高速な分散処理フレームワークの実現に向けた性能解析と高速な可変長マージソートの提案

小沢 健史[1,a)]　山室 健[1,b)]　内山 寛之[1,c)]　鬼塚 真[,d)]　岩村 相哲[1,e)]

概要：MapReduce，Apache Spark，Apache Tez といった分散処理フレームワークでは，巨大なデータを効率的に処理するために外部ソートが用いられている．本稿では，代表的な分散処理フレームワークである Apache Tez を対象に性能解析を行い，IO 帯域が十分に確保され，搭載メモリ量が十分な構成の計算機において，IO が支配的なワークロードとされている TeraSort においても，外部ソートを実行する箇所が CPU ボトルネックになり，性能が飽和することを示す．その問題の解決のために， CPU 最適化されたソートアルゴリズムであり，分散処理フレームワークに組み込み可能な key-prefix bitonic merge sort を提案する．提案手法は，従来手法と異なり，可変長のキーに対しても適用可能であるという特徴を有する．提案手法の部分実装は Hadoop に用いられる quick sort よりも 10 倍高速に動作し，分散処理フレームワークである Apache Tez の性能を最大で 2 倍改善できる見込みを示す．

キーワード：MapReduce, Hadoop, 分散処理

Performance analysis and fast merge sort for variable-length keys on distributed processing framework

Ozawa Tsuyoshi[1,a)]　Takeshi Yamamuro[1,b)]　Uchiyama Hiroyuki[1,c)]　Onizuoka Makoto[,d)]
Iwamura Sotetsu[1,e)]

Abstract: In this paper, we analyse performacne of TeraSort, an important and IO-dominant workload for general distributed processing framework. Surprisingly, we show that the IO-dominant job on Apache Tez, a opensource general distributed processing framework, can have CPU bottleneck on modern hardwares. To solve the problem, we propose CPU-optimized sorting algorithms, key-prefix bitonic merge sort, for general distributed processing framework. Our proposal can be used not only against fixed-length keys, but also variable-length keys. The partial implementation of proposed sort algorithm improves 10 times faster than Hadoop's quick sort algorthm in microbenchmark.

Keywords: MapReduce, Hadoop, Distributed Processing

1. はじめに

蓄積された大量のデータを安く，実用的な速度で処理するために，安価な計算機クラスタを用いて効率的に分散計算を行うフレームワークに注目が集まっている．特に，MapReduce は最も普及している分散処理フレームワークの 1 つであり，Google, Facebook, Yahoo! をはじめとする多くの企業に利用されている [2], [4], [14], [15]．MapReduce

[1] 日本電信電話株式会社
NTT
[2] 大阪大学
Osaka University
[a)] ozawa.tsuyoshi@lab.ntt.co.jp
[b)] yamamuro.takeshi@lab.ntt.co.jp
[c)] uchiyama.hiroyuki@lab.ntt.co.jp
[d)] onizuka@ist.osaka-u.ac.jp
[e)] iwamura.sotesu@lab.ntt.co.jp

では，並列処理をおこなう Map 関数と，Map 関数の結果を集約する Reduce 関数を記述するだけで，計算機の故障や，計算機間の同期処理を意識することなく，大規模な分散処理を行うことができる [4].

MapReduce 処理系の中でも，Apache Hadoop は，幅広く利用されている MapReduce のオープンソース実装であり，Hortonworks や Cloudera，米 Yahoo! が主導で開発を進めている [1]．MapReduce 処理系の強みの一つとして，スキーマ定義のない大量のデータに対して高いスループットで処理を行うことができることが挙げられる．この特性により，従来の分散データベースでは処理しきれなかった量の生データを一箇所に集約しておき，正規化やソートなどの前処理を行った上で RDBMS にデータをインポートするといった ETL (Extract/Transform/Load) プロセスに用いられる利用事例も出てきている ([10], [17]).

MapReduce の処理系の特性を保持しつつ高速化を狙った主要な分散処理フレームワークに Apache Tez[13] と Apache Spark[19] がある，これらの処理系は，MapReduce の特性を引き継ぎつつも，Map 関数，Reduce 関数を記述する代わりにデータフローを表す有向非循環グラフ (Directed Acyclic Graph, DAG) を記述することで，分散ファイルシステムへの中間データの書き込みを除去し，単純な MapReduce よりも高速かつ効率良く処理を行うことができる．

これらの 分散処理フレームワークの性能特性はハードウェア層，OS 層，言語ランタイム層，ミドルウェア層に依存し，かつ計算機をまたがって並列に動作するため，これらの処理系における性能解析は困難である，Kay らは，特にこれらのフレームワークにおける並列性が原因で処理系におけるブロッキング時間に関するログを取得することで性能解析を試みている [12]．しかしながら，この手法においては CPU，ネットワーク，ディスク IO いずれかの箇所がボトルネックであるかについてのみ触れられており，詳細なボトルネック箇所の特定の方法およびその改善手法については触れられていない．

