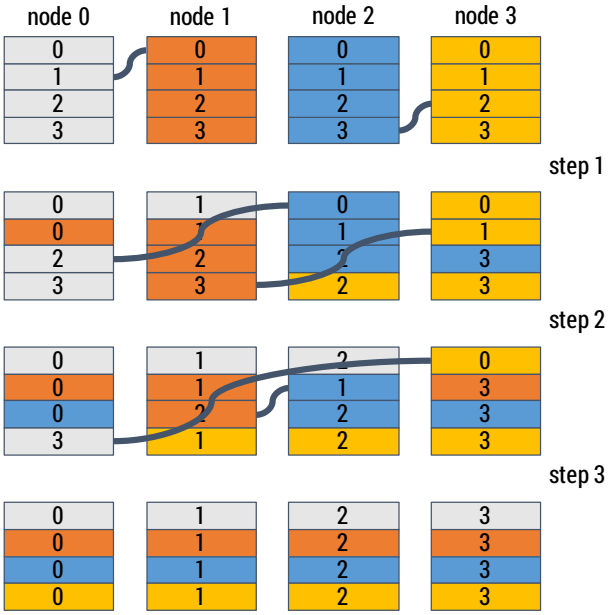

図 2 Pairwise All-to-All アルゴリズム

データの交換を行う．そのためプロセス数が 2 の冪乗の時に有効な手法である．また Pairwise アルゴリズムの特徴として宛先となるプロセスに直接データを転送するため，メッセージサイズが大きい場合にコスト面で有利となる，RDMA 通信を使った実装に適しているといったメリットが有る．MPI の主要な実装の 1 つである MPICH ではプロセス数が 2 の冪乗かつメッセージサイズが大きい場合の All-to-All 通信アルゴリズムとして Pairwise アルゴリズムが採用されている [24]．

図 2 は初め各ノードが持っている 0, 1, 2, 3 の 4 種類のデータが，Pairwise アルゴリズムによってそれぞれ node 0, 1, 2, 3 に集約される様子を表している．一般に通信に参加するプロセス数が n の場合，Pairwise アルゴリズムでは通信を完了するのに $n-1$ ステップを要する．図 2 の場合は，プロセス数が 4 であるため 3 ステップで通信が完了している．また図より，step 1 では 1 つ隣，step 2 では 2 つ隣，step 3 では 3 つ隣のプロセス同士でペアを作りデータの交換を行っていることが分かる．これを一般化すると，「step k において，rank i のプロセスは rank (i XOR k) のプロセスと通信を行う」となる．前述のとおり Pairwise アルゴリズムはプロセス数が 2 の冪乗の時のみ有効なアルゴリズムであるが，「step k において，rank i のプロセスは rank $(i-k)\%n$ のプロセスからデータを受け取り，rank $(i+k)\%n$ のプロセスへデータを送る」とすると 2 の冪乗以外の場合でも All-to-All 通信を実現できる．

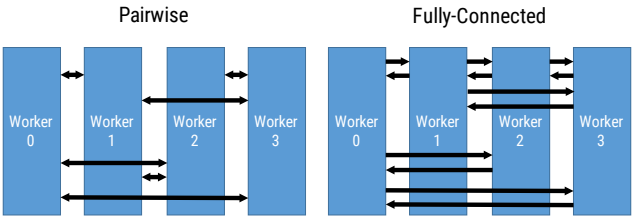

図 3 Pairwise/Fully-Connected 方式におけるリンク構築の概観

Pairwise アルゴリズムの長所として 通信完了までに張られるリンク数が少なく済むことが挙げられる．各ステップにおいて張られるリンク数はプロセス数 n に対して $n/2$ であるから，$n-1$ ステップでは $(n-1)\times(n/2)$ となる．前項で解説した Fully-Connected 方式の場合は $(n-1)\times n$ であるから，半分のリンク数で等価な通信を実現できることになる．プロセス数が 4 の時の Pairwise 及び Fully-Connected のリンクの概観図を図 3 に示す．Pairwise では $(4-1)\times(4/2)=6$ 本, Fully-Connected では $(4-1)\times 4=12$ 本のリンクがそれぞれ張られていることがわかる．

一方で短所としては，複数回の All-to-All 通信を行うと step 数の増加に伴い通信コストが増大することが挙げられる．前述のとおり既存のシステムでは map/shuffle フェーズのオーバラップが一般的に行われているが，Pairwise で同様の仕組みを実装すべく一連のステップを回し続けると，交換可能なデータが無いにもかかわらず通信が発生する「空回り」によって性能が劣化することが危惧されるため，現実的な方法であるとはいえない．

