分散 KVS に基づく MapReduce 処理系 SSS

中田 秀基, 小川 宏高, 工藤 知宏
独立行政法人 産業技術総合研究所

概要

大量ログ解析のための技術として提案された MapReduce は、新しい並列プログラミング手法として、より広範なアプリケーションへの応用が期待されているが、そのためには、Map と Reduce を自由に組み合わせたワークフローの実行が必須となる。しかし、広く利用されている MapReduce 処理系 Hadoop は、ファイルシステムをベースとし、Map と Reduce の間の通信を特別扱いにする構成を取っているため、Map と Reduce が 1 対 1 に対応する構造しか記述することができない。このためワークフローの実行が非効率となる。われわれは分散 KVS をベースに構成した MapReduce 処理系 SSS を提案する。SSS では Map と Reduce は、いずれも分散 KVS からデータを読み込み、分散 KVS に書き込む。このため Map と Reduce を自由に組み合わせたワークフローを構成し、効率的に実行することが可能になる。本稿では、SSS の実装を詳述するとともに、合成ベンチマークプログラムおよび、K-means による Hadoop との比較評価を示す。評価の結果、SSS は Hadoop よりも殆どの場合において高速であることが確認できた。

SSS: A MapReduce framework based on Distributed KVS

Hidemoto Nakada, Hirotaka Ogawa, Tomohiro Kudoh
National Institute of Advanced Industrial Sceince and Technology

Abstract

MapReduceis considered to be promissing as a parallel description model for broad range of applications. For that purpose, a flexble MapReduceframework where programmers can easily combine Mappers and Reducers into a workflow graph. Unfortunately, Hadoop, the most widely used MapReduceframework, is not flexible enough, because Hadoop is implemented on top of a filesystem called HDFS and Mappers and Reducers are tied together one by one. We propose a MapReduceframework based on distributed KVS, called SSS. In SSS, both of Mappers and Reducers read and write to/from KVS. This makes flexible combination of Mappers and Reducers. In this paper, we describe detailed design and implementation of SSS. We also provide benchmark result with synthetic IO benchmark and K-means application. The result showed that SSS is faster than Hadoop in general,

1 はじめに

大量ログ解析のための技術として提案された MapReduce[1] は、新しい並列プログラミング手法として、より広範な大規模データインテンシブアプリケーションへの応用が期待されている。そのために重要となる点の一つが、Map と Reduce の分離である。Map 処理と Reduce 処理はもともと独立した処理であり、これらを自由に組み合わせたワークフローとして、データ処理を記述することで、より多くのアプリケーションへ適用することが可能になる。

しかし、広く利用されている MapReduce 処理系 Hadoop[2] は、この条件を満たしていない。Hadoop は、HDFS と呼ばれるファイルシステムをベースとし、Map への入力および Reduce からの出力は HDFS へ書き出すが、Map と Reduce の間の通信に関しては、高速化のためローカルファイルシステムへ一時ファイルとして書き出す。このように Map と Reduce が 1 対 1 に対応することを前提として構成されているため、複雑なワークフローを記述することが難しい。空の Mapper、Reducer を用いることで擬似的にワークフローを表現することはできるが、実行が非

効率となる。

これに対してわれわれは分散 KVS(キーバリューストア) をベースに構成した MapReduce 処理系 SSS を提案している [3][4]。SSS では Map と Reduce は、いずれも分散 KVS からデータを読み込み、分散 KVS に書き込む。このため Map と Reduce は、データの流れとしては全く同等であり、これらを自由に組み合わせることが可能となっている。したがって、これらを組み合わせたワークフローを容易に構成、実行することができる。

本稿では、この SSS の実装を詳述する。また、合成ベンチマークプログラムおよび、K-means によって SSS を Hadoop と比較した評価を示す。評価の結果、SSS は Hadoop よりもほとんどの場合において高速であることが確認できた。

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 について概説する。3 節で SSS の設計と実装について述べる。4 節にベンチマークプログラムと K-means による評価を示す。5 節で考察を行う。6 節に関連研究を示す。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。