そこで本稿では，分散処理フレームワークである Apache Tez を題材として性能解析を行い，具体的なボトルネックの特定および解消に向けた提案を行う．具体的には，OS 層，言語ランタイム層をまたがったプロファイリングツールである perf および perf-map-agent を併用することで OS，言語ランタイム，ミドルウェアをまたがった性能解析を実施する．結果として，十分にチューニングが行われ，かつ IO 帯域が十分な計算機上においては，ディスク IO が処理の大部分を占めるワークロードである Terasort においても，シャッフル時におけるソートに関する処理 (オブジェクトのシリアライズとソートにおけるスワップ処理) が原因で CPU ボトルネックになることを示す．

判明した CPU ボトルネックを解消するため， 本稿では CPU 最適化されたソートアリゴリズムである key-prefix bitonic merge sort の提案を行う．マイクロベンチマークの結果から，その実装は Hadoop に用いられる quick sort よりも 10 倍高速に動作し，分散処理フレームワークの性能を改善できる見込みを示す．

2. 前提知識

2.1 MapReduce

図 1 を用いて，本研究のベースなっている MapReduce について説明する．MapReduce ジョブは Map フェーズと Reduce フェーズの 2 フェーズから構成される．Map フェーズでは入力データ（D）を読み込んで Key/Value ペア (K_1, V_1) を生成し，それを入力として各ペアに対してユーザが定義した Map 関数を実行し，新たな Key/Value ペアリスト (K_2, V_2) を中間データとして出力する．Reduce フェーズでは，まず中間データを Key 毎にグルーピング（Shuffle）する．そして各グループを入力としてユーザが定義した Reduce 関数を実行し，結果として Key/Value ペアリスト (K_3, V_3) を出力する．この一連の流れを式として表現すると，以下のようになる．

$D \rightarrow map(K_1, V_1) \rightarrow \{K_2, V_2\}$
$shuffle(\{K_2, V_2\}) \rightarrow \{K_2, \{V_2\}\}$
$reduce(K_2, \{V_2\}) \rightarrow (K_3, V_3)$

MapReduce 処理は複数のノード（計算機）をネットワークで相互接続したクラスタ内で行われる．クラスタはリソースの割り当ておよびタスクの割り当てを行うマスタノードと計算を行うワーカノードで構成される．クラスタ上には DFS（分散ファイルシステム）が構築され，入力データが格納される．MapReduce ジョブを開始すると，まずマスタが入力データをユーザの指定したサイズで分割し複数の InputSplit を生成する．次にマスタが InputSplit の数だけ MapTask を生成し，各ワーカに割り当てる．MapTask が割り当てられたワーカでは Mapper が起動され，ユーザが定義した Map 関数が実行されて，Key/Value 形式の中間データが出力される．中間データは同じ Key 値を持つものが一つのワーカに集まるようにネットワークを介して相互に移動（shuffle）される．マスタはユーザが定義した数の ReduceTask を生成し，各ワーカに割り当てる．ReduceTask が割り当てられたワーカでは Reducer が起動され，ユーザが定義した任意の Reduce 関数が実行されて，新たに Key/Value 形式の処理結果が DFS に出力される．

現在，MapReduce から派生したオープンソースの分散処理フレームワークは複数存在する．中でも主要なものは，Apache Hadoop に含まれている Hadoop/MapReduce，Apache Spark，Apache Tez である．いずれの分散処理フレームワークにおいても，Shuffle 時にメモリに収まらない規模のデータを効率的に処理するため，外部マージソートが採用されている．これらの分散処理フレームワークにお

