情報処理学会研究報告
IPSJ SIG Technical Report

合成ベンチマークによる MapReduce 処理系 SSS の性能評価

小川宏高[†1]　中田秀基[†1]　工藤知宏[†1]

MapReduce プログラムで大容量データを処理する際の実行速度は、Map への入力されるデータ、Reduce から出力されるデータ、さらに Map 処理と Reduce 処理の間でやりとりされる中間データの量と性質によって大きく変化する。特に中間データを蓄積する方法は、処理系によって大きく異なり、システム全体の特性を決定する要因となりうる。われわれは開発中の MapReduce 処理系 SSS の特性を確認するために、先行研究で開発した合成ベンチマークプログラムを用いて、SSS の評価を行い、Hadoop と比較した。合成ベンチマークプログラムは、MapReduce プログラムのデータ入出力部分のみを抽出したもので、パラメータを変更することでさまざまなプログラムの入出力パターンを再現することが可能となっている。評価の結果、以下を確認した。1) SSS は Hadoop と比較して一般に高速に動作する、2) SSS の Map 後の Combine 処理は非常に有用である、3) ベンチマークの設定には改善が必要である。

Performance Evaluation of a MapReduce System SSS with a Synthetic Benchmark Suite

Hirotaka Ogawa ,[†1] Hidemoto Nakada [†1]
and Tomohiro Kudoh [†1]

The performance of MapReduce programs heavily depend on their data I/O workloads, namely; input data to Mappers, output data from Reducers, and shuffled data between Mappers and Reducers. We evaluated SSS, a MapReduce system we are developing, and compared it with Hadoop, the most widely deployed MapReduce system, using a synthetic benchmark suite, which is proposed in our previous work. The benchmark suite is designed to mimic three extreme case of I/O worlkloads;n namely, read, write, and shuffle intensive workloads. The result showed that; 1) SSS is faster than Hadoop in general, 2) SSS's combiner is quite effective for shuffle intensive tasks, 3) the benchmark method has several points to be improved.

1. はじめに

MapReduce[1] は、大規模なデータインテンシブアプリケーションの実装手段の一つであり、そのオープンソース実装である Hadoop[2] は、当初のターゲットであったログ解析の分野だけでなく、科学技術分野においても広く用いられつつある。われわれは、大規模データの高速な処理を目的とした MapReduce 処理系 SSS[3] を開発している。Hadoop が HDFS と呼ばれるファイルシステムを基盤としているのに対して、SSS は分散 Key Value ストア（KVS）を基盤とする点が大きく異なる。SSS は、これまでにいくつかのトイアプリケーションでは Hadoop と比較して高速であることが立証されているが、その特性はよく分かっていない。

本稿では、先行研究[4] でわれわれが提案した合成ベンチマークセットを用いて SSS を評価し Hadoop と比較することで、SSS の特性を明らかにする。目的は、任意のアプリケーションに対する SSS の適否を事前に知ることである。

本稿の構成は以下のとおりである。2 節で、MapReduce プログラミングモデルおよび比較の対象である Hadoop について概説し、続いて 3 節で SSS について概説する。4 節で、利用するベンチマークプログラムの構成について述べる。5 節で、ベンチマークプログラム実行の結果を示し、6 節で実行結果に関する議論を行う。7 節はまとめである。

2. MapReduce と Hadoop

2.1 MapReduce

MapReduce とは、入力キーバリューペアのリストを受け取り、出力キーバリューペアのリストを生成する分散計算モデルである。MapReduce の計算は、Map と Reduce という二つのユーザ定義関数からなる。これら 2 つの関数名は、Lisp の 2 つの高階関数からそれぞれ取られている。Map では大量の独立したデータに対する並列演算を、Reduce では Map の出力に対する集約演算を行う。

一般に、Map 関数は 1 個の入力キーバリューペアを取り、0 個以上の中間キーバリューペアを生成する。MapReduce のランタイムはこの中間キーバリューペアを中間キーごとに

†1 独立行政法人 産業技術総合研究所 / National Institute of Advanced Industrial Science and Technology (AIST)