4. 設計と実装

本節では前節で示した提案手法のプロトタイプ実装について解説する．本プロトタイプは Apache Spark の核である RDD（Resilient Distributed Datasets） [25] のデザインを参考に実装された．実装の詳細を表 1 に示す．またプログラム構成の概観図を図 4 に示す．以下の項では Master/Worker プログラムについて，図中に示されているそれぞれの内部モジュールの設計と実装を解説する．

4.1 Master

RPC Client: Master ノードと Worker ノードの通信を

表 1　プロトタイプ実装環境

言語	C/C++
行数	3302
動作確認コンパイラ	clang 7.0.2, gcc/g++ 5.3.0, Intel®Compiler 16.0.1
依存ライブラリ	MessagePack 0.5.9, jubatus-mpio 0.4.5, jubatus-msgpack-rpc 0.4.4, Intel®TBB (Threading Building Blocks) 4.4-20150728, google-sparsehash 2.0.3, rdmacm 1.0.21mlnx-OFED.3.0.1.5.2

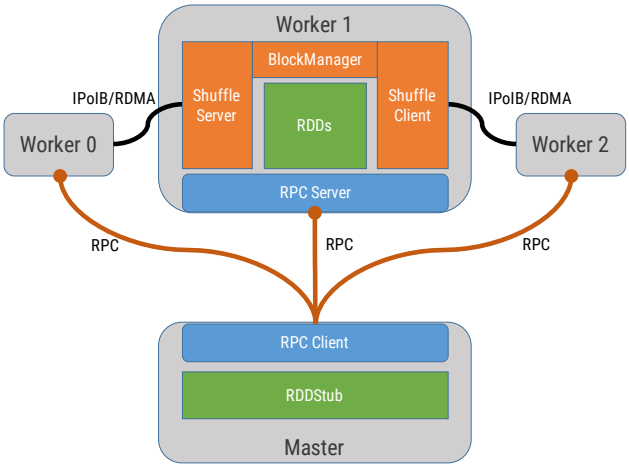

図 4　プロトタイプ概観

つかさどる．プログラム開始時に指定された設定ファイルから Worker ノードのホスト名及びポート番号を読み込み，接続を試みる．また MapReduce の入力データとしてテキストファイルを用いる場合，決められたパーティションサイズ（デフォルト: 128 MiB）に応じてファイルのインデックスを作成し，RPC を介して Worker ノードに送信する．Worker ノードでは受信したインデックスに基いて実際にファイルの read を行う．

RDDStub: *Map* や *Reduce* などの RDD に対する各種オペレータをスタブとして定義されており，さらに対応する RDD を所有する Worker ノードの ID を管理している．アプリケーションが RPCStub で定義されているオペレータを呼ぶと，RPC Client を介して Worker ノードで実際の処理が実行される．

4.2　Worker

RPC Server: Master ノードからの RPC を受け取り，指定された RDD に対してオペレータの適用を行う．

RDD: Apache Spark の RDD の一部の機能を実装している．本プロトタイプでは 1 つの key に対して 1 つの value がペアになっている *KeyValueRDD* と 1 つの key に対して複数の value がペアになっている *KeyValuesRDD* の 2 種類が用意されている．前者には *Map* が，後者には *Reduce* と *GroupBy* がそれぞれオペレータとして定義されている．Map や Reduce の実際の処理ルーチンは共有ライブラリとしてコンパイルし，モジュール化される．ユーザはそれらを NFS などに配置し，Master プログラムからその path を RPC を介して Worker プログラムに送信すると，実行時にダイナミックリンクされる．また Map と Reduce の処理は RDD のパーティション単位でマルチスレッド処理され，最大でシステムが持つ CPU コア数と同数のスレッドが並列に処理を行う．マルチスレッド化には Intel®TBB ライブラリを使用した．

BlockManager: shuffle フェーズの前後の中間データを管理する．内部では Intel®TBB ライブラリの提供するスレッドセーフな queue を RDD のパーティションの数だけ保持しており，複数スレッドからの push/pop に対応している．shuffle の開始前に RDD が管理している key-values のペアは MessagePack ライブラリを使用してシリアライズされ，本モジュールに push される．この際 key のハッシュ値をパーティション数で割った剰余を用いて push する queue を指定する．続いて後述の Shuffle Server/Client が本モジュールよりシリアライズされデータを pop して宛先に送信し，受信したデータを push する．reduce フェーズでは本モジュールからデータを pop し，デシリアライズした上でそれ以降の処理を進める．