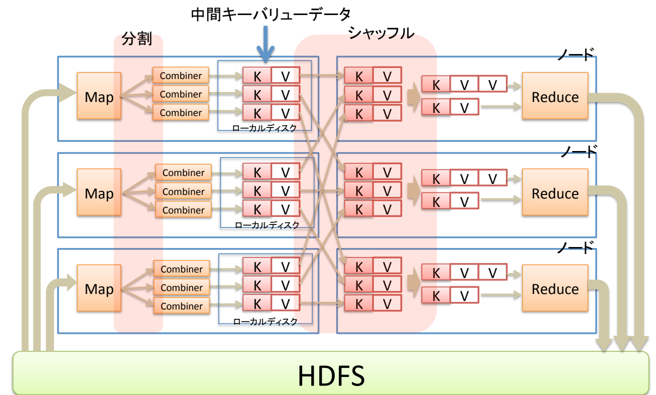

図 1　Hadoop でのデータの流れ

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、データの流れに着目した動作図を図 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、分割されたデータブロックごとに、キーについてソートされる。ソートされたデータブロックはさらに 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データブロック

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。この過程が Hadoop におけるシャッフルである。（この結果得られた）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ができない。したがって、ひとつの Mapper が複数の出力を行い、複数の Reducer に接続するといった操作を記述することができない。また、Mapper のみ、Reducer のみを動作させることができない。空の Reducer を組み合わせることで、Mapper のみの動作を擬似的に実現することはできるが効率は低下する。

3　SSS の設計と実装

SSS[4] は、われわれが開発中の MapReduce 処理系である。本節では SSS の設計と実装について述べる。

3.1　SSS の概要

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。

分散 KVS は、複数の単体 KVS をハッシュ関数で使い分けることによって実現する。つまりキーのハッシュ値に応じて、書きこむ単体 KVS を変更することで、複数のノードに分散された一体としての KVS を構成する。

さらに SSS では、Owner Computes ルールにしたがって計算を行う。すなわち、各ワーカノードは、計算ノードと単体 KVS を兼ねており、各ノード上の Mapper/Reducer は自ノードの単体 KVS 内のキーバリューペアのみを対象として処理を行う (図 2)。

したがって、Map/Reduce 実行時、入力データに関しては、ネットワーク転送が全く発生しない。これによって、計算開始時のデータ転送の時間を削減し、ネットワーク上でのデータの衝突による、予想の難しい性能低下を未然に防ぐことができる。一方、出力時には、分散した KVS に対する書き込みが行われるため、ノード間のデータ転送が発生するが、出力時のデータ転送レイテンシの影響は入力時のレイテンシに比べて性能低下に影響することは少ない。

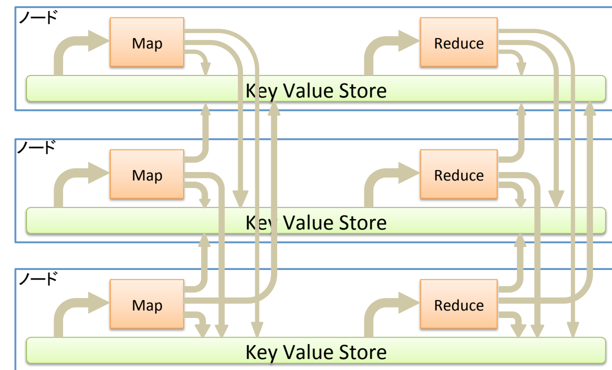

図 2　SSS におけるデータの流れ

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分散 KVS に保存されるため、Map と Reduce が 1 対 1 に対応している必要がない。このため、任意個数、段数の Map と Reduce から構成される、複雑なワークフロー構造を容易に表現し、実行する事が可能である。

3.2　SSS におけるシャッフル

SSS の Mapper は生成したキーバリューペアを分散 KVS に書きこむ。この際、キーに対するハッシュで書き込まれる単体 KVS が決定されるため、同一キーのキーバリューペアは同一のノード上の単体 KVS に書き込まれることが保証される。単体 KVS に書きこまれたデータは、単体 KVS の機能によって、自動的にキー毎にグループ分けされる。これでシャッフルが完成する。

つまり SSS においては、シャッフルは、キーのハッシングによる単体 KVS の決定と、単体 KVS 内でのキー毎のグループ分けで実現されることになる。この機構は、基盤となる分散 KVS の基本的な機能を自然に活用したものであり、実装の容易化に貢献している。

3.3　キースペース

SSS では、データの空間をキースペースと呼ぶサブ空間に分割する。個々の Map/Reduce 処理の演算対象は個々のキースペースとなる。一つの演算への入力は常に一つのキースペースとなるが、出力は複数となり得る。図 3 上段に示すものが一般的な MapReduce である。Map と Reduce