グルーピングし、Reduce 関数に引き渡す。このフェイズを**シャッフル**と呼ぶ。Reduce 関数は中間キーと、そのキーに関連付けられたバリューのリストを受け取り、0 個以上の結果キーバリューペアを出力する。各 Map 関数、Reduce 関数はそれぞれ独立しており、相互に依存関係がないため、同期なしに並列に実行することができ、分散環境での実行に適している。また、関数間の相互作用をプログラマが考慮する必要がないため、プログラミングも容易である。これは並列計算の記述において困難となる要素計算間の通信を、シャッフルで実現可能なパターンに限定することで実現されている。

シャッフルは MapReduce システムの実装上でも重要なポイントであり、シャッフルの実装方法によって MapReduce システムの特性は大きく異なりうる。

2.2 Hadoop

代表的なオープンソースの MapReduce 処理系である Hadoop の動作を**図 1** に示す。

Hadoop では、入力データは HDFS 上のファイルとして用意される。Hadoop は、まず MapReduce ジョブへの入力をスプリットと呼ばれる固定長の断片に分割する。スプリットのサイズが小さければ負荷分散が容易になる一方、スプリットの管理とタスク生成のオーバーヘッドが顕著になる。このため、スプリットのサイズは通常バックエンド分散ファイルシステム HDFS のブロックサイズと等しくなるようになっている。次に、スプリット内のレコードに対して map 関数を適用する。この処理を行うタスクを Map タスクと呼ぶ。Map タスクは各スプリットに関して並列に実行される。map 関数が出力した中間キーバリューデータは、partition 関数（キーのハッシュ関数など）によって、Reduce タスクの個数 R 個に分割され、各パーティションごとにキーについてソートされる。ソートされたパーティションはさらに combiner と呼ばれる集約関数によってコンパクションされ、最終的には Map タスクが動作するノードのローカルディスクに書き出される。

一方の Reduce 処理では、まず Map タスクを処理したノードに格納されている複数のソート済みパーティションをリモートコピーし、ソート順序を保証しながらマージする。（結果的に得られた）ソート済みの中間キーバリューデータの各キーごとに reduce 関数が呼び出され、その出力は HDFS 上のファイルとして書き出される。

Hadoop システムは、入力データや結果出力データと全く異なる方法で、中間データを保持する。このため、中間データを入力データとして再利用することは難しい。また、入力データや結果データはフラットなファイルシステムである HDFS に置かれる。このため、入力時出力時にファイルフォーマットとキーバリューペア間の変換操作が必要となる。

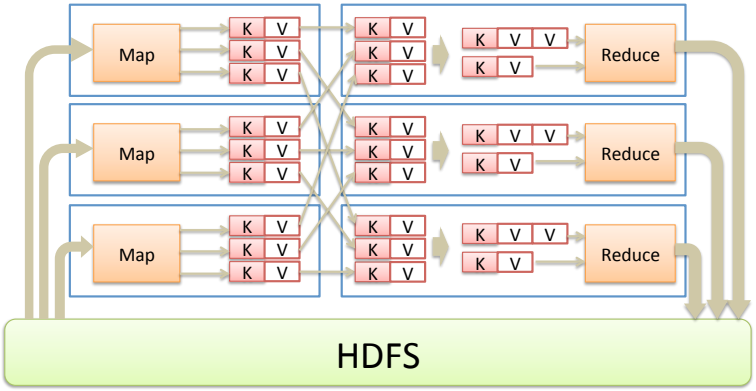

図 1 Hadoop の概要

3. SSS

SSS[3] は、われわれが開発中の MapReduce 処理系である。本節では SSS の動作について概説する。

3.1 SSS の設計

SSS は HDFS のようなファイルシステムを基盤とせず、分散 KVS を基盤とする点に特徴がある。入力データは予めキーとバリューの形で分散 KVS にアップロードしておき、出力結果も分散 KVS からダウンロードする形となる。これは煩雑に思えるかもしれないが、Hadoop の場合でも同様に HDFS を計算時のみに用いる一時ストレージとして運用しているケースも多く、それほどデメリットであるとは考えていない。

SSS ではデータをキーに対するハッシュで分散した上で、Owner Compute ルールにしたがって計算を行う。つまり、各ノード上の Mapper/Reducer は自ノード内のキーバリューペアのみを対象として処理を行う。これは、データ転送の時間を削減するとともに、ネットワークの衝突を防ぐためである。

Mapper は、生成したキーバリューペアを、そのキーでハッシングして、保持担当ノードを決定し直接書き込む。書きこまれたデータは各ノード上の KVS によって自動的にキー毎にグループ分けされる。これをそのまま利用して、Reduce を行う。つまり SSS においては、シャッフルは、キーのハッシングと KVS によるキー毎のグループ分けで実現されるこ