4.2.1　Shuffle Server/Client

Fully-Connected 方式

Fully-Connected 方式では map フェーズの開始前に Shuffle Server/Client 用のスレッドを 1 つずつ起動する．Client は他の Worker ノードの Shuffle Server にブロックフェッチのリクエストを送り，Server はリクエストを受け取ると BlockManager からその Client が要求したパーティションのブロックを pop して送信する．すべての map タスクが終了した時点で BlockManager は終了状態にセットされ，Server 側は BlockManager が終了状態かつ Client が要求したパーティションの queue が空になると，その Client に対し終了通知を送る．Client 側は自身が担当する全てのパーティションに対して終了通知を受け取るとスレッドを終了し，Server 側はすべての Client からの接続が切断されるとスレッドを終了する．

実装に関しては Server/Client ともに non-blocking I/O と *poll* システムコールの組み合わせによって，複数ノードとの通信をそれぞれ 1 スレッドで実現している．また通信手段として一般的な “BSD socket API” を用いるものと，rdmacm ライブラリの提供する “rsocket API” を用いるものの 2 種類を実装した．前者は通信時に使用するアドレスにマシンの InfiniBand カードに割り振られているアドレスを指定することで IPoIB を利用することができる．後者は RDMA Write 通信を BSD socket API-like にラップし

表 2　実験環境

# of nodes	32
OS	CentOS release 6.5 (Final)
Kernel	Linux 2.6.32-431.el6.x86_64
Apache Hadoop	2.6.2
Apache Spark	1.5.2 (Standalone Deploy Mode)
CPU	Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz x 2
# of CPU cores	10 x 2
Memory	64 GB
InfiniBand	Mellanox Technologies MT27500 Family [ConnectX-3]

たもので，利用する構造体は異なるものの BSD socket とほぼ同等の流れで Server/Client プログラムを記述することができる．

Pairwise 方式

Pairwise 方式では Master ノードからの RPC に応じて Shuffle Server/Client を起動してペアとなるノードと接続し，互いにデータを交換した後で Master ノードへ終了を通知する．Master 側では全ペアの終了通知を受け取った後，次のステップへと処理を進める．また通信手段として Fully-Connected と同様に “BSD socket API” を用いるものと，“rsocket API” を用いるものの 2 種類を実装した．

前節で述べたとおり，Pairwise 方式の map/shuffle フェーズのオーバラップは空回りによる通信コストの増加が懸念されることから，本実装ではオーバラップさせずに全ての Mapper が処理を完了してから shuffle フェーズを開始する．

5. 評価実験

本節では前節で紹介したプロトタイプ実装の評価を行うために実施した 3 種類の性能評価実験（GroupBy, Word Count, BiGram Count）について，その概要と結果，及び考察を示す．各実験で用いた処理対象のテキストファイルは，Hadoop パッケージに含まれるサンプルプログラム集の中の “RandomTextWriter” [26] を使用して生成した．このプログラムは Unix 環境で提供されている辞書ファイル /usr/share/dict/words の中からあらかじめ選んだ 1000 種類の英単語を，指定されたバイト数に達するまでランダムに書き込んだテキストファイルを出力する．

実験環境のクラスタ構成を表 2 に示す．

5.1 実験 1: GroupBy

この実験ではテキストファイル中の単語をその文字数に応じてグループ分けを行う GroupBy ワークロードで 4 つの shuffle 手法の性能を評価する．このワークロードでは入力データ全体が shuffle の対象となるため，後述の 2 つのワークロードと比べて転送データ量及び shuffle フェーズが占める時間が多くなり，IPoIB と RDMA の性能差を明らかにするのに適している．本実験では 2, 4, 8, 16, 32 GiB の 5 種類をテキストファイルを用い，Fully-Connected（IPoIB），Fully-Connected（RDMA），Pairwise（IPoIB），Pairwise（RDMA）の 4 種類の shuffle 手法の実行時間を計測する．また使用するノード数は 2 GiB の場合のみ 16 ノード，それ以外は 32 ノードとする．これはいずれのシステムも 128 MiB 毎に 1 つの Map タスクがスケジュールされるようになっており，2 GiB の場合は 2048/128 = 16 個のみタスクが実行されるためである．

```
void
Map(dense_hash_map<int, vector<string>> &kvs,
        const long long int &key,
        const string &val) {
  size_t cur = 0, pos;
  while ((pos = val.find_first_of(' ', cur))
        != val.npos) {
    auto word = val.substr(cur, pos - cur);
    kvs[word.length()].push_back(word);
    cur = pos + 1;
  }
  auto word = val.substr(cur, val.length() - cur);
  kvs[word.length()].push_back(word);
}
```

