Vol.2012-HPC-133 No.18
2012/3/26

MapReduce処理系SSSのPrefixSpan法による評価

中田秀基[†1]　小川宏高[†1]　工藤知宏[†1]

MapReduce プログラミングモデルの幅広いアプリケーションへの適用をめざし、より柔軟で複雑なワークフロー実行を可能にする MapReduce 処理系 SSS を開発している。SSS は代表的な MapReduce 処理系である Hadoop とは大きく構成が異なり、したがって性能的特性も大きく異なる。われわれは SSS の有効なアプリケーション領域を知るべく、合成的ベンチマークでの評価とともに実アプリケーションでの評価を行っている。本稿では、先行研究で述べた PrefixSPan 法による系列パターン抽出の改良と、疎行列ベクトル積について述べる。評価の結果、いずれのアプリケーションにおいても SSS は Hadoop よりも高性能であることが確認できた。

An Evaluation of MapReduce middleware SSS

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

We have been developing a KVS based MapReduce System called SSS to provide flexible platform for broader application area. SSS differs a lot in architecture from Hadoop, the most widely used MapReduce system, so do in the performance behavior. To precisely know the applicable area, we are performing a series of benchmark evaluations including prefixSpan method. This paper describes improvement of the prefixSpan implementation, and a new application; sparse matrix-vector multiplication. From the results, we confirmed that SSS outperforms Hadoop in all the test we performed.

1. はじめに

MapReduce[1] は大規模なデータ処理向けのプログラミングモデルである。特に MapReduce の代表的な実装である 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。

Hadoop は非常に大規模なデータに対して、一組の Map と Reduce を実行することに特化した設計となっている。一方、MapReduce の普及に従って用途が拡大し、比較的小規模な MapReduce を多数繰り返すアプリケーションや、Hive[3] や Jaql[4] などクエリ言語のバックエンドとして MapReduce のワークフローを利用するシステムも登場している。このようなケースに関しては Hadoop は必ずしも適していない。Hadoop は、ジョブ起動のコストが大きく、繰り返し処理には適さない。また、Map と Reduce が強固に結合しており、Map と Reduce の間の中間データを再利用することもできない。

このような観点から、われわれはイタレーションが高速で柔軟なワークフローの構成が可能な MapReduce システム SSS[5],[6] を開発している。これまでに、合成ベンチーマーク[7]、およびクラスタリングアルゴリズム K-means を用いた評価[6] とともに、実アプリケーションである PrefixSpan 法による評価[8] を行なってきた。しかし文献[8] で用いた PrefixSpan 法は不要なデータ入出力が多く、実アプリケーションの評価としては理想的ではないことがわかった。本稿では入出力データ量を大きく減らした新たな PrefixSpan 法の実装を紹介し、それを用いて SSS の評価を行う。さらに、新たなアプリケーション事例として疎行列ベクトル積を取り上げる。

本稿の構成は以下のとおりである。2 節で、MapReduce およびその実装である Hadoop と SSS について概説する。3 節で PrefixSpan 法を概説する。4 節では疎行列ベクトル積の MapReduce での実装について述べる。5 節に評価結果を示す。6 節はまとめである。

2. MapReduce と Hadoop と SSS

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

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

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。

2.2 Hadoop

Hadoop は、代表的なオープンソースの MapReduce 処理系である。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。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。

2.3 SSS

SSS[5] は、われわれが開発中の MapReduce 処理系である。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。

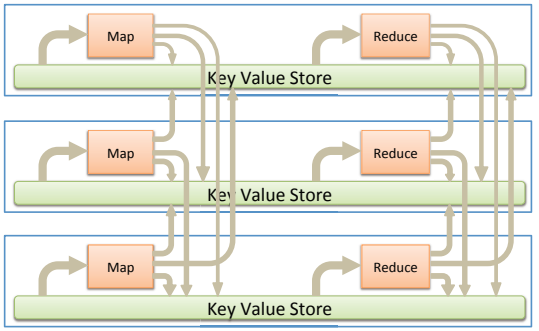

図 1 SSS におけるデータの流れ

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。

2.3.1 SSS の構成

SSS の構成を図 2 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。

2.3.2 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 2 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[9] を、SSS が多用するソート済みデータのバルク書