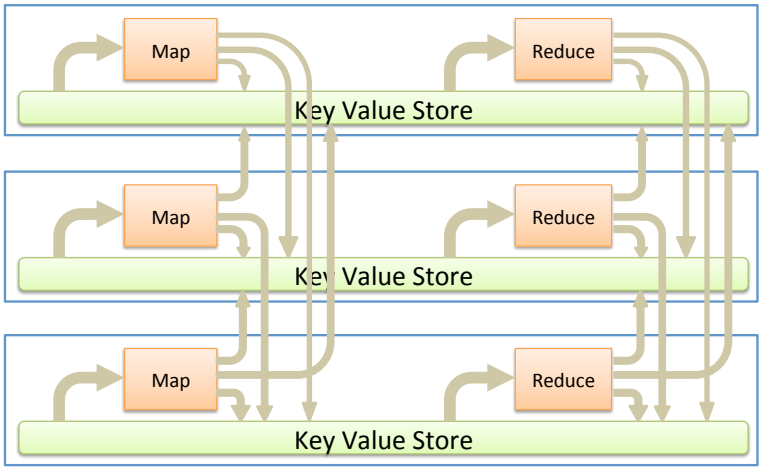

図 2 SSS におけるデータの流れ

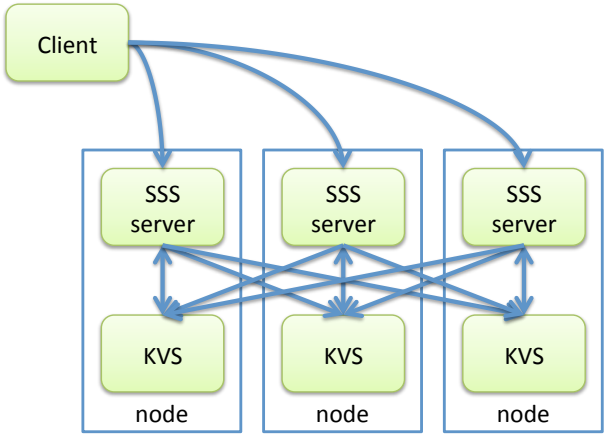

図 3 SSS の構成

とになる。

SSS のもうひとつの特徴は、Map と Reduce を自由に組み合わせた繰り返し計算が容易にできることである。前述のように、SSS では Map と Reduce の間でやりとりされるデータも KVS に蓄積されるため、Map と Reduce が 1 対 1 に対応している必要がない。したがって、任意個数、段数の Map と Reduce から構成される、より柔軟なデータフロー構造を対象とすることができる。

3.2 SSS の構成

SSS の構成を図 3 に示す。各ノード上では、SSS Server と KVS のサーバとを実行する。Client プログラムは Map ジョブ、Reduce ジョブを管理する役割を担い、各ノード上の SSS Server に対してジョブの実行を指示する。SSS サーバは Java で記述されている。

3.3 SSS の分散 KVS 実装

SSS は、単体 KVS をキーに対するハッシングで分散化したものを分散 KVS として用いる。図 3 に示したとおり、単体 KVS は独立して動作しており、相互の通信は行わない。KVS に対するクライアントである SSS サーバ群が共通のハッシュ関数を利用することで、総体としての分散 KVS が構成されている。

単体 KVS としては、Tokyo Cabinet[5] を、SSS が多用するソート済みデータのバルク書き込み、およびレンジに対するバルク読み出しに特化してカスタマイズしたものを用いた。Tokyo Cabinet へのリモートアクセスには、サーバ側に Tokyo Tyrant[6] を、クライアント側に jTokyoTyrant[7] を用いている。

4. MapReduce 合成ベンチマーク

本節では、先行研究[4] で提案した合成ベンチマークを概説する。

4.1 仮　　定

本ベンチマークでは、単純化のために以下の仮定をおく。

- 個々のキーバリューペアを、基盤となるストレージシステム上の要素オブジェクトにマップする。例えば、Hadoop では HDFS 上の 1 ファイルに対してキーバリューペアをマップする。
- バリューのサイズは各フェーズにおいて一定であるものとする。
- Reduce タスクはストレージノードの数だけ実行されるものとする。言い換えると、最大の並列度で Map タスク、Reduce タスクとも実行されるものとする。