図 5　GroupBy - Mapper コード例

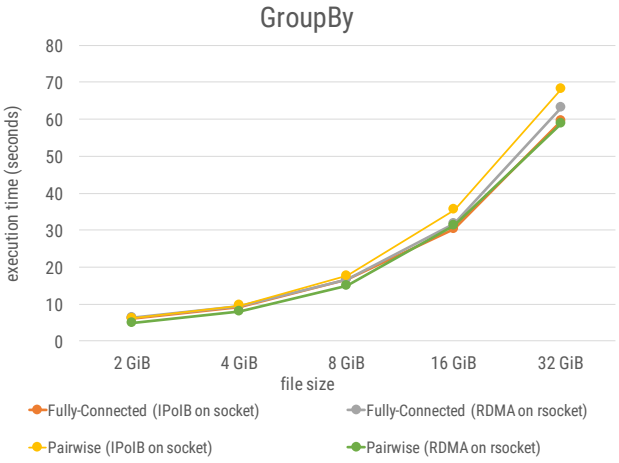

図 6　GroupBy - ファイルサイズ別実行時間

この実験で用いたプロトタイプ実装用の Mapper コード例を図 5 に示す．

5.1.1 実験結果

評価対象の 4 システムに関して GroupBy の実行時間を計測した結果が図 6 である．おおむね Pairwise（IPoIB），Fully-Connected（RDMA），Pairwise（RDMA），Fully-Connected（IPoIB）の順に高い性能を示しており，最も高速な Fully-Connected（IPoIB）と次点の Pairwise（RDMA）との性能差は数%程度であった．

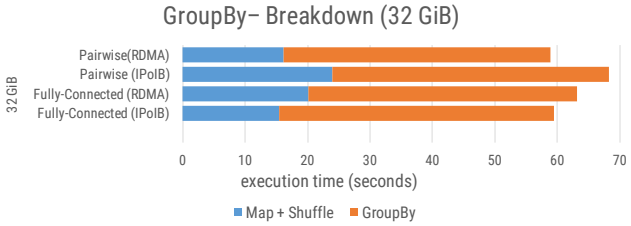

図 7 GroupBy - ファイルサイズ別実行時間内訳

```
void
Map(dense_hash_map<string, vector<int>> &kvs,
    const long long int &key, const string &val) {
  size_t cur = 0, pos;
  while ((pos = val.find_first_of(' ', cur))
         != val.npos) {
    kvs[val.substr(cur, pos - cur)].emplace_back(1);
    cur = pos + 1;
  }
  kvs[val.substr(cur, val.size() - cur)]
    .emplace_back(1);
}
```

図 8 Word Count - Mapper コード例

```
pair<string, int>
Reduce(const string &key, const vector<int> &vals) {
  return make_pair(
    key, accumulate(vals.begin(), vals.end(), 0)
  );
}
```

図 9 Word Count - Reducer コード例

32 GiB の時の処理時間の内訳を図 7 に示す．Pairwise (IPoIB) と Pairwise (RDMA) の両者を比較すると，RDMA が *map + shuffle* フェーズにおいて 1.49 倍高速であることが確認できる．一方で Fully-Connected（IPoIB）と Fully-Connected（RDMA）を比較すると，RDMA の性能が劣るという結果となった．

5.2 実験 2: Word Count

5.2.1 実験内容

この実験ではテキストファイル中の単語の出現頻度の集計を行う Word Count ワークロードで性能を評価する．本実験では 2, 4, 8, 16, 32, 64, 128, 256, 512 GiB の 9 種類をテキストファイルを用い，2 GiB の場合は 16 ノード，それ以外は 32 ノードで評価する．加えていずれのシステムも *in-mapper combining* と呼ばれる，shuffle フェーズの前に各 Mapper が自身の出力に対して部分的に reduce 処理を行う手法を適用している．