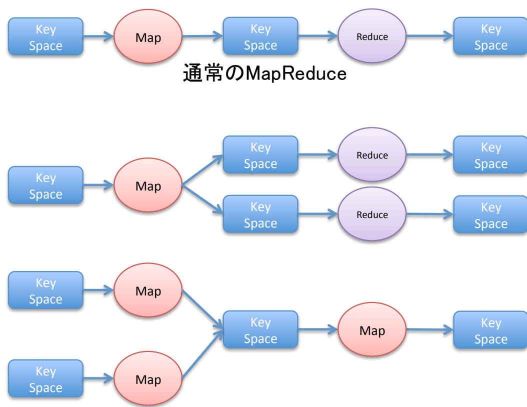

図 3　キースペースと Map/Reduce

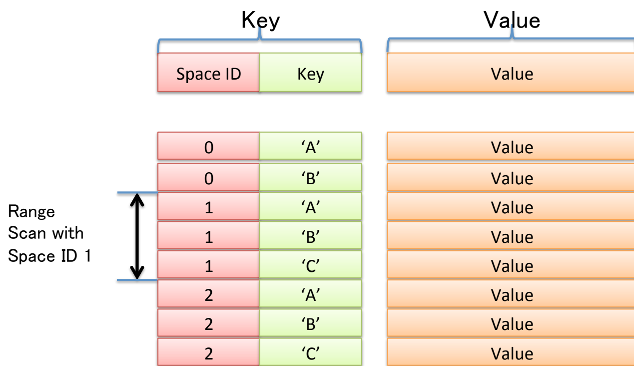

図 4　キースペースの実装

の間は、キースペースで接続されている。図 3 下段に、Map と Reduce で構成されるより複雑なワークフローの例を示す。SSS では、このように複数の Map と Reduce を自由に組み合わせてワークフローを構成することが可能である。

キースペースの実装手法の概念図を図 4 に示す。キースペースはそれぞれユニークな SpaceID を持つ。これを KVS のキーへのプレフィックスとすることで、KVS の提供するフラットな空間にマップする。

これによってキースペースの要素の取り出しは、キーに対するレンジスキャンとして実装することができ、高速に読み出すことができる。

3.4　SSS の構成

SSS の構成を図 5 に示す。各ノード上では、SSS サーバと単体 KVS のサーバとが動作する。SSS サーバは、Mapper、Reducer を実行するサーバである。単体 KVS サーバの集

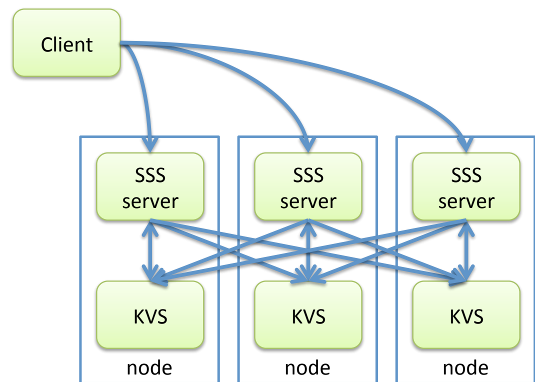

図 5　SSS の構成

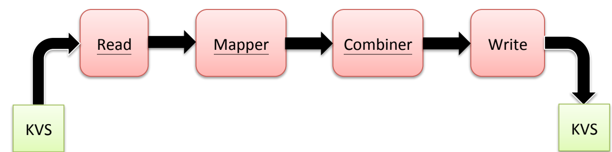

図 6　SSS サーバのパイプライン

合が全体として、分散 KVS を構成する。1 つのノード上の SSS サーバと KVS サーバは特別な対応関係にある。すなわち、SSS サーバ上の Mapper/Reducer は、対応するローカルな KVS サーバからしかデータを読み込まない。これに対して出力はすべての KVS サーバに対して行う。

クライアントプログラムは 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 サーバに対してジョブの実行を指示する。

クライアントプログラム、SSS サーバはともに Java 言語で記述されており、Java RMI (Remote Message Invocation) を用いて通信を行う。

3.5　SSS サーバのパイプライン

SSS サーバ内部は、図 6 に示すようなパイプライン構成を取っている。読み出し部、Mapper 部、Combiner 部、書き出し部はそれぞれ独立した複数のスレッドで動作し、その間をサイズ制限付きのキューで接続している。