これらの前提は、基盤の異なる MapReduce 処理系を比較するための便宜的なものである。この前提の妥当性に関しては、6 節で議論する。

4.2 設　　計

前項で述べた仮定の範囲内で、いくつかのパラメータを変化させ、さまざまな I/O ワー

表 1 Benchmarking Environment

Mapper 関連	
nodes	ノード数。ストレージノード数、Map タスクが起動されるノード数、Reduce タスクの数に等しい。
initialKeyCount	入力キーの個数。ノードあたりの入力キーバリューペアの個数は (initialKeyCount/nodes) に等しい。
initialValueLength	入力キーに対応する値のサイズ。
mapoutKeyCount	map 関数が生成する中間キーの個数。ノードあたりの中間キーバリューペアの個数は (mapoutKeyCount/nodes)、一回の map 関数の出力する中間キーバリューペアの個数は (mapoutKeyCount/initialKeyCount) にそれぞれ等しい。
mapoutValueLength	map 関数が生成する中間キーに対応する値のサイズ。
mapoutUniqueKeyCount	map 関数が生成する中間キーのうち、ユニークなものの個数。この値は reduce 関数の入力キー、ならびに出力キーの個数に等しい。また、各 map 関数は、同一のキーを持つ中間キーバリューペアを ((mapoutKeyCount/initialKeyCount)/mapoutUniqueKeyCount) 個生成する。
reduceoutValueLength	reduce 関数が生成する値のサイズ。
Partiotioner, Combiner 関連	
partitionMethod	中間キーを partition する際の方法。各 map 関数が生成する中間キーが平均的に Reduce タスクに分散される *average*、中間キーがローカルに起動される Reduce タスクに分散される *local*、中間キーが隣接ノードに起動される Reduce タスクに分散される *shifted* (必ず中間キーバリューデータの全交換が発生する) などを取り得る。ただし、Hadoop の実装では、局所性を利用した Reduce タスクの割り付けはできないため、local や shifted の実現は不可能である。
combinerEnabled	combiner の適用の可否の指定。combiner 自体は reduce 関数と同一の関数が適用されるものとする。ただし、Hadoop の実装では、combiner 関数の適用を強制することは不可能なため、適用を保証するものではない。
combiningFactor	combiner によってソート済みのパーティションが集約される比率。上述の通り、Hadoop では制御できない。

クロードを再現することのできるベンチマークプログラムを構成した。なお、本ベンチマークは、I/O ワークロード部分のみを対象としているため、演算処理を全く行わない。

表 1 に、MapReduce アプリケーションのワークロードを規定するパラメータ、および Partitioner、Combiner の動作設定パラメータを示す。

4.3 合成ベンチマークの実装

合成ベンチマークを SSS のアプリケーションとして実装した。また、比較のため、Hadoop 0.20.2 のアプリケーションとして記述した。すべてのキーバリューペアはキーを long 型で、値を byte 配列型でそれぞれ表現している。

表 1 で述べたパラメータを設定ファイルとして与えることで動作をカスタマイズし、特徴的な I/O ワークロードを再現する。

表 2 I/O ワークロードの設定

	Read Intensive	Write Intesive	Shuffle Intensive
initialKeyCount	16 - 65536	16	16
initialValueLength	1GiB - 256KiB	1	1
mapoutKeyCount	256	4194304	16 - 262144
mapoutUniqueKeyCount	16	16 - 65536	16
mapoutValueLength	1	1	1GiB - 64KiB
reduceoutValueLength	1	1GiB - 256KiB	1
combinerEnabled	false	false	false | true

4.4 I/O ワークロードの設定

以下のように特徴的な 3 種類のワークロードを設定した。各ワークロードで用いたパラメータを、表 2 に示す。

- **Read Intensive**
  このワークロードでは、入力データが多く、相対的に Map タスクの処理がドミナントになるアプリケーションを再現する。具体的には入力データの総量が 16GiB となるように、initialKeyCount と initialValueLength の組み合わせを選んだ。
- **Write Intensive**
  このワークロードでは、出力データが多く、相対的に Reduce タスクの処理がドミナントになるアプリケーションを再現する。具体的には出力データの総量が 16GiB となるように、mapoutUniqueKeyCount と reduceoutValueLength の組み合わせを選んだ。
- **Shuffle Intensive**
  このワークロードでは、入出力データそのものは小さいが、シャッフルに要する処理がドミナントになるアプリケーションを再現する。具体的には Map タスクの出力キーバリューペアの総量が 16GiB となるように、mapoutKeyCount と mapoutValueLength の組み合わせを選んだ。
  `mapoutUniqueKeyCount` を 16 としているが、これは Map で生成されるキーバリューペアのキー部分が全部で 16 種類しかないことを意味する。このため、shuffle のフェイズで大規模なマージ処理が必要になる。