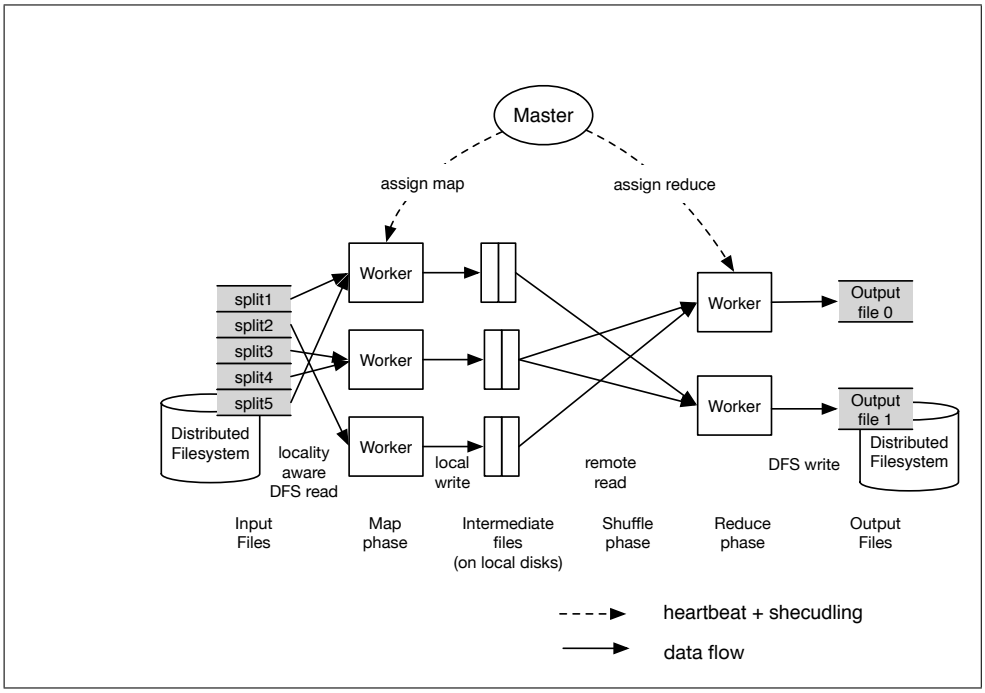

図 1 MapReduce の概要

ける，外部マージソート実行時の挙動は以下の通りである．まず，Shuffle 前に中間データをソートした上で外部記憶装置に書き込む．この際に実行される書き込み前のソートアルゴリズムには，Hadoop/MapReduce および Apache Tez ではクイックソートが，Apache Spark ではティムソートが利用されている．次に，書き込まれた中間データを Shuffle の受け取り側がフェッチし，マージすることでマージソートが完了する．

Apache Hadoop/MapReduce，Apache Spark，Apache Tez といった分散処理フレームワークにおいて，外部マージソートは巨大なデータの整形する際に必ず実行される処理である．例えば，これらの分散処理フレームワーク上において結合処理や集約処理を実行する場合においても，外部マージソートが内部的に実行される．

2.2 MapReduce 用 DSL(Domain Specific Language) と Apache Tez

MapReduce プログラムを正しくスケーラブルに設計するには，分割統治法を用いて注意深くアルゴリズム設計を行う必要がある．これらの設計を全てのプログラムについて行うのは非生産的であるため，Hive[16] や Pig[11] といった DSL 上を入力にとり，MapReduce 処理を出力するような MapReduce コンパイラが利用されることが一般的である．

DSL によるプログラム記述を前提をした場合, 必ずしもランタイムが MapReduce 処理系である必要はない．Apache Tez は，Dryad[7] をベースに作成された，MapReduce をさらに汎用化した分散処理フレームワークである．Apache Tez では，ユーザは Map 関数，Reduce 関数の変わりに有向非循環グラフ (Directed Acyclic Graph, DAG) を記述することで，冗長な分散ファイルシステムへの書き込みを除去し，単純な MapReduce よりも高速かつ効率良く処理を行うことができる．また，Hadoop MapReduce 用に作成された入力データのパースに用いられる InputFormat やタスク間で受け渡す Key，Value のデータ型 (Writable) およびその比較器 (Comparator) を変更なしに利用可能である．このように既存のインタフェースが利用可能になっていることで，既存の MapReduce アプリケーションが Tez 上で動作するだけでなく，Hive や Pig といった MapReduce 用 DSL を互換性を保ったまま動作させることが可能としている．

3. 性能解析

本章では，Apache Tez を対象に性能解析を行う．計算機にはマスタとして Amazon EC2 の m3.xlarge(E5-2670 v2 2.50GHz vCPU 4 コア，メモリ 15GB，ローカル SSD 40GB x 2) インスタンスを 1 台，ワーカとして i2.8xlarge (Xeon v 2.4 GHz vCPU 32 コア，メモリ 244GB，ローカル SSD 800GB x 8，10Gbps) インスタンスを 3 台を利用した．また，今回の性能解析の対象とするワークロードとして，IO が支配的なワークロードであり，応用先も広い TeraSort [18] を用いた．TeraSort の入力には TeraGen で用いたデータサイズ 240GB を用いた．また，ベンチマークの度にファイルキャッシュはクリアしている．TeraSort は，検索エンジンの構成するための転置インデックスの作成や，後段のジョイン処理の高速化などに応用可能なベン

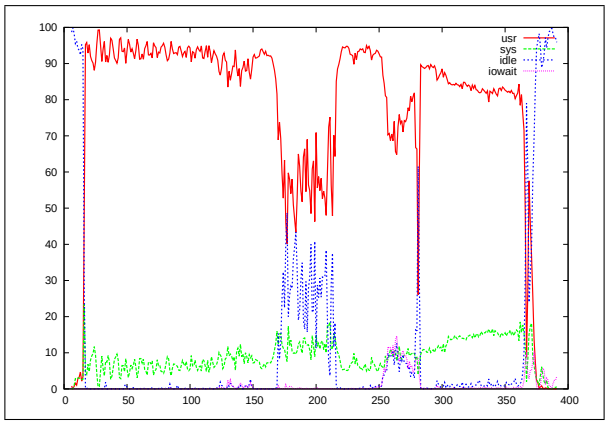

図 2 リソース利用率

チマークプログラムである．なお，TeraSort プログラムの実行には Apache Tez の機能である MapReduce job を変更なく実行するための機能を用いた．実験に利用したソフトウェアは Hadoop 2.7.1，Tez(0.8.0-SNAPSHOT)，OS は Linux 3.13.0-48 である．実験の再現性を高めるため，実験にあたり行った主要な設定を表 1 に示す．掲載されている値以外の値で，性能に影響のあるパラメータは変更していない．