このような構成を取ることにより、読み出しと書き出しの I/O 処理と、計算本体を重複して行うことができ、I/O のレイテンシをある程度隠蔽することができる。

3.6　分散 KVS の実装

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 5 に示したとおり、単

表 1 Benchmarking Environment

# nodes	17 (*)
CPU	Intel(R) Xeon(R) W5590 3.33GHz
# CPUs/node	2
# Cores/CPU	4
Memory/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

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 評価

本節では SSS と Hadoop の比較評価を行う。評価プログラムとしては、I/O 性能を評価するための合成ベンチマークと、K-means 法によるクラスタリングを用いた。

4.1 評価環境

評価には、表 1 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。

4.2 MapReduce 合成ベンチマーク

I/O 性能を評価するための合成ベンチマークとしては先行研究 [8][9] で提案した合成ベンチマークを用いた。

4.2.1 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。

4.2.2 ベンチマークの構成

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。

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。

表 2　I/O ワークロードの設定

	Read Intensive	Write Intesive	Shuffle Intensive
Map 入力キー数	16 - 65536	16	16
Map 入力バリューサイズ	1GiB - 256KiB	1	1
Map 出力キー数	256	4194304	16 - 262144
Map 出力キーユニーク数	16	16 - 65536	16
Map 出力バリューサイズ	1	1	1GiB - 64KiB
Reduce 出力キーサイズ	1	1GiB - 256KiB	1
Combiner 有る無し	false	false	false | true

SSS においては一つのキーバリューペアを、分散 KVS 上のキーバリューとしてそのまま展開している。Hadoop では、1 ファイルあたり 64MiB を上限に集約している。[*1]

4.2.3　ベンチマーク結果

4.2 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。

4.2.4　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。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。

■Read Intensive　図 7 に Read Intensive ワークロードの結果を示す。まず、Hadoop では InitialKeyCount（入力キー数）にかかわらず性能が安定している。

これに対して SSS では InitialKeyCount が小さい領域で速度が低下している。この理由は、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。

■Write Intensive　図 8 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、

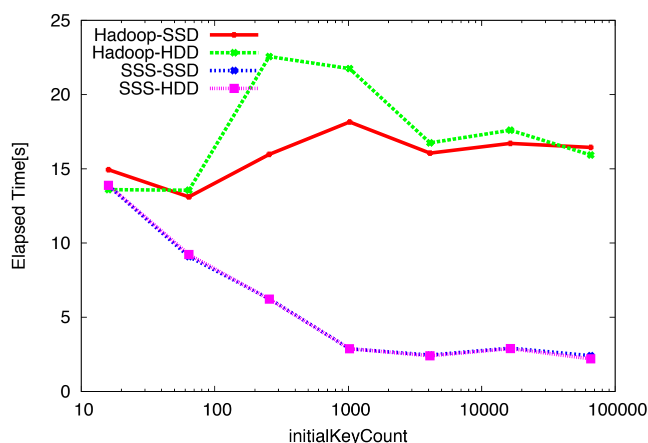

図 7　Read Intensive

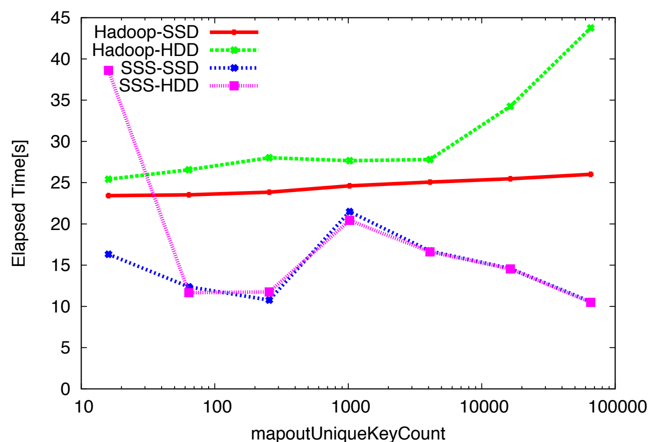

図 8　Write Intensive

有意な差は見られない。SSS での実行は 1024 近辺にピークがあるが全般に安定している。Hadoop では、mapoutUniqueKeyCount が 4096 以上の HDD で急速な性能の悪化が見られる。SSD ではそのような現象はみられない。