5. ベンチマークによる評価

4 節で示した合成ベンチマークを用いて特徴的な 3 つの MapReduce の I/O ワークロー

表 3 Benchmarking Environment

Number of nodes	17 (*)
CPU	Intel(R) Xeon(R) W5590 3.33GHz
Number of CPU per node	2
Number of Cores per CPU	4
Memory per node	48GB
Operating System	CentOS 5.5 x86_64
Storage (ioDrive)	Fusion-io ioDrive Duo 320GB
Storage (SAS HDD)	Fujitsu MBA3147RC 147GB/15000rpm
Network Interface	Mellanox ConnextX-II 10G Adapter
Network Switch	Cisco Nexus 5010

(*) うち 1 ノードはマスタサーバ

ドを再現し、実クラスタ環境での性能を評価した。また比較対象として Hadoop でも実行を行った。また、データを保持する媒体としてフラッシュストレージ（以下 SSD) と SAS ハードディスク（以下 HDD）を用い比較した。

以下、まず評価環境について説明し、次に各ワークロードごとの実験結果を示す。

5.1 評価環境

評価には、**表 3** に示すように、1 台のマスターノードと 16 台のワーカーノード（ストレージノードと MapReduce 実行ノードを兼ねる）からなる小規模クラスタを用いた。各ノードは 10Gbit Ethernet で接続され、各ワーカーノードは Fusion-io ioDrive Duo 320GB と Fujitsu の 147GB SAS HDD を備えている。

比較評価に使用した Hadoop のバージョンは、Cloudera Distribution for Hadoop の 0.20.2+320 である。dfs.replication を 1 に設定することで HDFS のレプリカ生成を抑制している。また、各 Map タスク、Reduce タスクが使用できるヒープのサイズは 2GB に設定してある。

5.2 評価結果

以下、個々のワークロードでの結果を示す。グラフは X 軸に主要パラメータ Y 軸に実行時間を取っている。X 軸 Y 軸ともに対数表記となっていることに注意されたい。また、パラメータに関わらず、入出力されるデータの総量は常に一定であることにも注意が必要である。すなわち、I/O 処理そのもののみに限れば、パラメータによらず実行時間は一定になるはずである。

5.2.1 Read Intensive

図 4 に Read Intensive ワークロードの結果を示す。まず、Hadoop では InitialKeyCount が大きくなるにつれ、実行時間が長くなるのに対して SSS ではむしろ高速化することが目

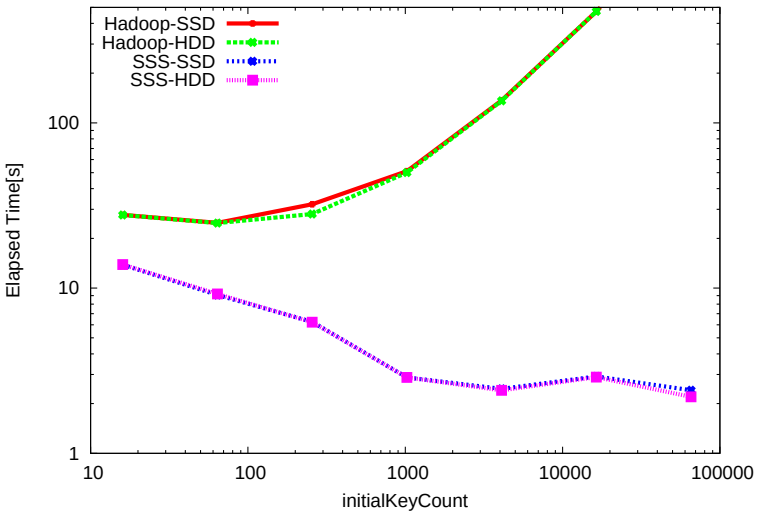

図 4 Read Intensive

を引く。これは、Hadoop が利用する HDFS が多数の小容量ファイルを扱うのに適していないからであると思われる。

SSS では InitialKeyCount が小さい領域で遅いのは、負荷分散がうまくいっていないためだと考えられる。SSS はハッシングで、入力データを各ノードに分散し、そのノードで処理する。InitialKeyCount が小さいとハッシュ値で分散を行っても均等に分散されないため、負荷の不均衡が発生し、終了が遅いノードの終了を全体が待つことになり低速化する。