この実験で用いたプロトタイプ実装用の Mapper/Reducer（Combiner）のコード例を，図 8 と 9 にそれぞれ示す．

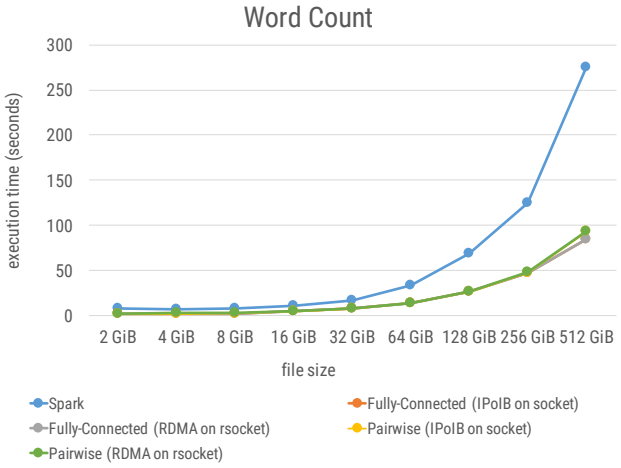

図 10 Word Count - ファイルサイズ別実行時間

5.2.2 実験結果

評価対象の 5 システムに関して Word Count の実行時間を計測した結果が図 10 である．Spark と提案 4 手法を比較するとファイルサイズの増加に伴い両者の差が大きくなり，512 GiB の場合の Fully-Connected（RDMA）を Spark と比較すると 3.27 倍高速となっている．また 4 つの shuffle 手法のグラフの重なり具合から，それらの間では性能に大差がないことがわかる．これは Word Count ワークロードの特性によるもので，Combiner によって Mapper の出力が大幅に集約され shuffle フェーズが占める時間が小さくなり，shuffle 手法の違いによる差が現れにくいためである．一般に Word Count においてテキスト中に出現する単語の種類を k とすると，key の種類も k となることから，Map タスクの数を m とすると Combiner によって Mapper の出力は，$m \times k \times (pair\_of\_one\_string\_and\_one\_int)$ 程度に集約される．

2, 32, 512 GiB の時の処理時間の内訳を図 11 に示す．512 GiB の場合の提案 4 手法の *map + shuffle* に着目すると，前述のような理由で通信手段の違い（IPoIB，RDMA）による差はほとんど見られないが，通信アルゴリズムの違いでは Fully-Connected が 10.8%（10 秒）程度高い性能を示している．これは 512 GiB の時，各ノードではコア数 20 に対し 524288/128/32 = 128 個の map タスクが実行されるため，map と shuffle のオーバラップによって完了した map タスクの出力を随時 Reducer に転送可能な Fully-Connected の優位性が現れた結果である．

5.3 実験 3: BiGram Count

5.3.1 実験内容

この実験ではテキストファイル中に登場する単語の共起頻度の集計を行う BiGram Count ワークロードで性能を評価する．具体的な処理内容はテキストを先頭から 2 単語ずつ読み込み，それらのペアを key として Word Count と同

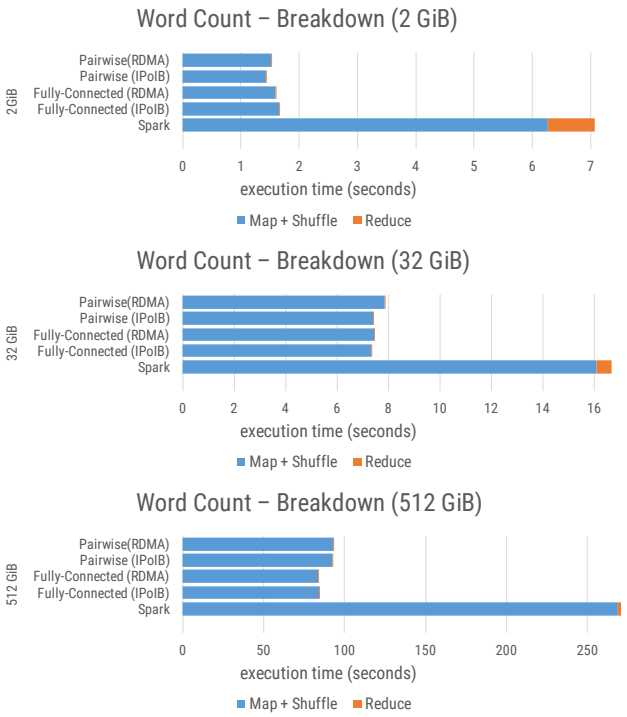

図 11 Word Count - ファイルサイズ別実行時間内訳

様の処理を行う．WordCount と同じく 2 GiB のテキストファイルに関しては 16 ノードで，それ以外は 32 ノードで計測を行い，それぞれのシステムで *in-mapper combining* を適用している．

BiGram Count では Word Count と比べて key のサイズが大きくなるだけでなく，key の種類が増え，それぞれの key の出現頻度が少なくなることから Combiner による Mapper 側での集約効果が小さくなり，結果としてより shuffle/reduce-heavy なワークロードとなる．BiGram Count においてテキスト中に出現する単語の種類を k とすると，key の種類，すなわち 2 単語の組み合わせは最大 $k \times k$ 個であるから，Map タスクの数を m とすると Combiner によって Mapper の出力は，$m \times k \times k \times (pair\_of\_two\_words\_and\_one\_int)$ となる．