性能解析には，Linux の perf と perf-map-agent を用いてプロファイリングを行う．perf を用いることにより，低オーバーヘッドで任意のプロセスのどの箇所がボトルネックになっているかを特定することができる．しかしながら，Java 仮想マシン (Java Virtual Machine, JVM) 内のどのメソッドに時間がかかっているかを特定することはできない．perf-map-agent を用いることで，Java のシンボル情報をメモリ空間とマッピングし，Java のどのメソッドでどの程度の時間がかかっているかを測定することができる[*1]．

プロファイリング結果のうち，CPU 利用率が高いトップ 10 をまとめたものを表 2 に，ジョブ実行中のリソース利用率を図 2 に示す．表 2 から，Tez のフレームワーク内におけるソートに関わる CPU の処理が， タスク内の 56.57 % に達することが判明した．また，図 2 から，ジョブ実行中に CPU 利用率 90% を超える時間が 70% を占めており，ジョブ実行時間の半分以上が CPU に費やされていることが判明した．つまり，IO が支配的なジョブと考えられている TeraSort においても，IO 帯域が十分かつメモリ量が十分な計算機クラスタ上において，Tez のボトルネックは CPU であるという結果が得られた．次章以降では，本章で示した CPU ボトルネックをどのように解消するかについて述べる．

[*1] ただし，一部の情報は JVM の JIT コンパイルやリフレクションによりそぎ落ちてしまう事がある

4. 分散処理フレークワークにおけるソートの高速化に向けた提案

2 章で述べた通り，MapReduce を初めとする分散処理フレームワークでは，クラスタに存在するメモリサイズよりも大きなデータを効率良く処理するために外部ソートを効率良く処理する機構が組み込まれている．3 章では，その箇所のうち，CPU で処理する箇所がボトルネックになることを示した．この箇所は，どのジョブにおいても通るパスであるため，該当箇所のボトルネックを解消することで，Hive や Pig 上で動作するクエリを含めた多くの MapReduce ジョブ全体の処理を高速化することが可能となる．しかしながら，MapReduce 処理系のソートは柔軟なテキスト処理を実現するために，高速化を行うにあたって多くの前提がある．本章では，その前提を明確化した上で，改善の方針を示す．

4.1 改善にあたっての前提条件

MapReduce 由来の分散処理フレームワークに組み込まれているソートの特徴として，可変長データをキーとしてソートができることが挙げられる．これにより，テキスト処理を初めとした柔軟な処理が可能になっている．よって，3 章で述べたボトルネックを解消するにあたっても，可変長データのソートを前提として高速化することが必要である．また，既存の MapReduce，Hive，Pig のクエリが無変更で動作するようにするために，可変長データの表現に用いられている構造体を再利用することができるようにすることも重要である．MapReduce，Hive，Pig においては，可変長データを表現するために Writable というインタフェースが用いられている．MapReduce の TeraSort のソートに用いられるキーは，Text クラスが用いられており，比較にはその内部で実装されている compare メソッドを用いる．また，Hive においては入力データは基本的に全て ByteWritable というバイト配列を表現するためのクラスが用いられており，比較にはその内部で実装されている compare メソッドを用いる．

4.2 改善の方針

本稿では，Hive における高速化を目的として，Text クラスおよび ByteWritable クラスを初めとしたバイト配列を内部に持つような可変長キーのソートに関する高速化を狙う．CPU 利用効率の高いソートとして，CPU の分岐予測ミスを減らす方法，CPU キャッシュ効率を高める方法，その組み合わせによりさらなる高速化を実現する方法などがある [3], [6], [9]．本稿では，Tez への組み込みやすさの観点から，キャッシュヒット効率が高い外部ソートアルゴリズムである AlphaSort で用いられている key-prefix sort [9] をベースに考える．key-prefix sort の概要を図 3 に示

表 1 実験に利用した設定

HDFS	dfs.client.read.shortcircuit	true	ローカルファイルの読み込み時に TCP ソケットの代わりに Unix Domain Socket を利用
YARN	yarn.nodemanager.resource.memory-mb	244GB	ワーカノードの利用可能メモリ
Tez	tez.task.resource.memory.mb	7168MB	1 タスク辺りが利用するメモリ
Tez	tez.runtime.io.sort.mb	3196	シャッフル時のソートに利用できるバッファサイズ
Tez	tez.am.container.reuse.enabled	true	タスクをまたがってコンテナを再利用
Tez	tez.runtime.pipelined.sorter.sort.threads	2	マルチスレッドのソートを利用
Linux	/sys/kernel/mm/transparent_hugepage/defrag	never	Transparent hugepage のデフラグを無効化.
Linux	/sys/kernel/mm/transparent_hugepage/enable	never	Transparent hugepage の機能自体を無効化.