また、SSD と HDD の明らかな差異は見られなかった。

5.2.2 Write Intensive

図 5 に Write Intensive ワークロードの結果を示す。mapoutUniqueKeyCount が 16 の場合に、SSD と HDD で大きな差が出ているが、その他の部分では、有意な差は見られない。SSS での実行は安定している。Hadoop では、mapoutUniqueKeyCount が 4096 以上、initialValueLength が 4MiB 以下で性能の低下が見られる。これは Reduce タスクの生成と出力データの管理に要するオーバーヘッドの増加によるものである。

なお、mapoutUniqueKeyCount を 64K 以上に設定すると、Hadoop では Reduce タスクが同時にオープンできるファイル数を超えるためにベンチマークは完走できなかった。

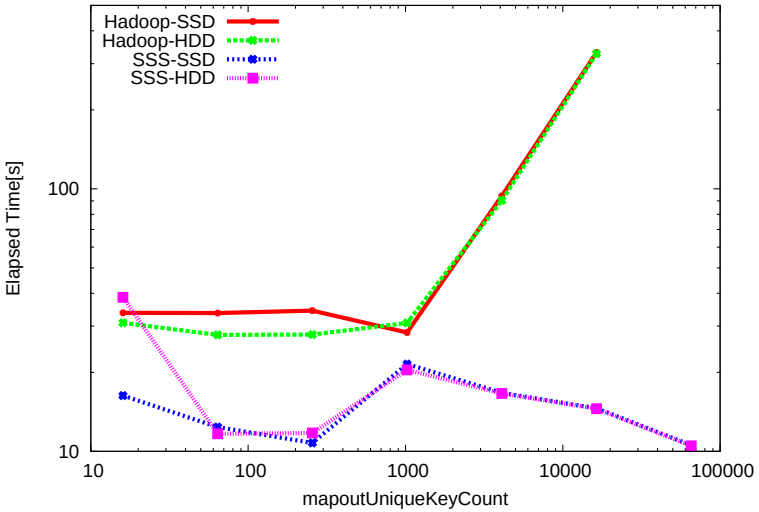

図 5 Write Intensive

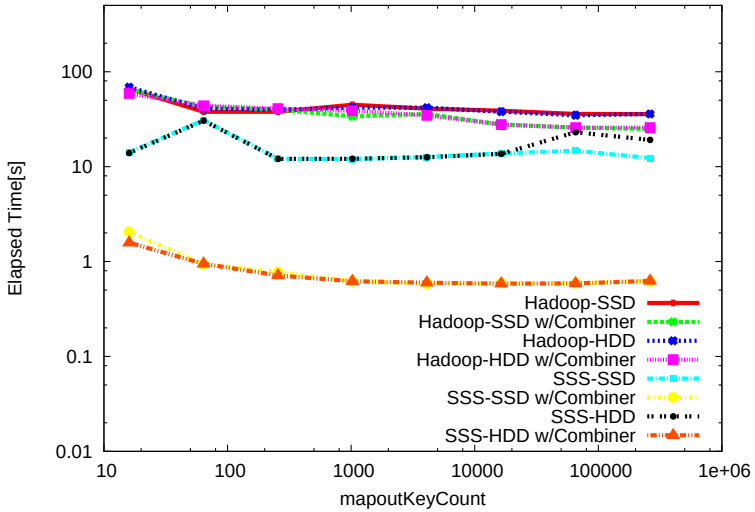

図 6 Shuffle Intensive

5.2.3 Shuffle Intensive

図 6 に Shuffle Intensive ワークロードの結果を示す。SSD と HDD では大きな差は見られない。

combiner に関しては SSS では劇的に有効で 1 桁以上の性能向上が見られる。これに対して、Hadoop ではそれほど大きな向上はみられない。それでも、mapKeyCount が 1024 以上では combiner によって 30%ほど性能が向上している。