この実験で用いたプロトタイプ実装用の Mapper/Reducer（Combiner）のコード例を，図 12 と 13 にそれぞれ示す．

5.3.2 実験結果

評価対象の 5 システムに関して BiGram Count の実行時間を計測した結果を図 14 に示す．4 種類のプロトタイプの結果に着目すると，ファイルサイズの増加に伴い Word Count の結果よりもはっきりとした性能差が見られる．これは前述のとおり shuffle 時にやり取りするデータ量が Word Count と比較して多いことに起因するものであり，概ね Fully-Connected（RDMA），Pairwise（RDMA），Fully-Connected（IPoIB），Pairwise（IPoIB）の順に高い性能を示している．256 GiB の場合のみ Pairwise の性能に

```
void
Map(dense_hash_map<string, vector<int>> &kvs,
    const long long int &key,
    const string &value) {
  size_t cur = 0, p1, p2 = 0;
  p1 = value.find_first_of(' ', cur);
  while (true) {
    p2 = value.find_first_of(' ', p1 + 1);
    if (p2 == value.npos) {
      kvs[value.substr(cur, value.size() - cur)]
        .emplace_back(1);
      break;
    }
    kvs[value.substr(cur, p2 - cur)]
      .emplace_back(1);
    cur = p1 + 1;
    p1 = p2;
  }
}
```

図 12 BiGram Count - Mapper コード例

```
pair<string, int>
Reduce(const string &key,
       const vector<int> &values) {
  return make_pair(
    key,
    accumulate(values.begin(), values.end(), 0)
  );
}
```

図 13 BiGram Count - Reducer（Combiner）コード例

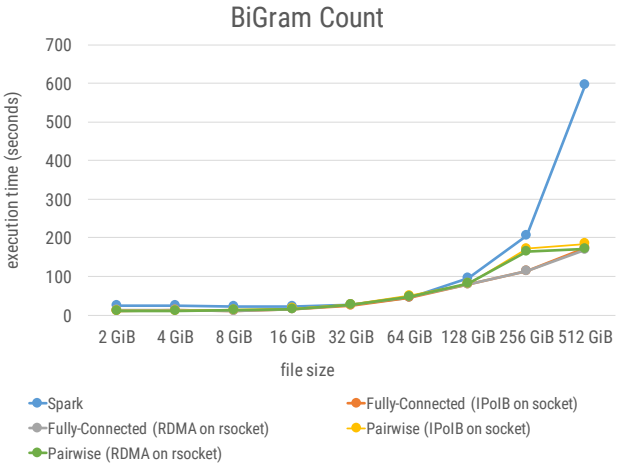

図 14 BiGram Count - ファイルサイズ別実行時間

劣化が見られるが，これは Reducer に対する key の分配に偏りが生じ，ある特定のペア間の通信データ量が大きく，時間がかかったためである．

また Spark との比較では 64 GiB の場合を除いて高い性能を示しており，512 GiB の場合の Fully-Connected（RDMA）は 3.51 倍高速となっている．Spark は 512 GiB の時に性能が急激に劣化しているが，この理由については後述する．

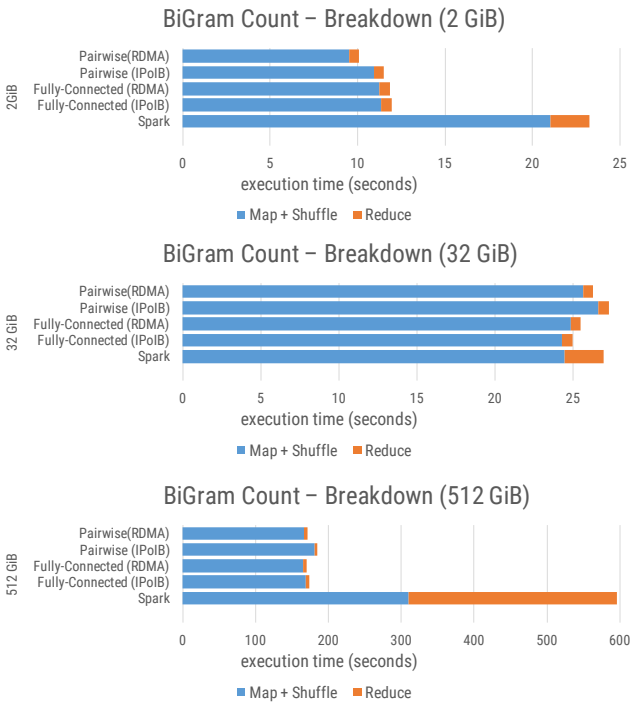

図 15 BiGram Count - ファイルサイズ別実行時間内訳

2, 32, 512 GiB の時の処理時間の内訳を示したものが図 15 である．512 GiB の時の Fully-Connected（RDMA）は Fully-Connected（IPoIB）と比べて 2.10%の性能向上が見られるのに対し，Pairwise（RDMA）を Pairwise（IPoIB）と比較すると 8.46%と，より大幅な性能向上が見られる．Pairwise は全ての map タスクが終了した後にその出力をまとめて転送するのに対し，Fully-Connected では完了した map タスクの出力を随時転送するため 1 転送あたりのデータ量は比較的少なくなる．また図 1 で示したとおり，データサイズの増加に伴い IPoIB と RDMA の帯域幅の差が大きくなる．以上の点より，Fully-Connected よりも Pairwise の方が RDMA 適用による性能改善幅が大きく，RDMA に適したアルゴリズムであるといえる．