表 2 TeraSort に対するプロファイリング結果

プロファイリング内で占める割合	コンポーネント名	ボトルネックの箇所
23.70%	Tez(ソート)	Lorg/apache/hadoop/util/QuickSort;.sortInternal[*2]
8.44%	Tez(ソート)	Lorg/apache/hadoop/io/Text$Comparator;.compare
8.35%	Tez(ソート)	Lorg/apache/hadoop/util/QuickSort;.sortInternal
8.04%	Tez(ソート)	Ljava/nio/IntBuffer;.put
8.04%	Tez(ソート)	Ljava/nio/IntBuffer;.get
6.28%	JVM	jbyte_disjoint_arraycopy
4.86%	Hadoop	Ljava/nio/ByteBufferAsIntBufferL;.get
3.00%	libz	crc32
2.41%	JVM	Interpreter
2.35%	Hadoop	Ljava/io/BufferedInputStream;.read

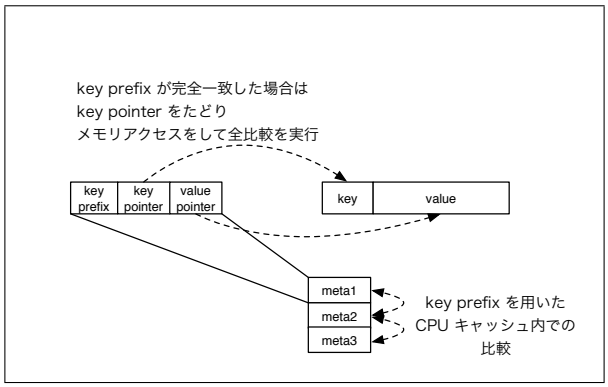

図 3 key-prefix sort の概要

```
1 interface ProxyComparator<KEY> {
2   public int compare(byte[] b1, int s1, int l1, byte
        [] b2, int s2, int l2);
3   int getProxy(KEY key);
4 }
```

図 4 ProxyComprator インタフェースの定義

す．key-prefix sort は，CPU キャッシュのヒット率を向上させるために，key の全体を比較しスワップする代わりに，プレフィックスをあらかじめメタデータとして持ち，このメタデータのスワップによりキャッシュ効率良くソートを行う手法である．prefix が完全一致していた場合は，ポインタ経由でキー本体にアクセスし，キー本体を用いた完全比較をおこなう．この場合は，CPU キャッシュ内に key が保存されている可能性が低いため，性能の劣化が発生する．

Apache Tez では, key-prefix 相当のアルゴリズムを Hive, Pig から見て疎結合に実装するためのインタフェースとして，ProxyComparator を用意している．ProxyComparator により，オブジェクトをバッファにシリアライゼーションする際に prefix 情報を Tez のメタ情報領域に書き込み，Tez はそれを利用してソートを行う．

Tez 組み込みの key-prefix sort 単体の最適化が十分かどうかを評価するために，ProxyComparator を用いた TeraSort プログラムを実装し，評価を行った．結果を図 5 に示す．key-prefix sort により，ジョブ全体を 15%程度高速化することができた．しかしながら，より高速なソートアルゴリズムを Tez に組み込むことができれば，さらなる高速化が見込める．3 章で示したプロファイリング結果のうち，56%がソートに関連する処理であるから，ソートの占める CPU 使用率を全体の 5%ほどに抑えることができれば，最大で 2 倍ほどの高速化が可能となる．

5. 提案手法: key-prefix bitonic merge sort

bitonic merge sort[3] は，複数回の Min 命令, Max 命令, Shuffle 命令を利用することで，条件分岐なしにをソートを実行することができるマージソートの一種である．bitonic merge sort により実行すべき CPU 命令数は増えるが，条件分岐によるパイプラインストールが発生しずらくなり，高速に動作することが報告されている [3].

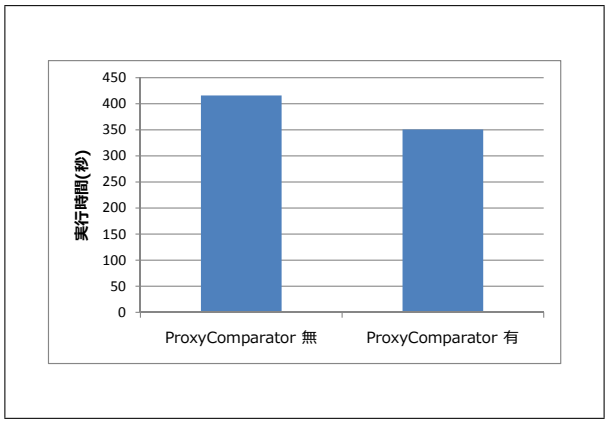

図 5 ProxyComparator を用いた TeraSort プログラムの比較

```
1 method KeyPrefixBitonicSort
2   metabuffer.execBitonicSort()
3   ranges = metabuffer.checkPerfix()
4   for r in ranges:
5     metabuffer.sortByCompleteKey(r.start, r.end)
```