[*1] 文献 [9] に示した結果と大きく異なるのはこのためである。

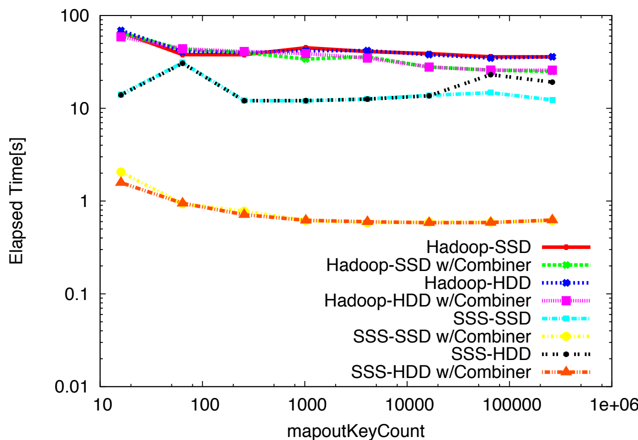

図 9　Shuffle Intensive

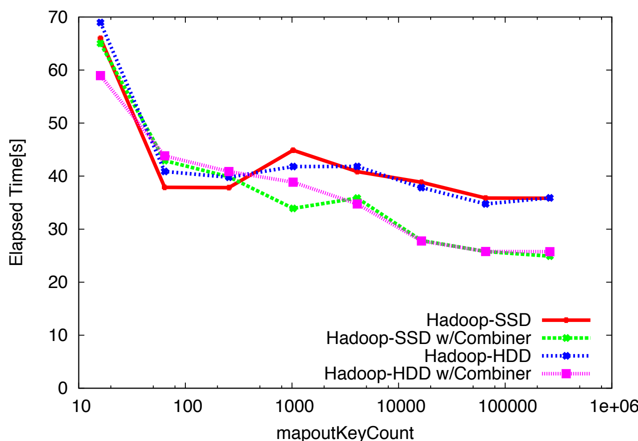

図 10　Shuffle Intensive(Hadoop)

■Shuffle Intensive　図 9 に Shuffle Intensive ワークロードの結果を示す。このグラフは縦軸も対数となっていることに注意されたい。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。図 10 に Hadoop のみの結果を抜き出したものを示す。このグラフは縦軸が対数表記となっていないことに注意されたい。Hadoop では、mapKeyCount が 1024 以上で combiner によって 30% ほど性能が向上している。ことが読み取れる。

4.3　K-means による評価

K-means 法は、大規模なデータを与えられた数のクラスタに分割するためのアルゴリズムである。

4.3.1　アルゴリズムの概要

K-means では、クラスタ中心を繰り返し更新することでクラスタ分割を行う。

まず、すべてのデータ点とクラスタ中心の距離を計算し、最寄りの中心で表されるクラスタに分類する。つぎに、この時点で計算されたクラスタ（データ点の集合）の重心をもとめ、これを新たなクラスタ中心とする。最初ののクラスタ中心はランダムに生成する。

演算としては、クラスタ中心が移動しなくなるまで実行を繰り返す収束演算である。また、各イタレーションで、すべての点とクラスタ中心の距離を演算することとなる。

MapReduce での実装では、データ点集合を Map への入力データとし、クラスタ中心集合をサイドデータとしてあたえた。Map 関数では、各データ点に対して、すべてのクラスタ中心との距離を計算する。Reduce 関数ではこの結果を総合して新たなクラスタ中心を算出する。

新たなクラスタ中心の集合が、前回のクラスタ中心の集合と同一だった場合、計算が収束したことになるので、繰り返しを停止する。

4.3.2　評価方法

独自に実装した K-means を用いて評価を行った。対象となるデータは、2 次元の点とし、距離としてユークリッド距離を用いた。計測対象は、1 イタレーションあたりの時間とした。

4.3.3　結果

結果を図 11 に示す。横軸縦軸ともに対数となっている点に注意されたい。点の数は、256Mi 点、1Gi 点、4Gi 点とした。データ量はこの 16 倍の 4GiB, 16GiB, 64GiB 点となっている。いずれの場合も SSS が高速であることが見て取れる。

5　議論

5.1　全般的な実行時間に対するジョブ起動オーバヘッドの寄与

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。Hadoop でのジョブ起動時間は、ノード数にもよるが 10 数秒かかる。