また Fully-Connected（IPoIB）を Pairwise（IPoIB）と比較すると 6.83%の性能改善が見られ，この実験結果からも map/shuffle フェーズのオーバラップを行う Fully-Connected の優位性を確認できる．

図 14 において Spark は 512 GiB の場合に性能劣化が見られたが，その内訳では reduce が全実行時間の半分近くを占めていることが分かる．この理由は図 16 に示す Spark の reduce フェーズの実行ログを可視化した図から推察することができる．shuffle されたデータをディスクから読み込む *Shuffle Read* に reduce フェーズの大半が費やされており，このことが Spark の性能が 512 GiB の時に急激に劣化している原因であると考えられる．よって shuffle フェーズにおいてデータをローカルディスクへ書き出さず，メモリ上に保持する独自処理系の優位性が顕著に現れる結果となった．

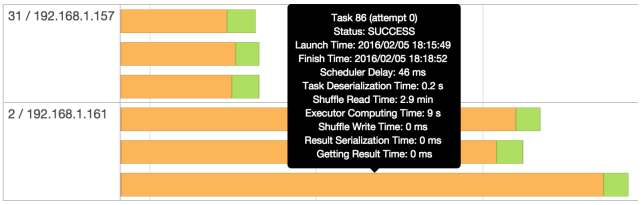

図 16 512 GiB の時の Spark の reduceByKey 処理時間内訳

6. 結論と今後の研究課題

6.1 結論

本研究では先端ネットワーク技術である InfiniBand と RDMA を活用し，RDMA と IPoIB の 2 種類の通信手段，Fully-Connected と Pairwise の 2 種類のアルゴリズムの合計 4 通りの shuffle 手法に関して検討を行った．またこれらの手法の評価のために独自 in-memory MapReduce 処理系を C/C++を用いて実装した．

shuffle-heavy な GroupBy ワークロードでの評価実験では，RDMA の適用により 1.46 倍の高速化を達成した．同じ in-memory 処理系である Apache Spark との比較では，Word Count ワークロードにおいては最大 3.27 倍，BiGram Count ワークロードにおいては最大 3.51 倍の高速化を達成した．また提案 4 手法の比較では map/shuffle フェーズのオーバラップを行う Fully-Connected が最大 10.8%高い性能を示したものの，RDMA 適用による性能改善幅は Pairwise の方が大きく，より RDMA に適したアルゴリズムであるという結論に至った．

6.2 今後の研究課題

前述のとおり Fully-Connected は多くの場合 Pairwise よりも高い性能を示すが，RDMA の適用により性能改善を図るという観点では Pairwise の方が適したアルゴリズムであることが明らかになった．しかし第 3 節で述べたとおり，Pairwise 方式のナイーブなオーバラップ化では交換可能なデータがないにも関わらず通信が発生する「空回り」や，1 転送あたりのデータサイズの減少によって性能が劣化することが懸念される．したがって RDMA-aware かつ map フェーズとのオーバラップに適した shuffle 手法の考案が課題となる．RDMA はその特徴として通信に要する CPU コストが少なく済むため，上記のような要件を満たす shuffle 方式を実現することでより多くの CPU リソースを map 処理に回すことができ，map 処理自体の性能向上も期待できる．