図 6 key-prefix bitonic merge sort の手順

一方で，bitonic merge sort を利用したソートの研究の多くは固定長データを対象としており，可変長データをソートする必要がある分散処理フレームワークにおいてこれらの組み込みを行う方法は自明ではない．そこで，本研究では、bitonic merge sort を可変長データにも対応させた key-prefix bitonic merge sort を提案する．key-prefix bitonic merge sort は，meta データに対する key-prefix sort を行う際に内部で bitonic merge sort を行うことで，より高速なソートを行う方法であり，Inoue らの手法 [6] を拡張し，可変長のデータに対するソートを実行できるようにしたアルゴリズムである．

提案手法の挙動を図 6 に示す．key-prefix bitonic merge sort では，まずキーの prefix を含むメタ情報バッファに対して bitonic merge sort を実行する (2 行目)．bitonic merge sort を行った後，ソート済みのメタ情報バッファの中身を全てスキャンし，同一の値の有無をチェックする．同一の値が存在する場合，キーの prefix が一致していることになるため，その範囲の開始地点と終了地点を表すインデックスを表す構造体の配列 ranges に保存する (3 行目)．最後に，ranges に含まれている範囲に対して，キーの prefix ではなくキー本体を用いて完全比較を行う (4-5 行目)．bitonic merge sort 内部でキー全体の比較を行う場合と比較し，スキャンに O(n) の操作がかかるため，key-prefix bitonic merge sort の計算量は O(nlog(n) + n) となり，定数項が増加するが，キャッシュヒット率が向上し，高速に動作させることができる．

6. 実装

key-prefix bitonic merge sort の効率の良い実装を実現するには，SIMD 命令を利用する必要がある．しかしながら，Tez をはじめとする JVM ランタイム上に構築された分散処理フレームワークの場合，JVM 上から SIMD 命令を生成することが現状不可能である．よって，Tez に key-prefix bitonic merge sort を組み込むには，JVM から C の関数を呼ぶ機能である JNI (Java Native Interface) を用いて実装する必要がある．本稿では，SIMD 命令を用いて最適化した　bitonic merge sort を JNI 経由で呼び出す様な実装を行うことで，その有効性と実現可能性を確認する．メモリアラインメントと，JNI オーバヘッドの削減を考慮に入れた API 定義を図 7 に示す．以下では、各 API についての設計と効果について述べる。

```
1 class NativeSort {
2   private static native void nativeChunkedSort(
3       ByteBuffer data, int dataOffset, int dataLength
          , int numItem);
4
5   private static native int getIndexForAligned(
        byte[] data);
6 }
```

図 7 key-prefix bitonic merge sort を JNI から呼ぶための API

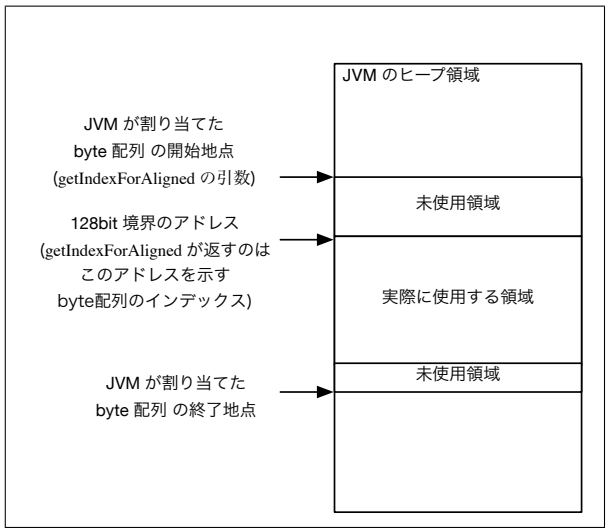

図 8 getIndexForAligned の挙動

6.1 メモリアラインメント

Intel CPU における SIMD の提供するメモリアクセス命令の一部は，アラインメントを保持する必要がある．例えば，Intel の提供する SIMD 命令である SSE 命令の場合は 16 バイト境界，AVX 命令の場合は 32 バイト境界のアラインメントを保持する必要がある．しかしながら，JVM の管理するメモリのアラインメントは仕様としては定められておらず，メモリを確保する API にもアラインメントを保持する方法が存在しない．そこで，JVM ヒープ上にメモリを確保する際に， 15 バイト多めにメモリを確保しておき，