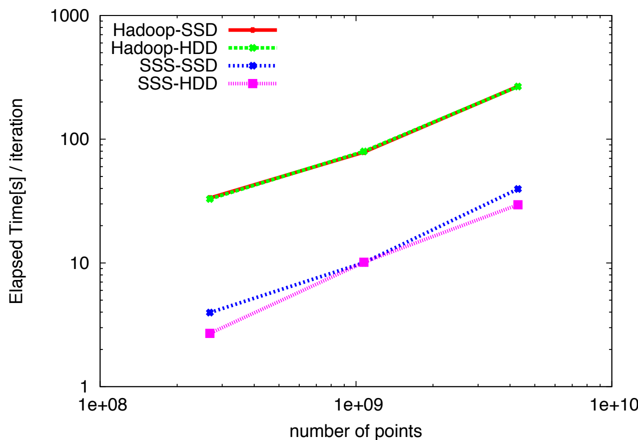

図 11　K-means

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。タスクの個数はつねにノード数に等しい。また、タスクはサーバの JavaVM 内で起動するため、起動のコストは非常に小さい。

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。

5.2　Read/Write の実行時間

上記の起動オーバヘッドを差し引いて考えても、SSS の実行時間は短い。これは、3.5 で述べた、パイプライン化の効果であると考えられる。

Hadoop の各タスクはシングルスレッドで動作し、データを読み出し、処理し、データを書きだす。データ読み出し、書き出しなどの I/O 処理には時間がかかるため、その間処理ができないことになる。

これに対して SSS では、I/O 処理とデータ処理が独立したスレッドで行われるため、可能な場合にはこれらが重畳して実行される。このため、実行時間が全体として短くなる。

5.3　Shuffle Intensive での Combiner の効果

Shuffle Intensive では、SSS では Combiner が大幅な効果を上げたのに対し、Hadoop では 30 %程度の実行時間短縮にとどまっている。

これは前述のジョブ起動オーバヘッドのために見かけ上効果が少なく見えているのではないかと考えられる。このため、実行時間が底上げされてしまい、実際の実行時間短縮の貢献が小さく見えている。

5.4　K-means の実行時間

K-means では、Mapper で大量のデータを読み出し、計算は Mapper,Reducer ともに軽微であり、Reducer の出力は非常に小さい。したがって、プログラムの動作としては、合成ベンチマークの Read Intensive に類似するといえる。

データ量は計測点のいずれの場合にも十分に大きい。したがって、図 7 の InitialKeyCount が大きい領域の動作に類似すると予想される。K-means での実験結果はこの予想と一致するものである。

6　関連研究

SSS と同様の問題意識に基づいて設計された並列処理系に Sphare [10][11][12] がある。Sphare は分散オブジェクトストアである Sector 上に構築されており、Sector 上のデータに対する処理を UDF(User Defined Function) として記述することでデータストア上で演算が行われると言うモデルであ。この UDF に Mapper もしくは Reducer 操作を記述することによって、MapReduce 処理系としても利用することが可能である。Sphare はユーザの記述可能な範囲が広い反面、ユーザへのプログラミング負荷が大きい点が SSS と異なる。

7　おわりに

分散 KVS をベースとした MapReduce 処理系 SSS に対して、合成ベンチマークと K-means を用いた性能評価を行い、代表的な MapReduce 処理系である Hadoop と比較した。その結果、SSS は Hadoop と比較してほぼすべての場合において高速であることがわかった。特に、K-means においては 10 倍近い高速化が見られた。

今後の課題としては、より大規模な実アプリケーションで実証的な評価を行うことがあげられる。

また、速度向上の原因をより詳細に解析することも今後の課題である。

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] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[2] Hadoop. http://hadoop.apache.org/.

[3] 小川宏高, 中田秀基, 美田晃伸, 広渕崇宏, 高野了成, 工藤知宏. 高速フラッシュメモリに適したキーバリューストアの予備的評価. 情報処理学会研究報告 2010-HPC-124, 2010.

[4] Hirotaka Ogawa, Hidemoto Nakada, Ryosei Takano, and Tomohiro Kudoh. Sss: An implementation of key-value store based mapreduce framework. In *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.

[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.

[8] 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.

[9] 小川宏高, 中田秀基, 工藤知宏. 合成ベンチマークによる mapreduce 処理系 sss の性能評価. 情報処理学会研究報告　2011-HPC-130, 2011.

[10] Sector/Sphare. http://sector.sourceforge.net/.

[11] Yunhong Gu and Robert Grossman. Sector and Sphere: the design and implementation of a high-performance data cloud. Vol. 367, No. 1897, pp. 2429–2445, June 2009.

[12] *Processing Massived Sized Graphs using Sector/Sphere*, November 2010.