謝辞

本研究の一部は，JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェ

ア」，JST CREST「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」による．

参考文献

[1] Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *Communications of the ACM*, Vol. 51, No. 1, pp. 107–113 (2008).
[2] Chu, C., Kim, S. K., Lin, Y., Yu, Y., Bradski, G., Ng, A. Y. and Olukotun, K.: Map-reduce for machine learning on multicore, *Annual Conference on Neural Information Processing Systems*, MIT Press, pp. 281–288 (2006).
[3] Zhao, W., Ma, H. and He, Q.: Parallel k-means clustering based on mapreduce, *International Conference on Cloud computing*, Springer, pp. 674–679 (2009).
[4] Apache: Apache Hadoop, http://hadoop.apache.org/.
[5] Apache: Apache Spark, http://spark.apache.org/.
[6] Davidson, A. and Or, A.: Optimizing shuffle performance in spark, *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep* (2013).
[7] Guo, Y., Rao, J. and Zhou, X.: iShuffle: Improving Hadoop Performance with Shuffle-on-Write, *International Conference on Autonomic Computing*, USENIX, pp. 107–117 (2013).
[8] Lu, X., Rahman, M. W., Islam, N., Shankar, D. and Panda, D. K.: Accelerating spark with RDMA for big data processing: Early experiences, *Annual Symposium on High-Performance Interconnects (HOTI)*, IEEE, pp. 9–16 (2014).
[9] Wasi-ur-Rahman, M., Islam, N. S., Lu, X., Jose, J., Subramoni, H., Wang, H. and Panda, D. K.: High-performance rdma-based design of hadoop mapreduce over infiniband, *International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IEEE, pp. 1908–1917 (2013).
[10] Lin, M., Zhang, L., Wierman, A. and Tan, J.: Joint optimization of overlapping phases in MapReduce, *Performance Evaluation*, Vol. 70, No. 10, pp. 720–735 (2013).
[11] Ahmad, F., Lee, S., Thottethodi, M. and Vijaykumar, T.: MapReduce with communication overlap (MaRCO), *Journal of Parallel and Distributed Computing*, Vol. 73, No. 5, pp. 608–620 (2013).
[12] Ullman, J.: MapReduce Algorithms, *Proceedings of the 2Nd IKDD Conference on Data Sciences*, CODS-IKDD '15, New York, NY, USA, ACM, pp. 1:1–1:1 (online), DOI: 10.1145/2778865.2778866 (2015).
[13] さくらインターネット株式会社：さくらインターネット、演算に特化した「高火力コンピューティング」への取り組みを開始〜Infiniband 接続による大規模な GPU クラスタを Preferred Networks 社と共同構築〜，さくらインターネット株式会社（オンライン），入手先 ⟨http://www.sakura.ad.jp/press/2016/0126_gpu/⟩（参照 2016-01-26）．
[14] Databricks: Spark Survey 2015 Infographic, http://cdn2.hubspot.net/hubfs/438089/DataBricks_Surveys_-_Content/Spark-Survey-2015-Infographic.pdf?t=1443057549926.
[15] Talbot, J., Yoo, R. M. and Kozyrakis, C.: Phoenix++: modular MapReduce for shared-memory systems, *Proceedings of the second international workshop on MapReduce and its applications*, ACM, pp. 9–16 (2011).
[16] Chen, R. and Chen, H.: Tiled-MapReduce: Efficient and Flexible MapReduce Processing on Multicore with Tiling, *ACM Trans. Archit. Code Optim.*, Vol. 10, No. 1, pp. 3:1–3:30 (online), DOI: 10.1145/2445572.2445575 (2013).
[17] He, B., Fang, W., Luo, Q., Govindaraju, N. K. and Wang, T.: Mars: a MapReduce framework on graphics processors, *International Conference on Parallel Architectures and Compilation Techniques*, ACM, pp. 260–269 (2008).
[18] Chen, L. and Agrawal, G.: Optimizing MapReduce for GPUs with Effective Shared Memory Usage, *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, New York, NY, USA, ACM, pp. 199–210 (online), DOI: 10.1145/2287076.2287109 (2012).
[19] Lu, M., Liang, Y., Huynh, H. P., Ong, Z., He, B. and Goh, R. S. M.: MrPhi: An Optimized MapReduce Framework on Intel Xeon Phi Coprocessors, *IEEE Transactions on Parallel and Distributed. Systems*, Vol. 26, No. 11, pp. 3066–3078 (2015).
[20] Plimpton, S. J. and Devine, K. D.: MapReduce in MPI for Large-scale Graph Algorithms, *Parallel Computing*, Vol. 37, No. 9, pp. 610–632 (2011).
[21] Matsuda, M., Maruyama, N. and Takizawa, S.: K MapReduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers, *International Conference on Cluster Computing*, IEEE, pp. 23–27 (2013).
[22] Ogawa, H., Nakada, H., Takano, R. and Kudoh, T.: SSS: An Implementation of Key-Value Store Based MapReduce Framework, *International Conference on Cloud Computing Technology and Science*, pp. 754–761 (2010).
[23] TOP500.org: List Statistics, http://www.top500.org/statistics/list/.
[24] MPICH: alltoall.c, https://trac.mpich.org/projects/mpich/browser/src/mpi/coll/alltoall.c.
[25] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, *Symposium on Networked Systems Design and Implementation*, USENIX, pp. 15–28 (2012).
[26] Apache: RandomTextWriter.java, https://github.com/apache/hadoop/blob/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/RandomTextWriter.java.