アラインメントがそろった番地を開始地点として配列を利用することとした．これを実現するための API が，図 7 の getIndexForAligned 関数である．getIndexForAligned 関数の挙動を図 8 に示す．getIndexForAligned 関数は，その内部で与えられた byte 配列の関数ポインタを引数にとり，下位 16 バイトが 0 になっているアドレスを示す開始地点の配列のインデックスを返す．この返り値のインデックスは JVM の割り当てた byte 配列のインデックスであるため，JVM のヒープ領域上であってもアラインメントを揃えた状態の配列を用意することができ，SSE 命令を利用することが可能となる．

6.2 JNI のオーバヘッド削減

2015 年時点では，JVM ランタイム上から SSE 命令を直接出力するようなバイトコードをプログラマが指定することで任意の箇所に SSE 命令を埋め込むようなインタフェースは存在しない．よって，SSE 命令を用いた C 言語で記述されたソートモジュールを利用し，JVM からの呼び出しには JNI を用いることとした．しかしながら，細粒度で JNI の関数を呼び出すと，JNI 関数呼び出しの際にオーバヘッドが生じてしまい速度低下が起こる可能性がある．この問題を回避するために，今回の関数では byte 配列自体を引数にとり，呼び出しの粒度を大きくすることとした．図 7 の nativeChunkedSort 関数はその内部で byte 配列を int の配列として見なしソートを行う関数である．

7. 評価

提案手法の有効性の確認するため，6 章で述べた JVM 上からネイティブコードと提案手法の部分実装の比較を行った．計算機には Amazon EC2 の m3.xlarge(E5-2670 v2 2.50GHz vCPU 4 コア，メモリ 15GB，ローカル SSD 40GB x 2) インスタンスを用いた．比較対象は，Hadoop 内で用いられている QuickSort の実装 (quick sort(Java))，図 7 で提案した API 内部で C の QuickSort を用いる実装 (quick sort(C))，図 7 で提案した API 内部で key-prefix bitonico sort の一部である bitonic merge sort を用いる実装 (bitonic merge sort) の 3 つである．

結果を図 9 に示す．quick sort(Java) と quick sort(C) は約 2 倍，quick sort(Java) と bitonic merge sort は約 10 倍の性能差であった．なお，同環境でソート後の同一の値チェックをするためのスキャンにかかる時間を測定したところ，0.1 秒程度で完了した．つまり，ProxyComparator を用いた場合よりも，最大で 10 倍の高速化が行える可能性があることを示している．3 章で示したプロファイリング結果のうち，56%がソートに関連する処理であるから，ソートの占める CPU 使用率を全体の 5.6%ほどに抑えることができ，最大で 2 倍ほどの高速化が可能となる．

しかしながら，可変長キーによる完全比較処理を追加することで現在よりも改善率は下がるため，キーの完全比較部分についても注意深く実装を行う必要がある．可変長キーの完全比較処理の実装と評価，および Hive を用いた全体性能の評価については今後の課題とする．

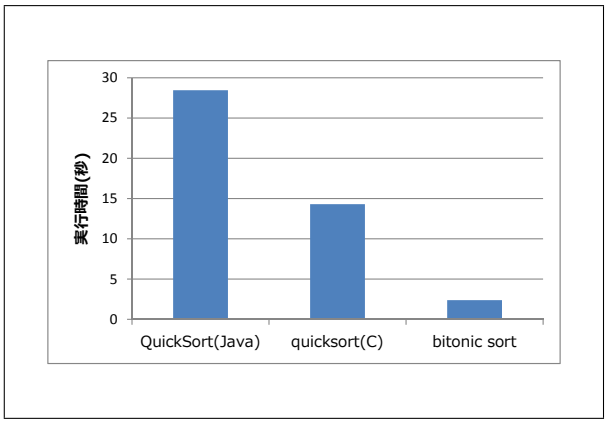

図 9 Hadoop 内で用いられている quick sort と，図 7 で提案した API を用いた C の quick sort および bitonic merge sort の比較

8. 関連研究

Dawei らは，Fingerprinting Based Grouping という集約演算を含むクエリを高速化する手法を提案している [8]．この手法では，キーをハッシュ関数にかけた結果 (Fingerprint) の比較を集約時のキーの比較の前に行うことで，キーの完全比較を行う回数を削減し，Fingerprint が一致しない場合のハッシュテーブル構築を高速化することが可能である．また, Mars では GPGPU 上に構築された MapReduce 処理系の内部で bitonic sort を利用する案を提案している [5]．Mars においても，キー全体にアクセスする代わりに固定長の Fingerprint を比較し，一致した場合はキーの完全比較を行うことで一部のジョブを高速化することができる．提案手法と比較して，Dawei らの方法および Mars においては，はキー全体にアクセスする代わりに固定長の Fingerprint を比較し，Fingerprint が一致した場合はキーの完全比較を行う点において似ている．しかしながら，これらの手法は，比較の際にキーの perfix ではなくハッシュ関数を用いるためキー間の大小関係を知ることができない点が異なる．このため，Fingerprinting Based Grouping は結果全体をソートするようなクエリ (ORDER-BY を含むクエリ) などと併用することができず，応用範囲が限られる．提案手法では，ボトルネック箇所であるソート自体を高速化することで，ソートを含むクエリと集約演算両方を高速化することが可能である．