6. 議　　論

6.1 全般的な実行時間

全般に SSS の実行時間は Hadoop に比して短い。これは、Map タスク起動のコストが小さいからであると考えられる。Hadoop では、各入力データチャンクに対してタスクが構成されマスタワーカ方式で各ノード上の独立した Java VM プロセス内で実行される。このためタスク数が多いうえ、個々の起動コストも大きい。

これに対して SSS では Map タスク、Reduce タスクは各ノード上の SSS サーバ内の一連のデータフローとして実現されている。タスクの個数はノード数に等しく、起動のコストは小さい。

このため、一般に SSS のタスク起動オーバヘッドは Hadoop のそれに比べて小さく、従って実行時間も短い。

6.2 データ規模の妥当性

今回のベンチマークでは、データの規模は、すべて 16GiB とした。この値は、ノードあたりでわずか 1GiB であり、各ノードのメモリ搭載量 48GB と比較すると非常に小さい。このため、データがディスクキャッシュ、もしくは KVS のメモリ領域にすべて乗ってしまっている可能性がある。

SSD と HDD の性能差があまり見られなかったことも規模が過小であることに起因すると考えられる。より大規模なデータセットに対応してベンチマークを行う必要がある。

また、キーバリューペアの個数も過小である可能性がある。今回は最大で 2^{18} 個までしか扱っていないが、これは MapReduce のプログラムとしては小さい。

6.3 仮定の妥当性

今回利用したベンチマークは 4.1 で示した仮定ををおいて実装されている。このうちの、「1 つのキーバリューペアを 1 つのストレージオブジェクトにマップする」と言う仮定に基

づき、Hadoop 版においては一つのキーバリューを一つの HDFS 上のファイルとして実装している。この実装は Hadoop としては一般的でなく、効率も悪い。とくに Read Intensive でキーの数の増加にともない性能が著しく低下するのはこのためであると考えられる。この仮定をとりのぞき、処理系に適した実装を許すべきである。

6.4 演算と I/O のオーバラップ

高性能並列計算では演算と I/O をオーバラップさせることが一つのキーポイントとなる。実際、SSS では演算と I/O を独立したスレッドに割り当てることで両者がオーバラップした動作を実現している。

しかし、今回利用した提案ベンチマークでは演算を全く行わないため、システムが演算と I/O のオーバラップを実現していてもいなくても結果が全く変わらない。より詳細な動作比較のためには I/O ワークロードだけでなく演算も再現したベンチマークの開発が必要である。

7. おわりに

開発中の MapReduce 処理系 SSS に対して、自由に I/O ワークロードを合成できる合成ベンチマークを適用し、性能評価を行った。その結果、今回ベンチマークで測定した範囲では Hadoop と比較して安定して高速であること、特に Shuffle インテンシブなジョブに関して Combiner により効率的に動作することが確認できた。一方、今回測定したベンチマークの妥当性にも一定の疑問が生じた。

今後の課題は以下の通りである。

- ベンチマークの設定を見直し、再度評価を行う。具体的には、データの規模、キーバリューの規模を拡大し、キーバリューペアとストレージ上の要素オブジェクトの 1 対 1 対応を改め、一つのファイルに複数のキーバリューを納めることを許す。
- 大規模な実アプリケーションで評価を行い、ベンチマークでの評価の妥当性を検証する。

謝　辞

本研究の一部は、独立行政法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務「グリーンネットワーク・システム技術研究開発プロジェクト（グリーン IT プロジェクト）」の成果を活用している。

参 考 文 献

1) Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (2004).
2) : Hadoop, http://hadoop.apache.org/.
3) Ogawa, H., Nakada, H., Takano, R. and Kudoh, T.: SSS: An Implementation of Key-value Store based MapReduce Framework, *Proceedings of 2nd IEEE International Conference on Cloud Computing Technology and Science (Accepted as a paper for First International Workshop on Theory and Practice of MapReduce (MAPRED'2010))*, pp.754–761 (2010).
4) 小川宏高，中田秀基，工藤知宏：合成ベンチマークによる MapReduce の I/O 性能評価手法，情報処理学会研究報告　2011-HPC-129 (2011).
5) FAL Labs: Tokyo Cabinet: a modern implementation of DBM, `http://fallabs.com/tokyocabinet/index.html`.
6) FAL Labs: Tokyo Tyrant: network interface of Tokyo Cabinet, `http://fallabs.com/tokyotyrant/`.
7) : Java Binding of Tokyo Tyrant, `http://code.google.com/p/jtokyotyrant`.