9. おわりに

本稿では，分散処理フレームワークである Apache Tez の性能解析を行い，IO が支配的なワークロードである TeraSort においてもフレームワーク側のソート機構が CPU ボ

トルネックになることを示した．

Apache Tez は CPU ボトルネックを部分的に解消する機構として ProxyComparator を用いた key-prefix sort を実現する機構を備えているが，実験によりその最適化機構には改善の余地があることを示した．今後は，Apache Tez に提案手法の組み込みを行い，様々な MapReduce ジョブや Hive クエリを用いて性能評価を行うことで，その有用性の評価を行っていく．

10. 謝辞

本稿の記述に辺り，Hortonworks および Apache Software Foundation の Bikas Saha 氏，Gopal Vijayaraghavan 氏，Rajesh Balamohan 氏には Apache Tez の詳細に関する助言を頂いた．心より感謝する．

参考文献

[1] Apache Hadoop: http://hadoop.apache.org/.
[2] Apache Hadoop Wiki: http://wiki.apache.org/hadoop/PoweredBy.
[3] Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y.-K., Baransi, A., Kumar, S. and Dubey, P.: Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture, *Proc. VLDB Endow.*, Vol. 1, No. 2, pp. 1313–1324 (online), DOI: 10.14778/1454159.1454171 (2008).
[4] Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, USENIX Association, pp. 10–10 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1251254.1251264⟩ (2004).
[5] He, B., Fang, W., Luo, Q., Govindaraju, N. K. and Wang, T.: Mars: A MapReduce Framework on Graphics Processors, *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, New York, NY, USA, ACM, pp. 260–269 (online), DOI: 10.1145/1454115.1454152 (2008).
[6] Inoue, H. and Taura, K.: SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures, *Proc. VLDB Endow.*, Vol. 8, No. 11, pp. 1274–1285 (online), DOI: 10.14778/2809974.2809988 (2015).
[7] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks, *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, New York, NY, USA, ACM, pp. 59–72 (online), DOI: 10.1145/1272996.1273005 (2007).
[8] Jiang, D., Ooi, B. C., Shi, L. and Wu, S.: The Performance of MapReduce: An In-depth Study, *Proc. VLDB Endow.*, Vol. 3, No. 1-2, pp. 472–483 (online), DOI: 10.14778/1920841.1920903 (2010).
[9] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J. and Lomet, D.: AlphaSort: A Cache-sensitive Parallel External Sort, *The VLDB Journal*, Vol. 4, No. 4, pp. 603–628 (online), available from ⟨http://dl.acm.org/citation.cfm?id=615232.615237⟩ (1995).
[10] Ohta, K.: Hadoop meets Cloud with Multi-Tenancy, http://www.slideshare.net/treasure-data/hadoop-meets-cloud-with-multitenancy-16107610 (2013).
[11] Olston, C., Reed, B., Srivastava, U., Kumar, R. and Tomkins, A.: Pig latin: a not-so-foreign language for data processing, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, New York, NY, USA, ACM, pp. 1099–1110 (online), DOI: 10.1145/1376616.1376726 (2008).
[12] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S. and Chun, B.-G.: Making Sense of Performance in Data Analytics Frameworks, *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, Berkeley, CA, USA, USENIX Association, pp. 293–307 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2789770.2789791⟩ (2015).
[13] Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A. and Curino, C.: Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, ACM, pp. 1357–1369 (online), DOI: 10.1145/2723372.2742790 (2015).
[14] Silberstein, A. E., Sears, R., Zhou, W. and Cooper, B. F.: A batch of PNUTS: experiences connecting cloud batch and serving systems, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, New York, NY, USA, ACM, pp. 1101–1112 (online), DOI: 10.1145/1989323.1989441 (2011).
[15] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R. and Liu, H.: Data warehousing and analytics infrastructure at facebook, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, New York, NY, USA, ACM, pp. 1013–1020 (online), DOI: 10.1145/1807167.1807278 (2010).
[16] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R. and Liu, H.: Data Warehousing and Analytics Infrastructure at Facebook, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, ACM, pp. 1013–1020 (online), DOI: 10.1145/1807167.1807278 (2010).
[17] Treasure Data ' s Plazma: Columnar Cloud Storage：http://blog.treasure-data.com/post/53534943282/treasure-datas-plazma-columnar-cloud-storage (2013).
[18] Yahoo!, O. O.: Terabyte Sort on Apache Hadoop, (online), available from ⟨http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf⟩ (2008).
[19] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: Cluster Computing with Working Sets, *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, Berkeley, CA, USA, USENIX Association, pp. 10–10 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1863103.1863113⟩ (2010).