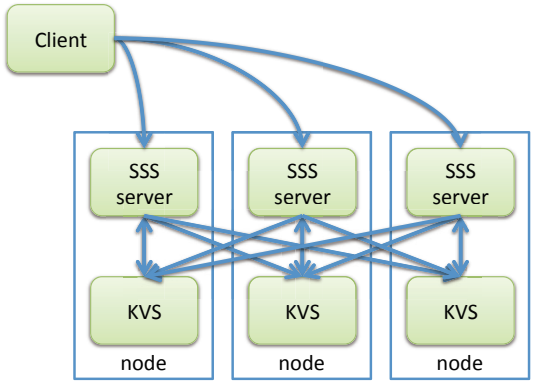

図 2 SSS の構成

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[10] を、クライアント側に jTokyoTyrant[11] を用いている。

3. PrefixSpan 法による系列パターン抽出

本節では、PrefixSpan 法[12] による系列パターン抽出について概説する。系列パターン抽出とは、データマイニングの一つで、与えられた系列の集合からそのなかに閾値以上の回数出現するパターンを抽出することである。

系列パターン抽出にはさまざまな応用が考えられる。例えば、顧客の購買パターン解析によるプロモーション、過去の診療履歴データの解析による診断、Web ログストリーム解析、遺伝子解析などである

系列パターン抽出にはいくつかアルゴリズムが知られている。PrefixSpan 法はその一つである。

3.1 PrefixSpan 法

PrefixSpan 法は、まず短いパターンを見つけ、閾値以上に頻出するものだけを長さ 1 ずつ拡張していくアルゴリズムである。まず候補となるサブパターンを**列挙**する。次に頻出するサブパターンのみを抜き出す。これを**限定**と呼ぶ。また、入力列からサブパターンに後続する可能性のあるあらたな後続列を作り出す。この操作を**射影**と呼ぶ。

図 3 に動作の概要を示す。この例では、入力系列データ **abcc**, **aabb**, **bdbc**, **bdc** から、

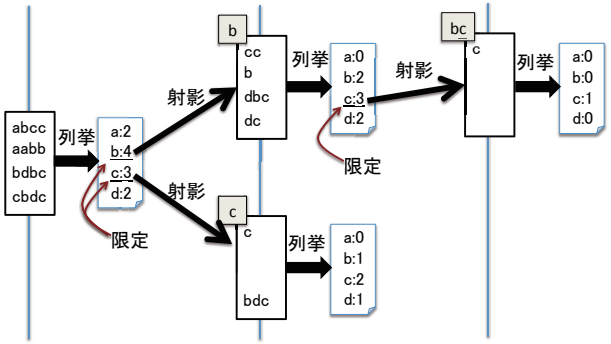

図 3 PrefixSpan 法の概要

3 回以上の頻度で出現するものを抽出している。

まず、長さ 1 のパターンを列挙する。つぎに 3 回以上出現しているパターンのみを選ぶ（限定）し、選択したパターンについて射影を行い後続列を作っている。この例では、**b** と **c** が 3 度以上出現しているので、これらから始まるより長いパターンを探している。上段では **b** で始まるパターンを探すために、**b** の後続列を作成している。下段では **c** を扱っている。

上段では、後続列に対して再度列挙を行う。すると **c** が 3 度出現しているのでこれのみを選択し、射影を行う。次の列挙では、3 度以上出現しているものがないので、ここで操作は終了する。下段も同様である。結果としては、入力系列データに 3 度以上出現したパターンは、**b**(4 回)、**c**(3 回)、**bc**(3 回) の 3 つであることがわかる。

このように、PrefixSpan 法では、データを複製しながら探索空間の分割を繰り返し、それ以上進めなくなった時点で終了する。

3.2 PrefixSpan 法の MapReduce による実装

井上ら[13] は、PrefixSpan を MapReduce で実装する手法として *s-EB*,*p-BE*,*s-BE* の 3 つの方法を提案している。以下に *s-EB*、*s-BE* および *s-EB* を改良した *s-EB PBI* について詳述する。

3.2.1 s-EB

この手法では、map で列挙と射影を行い、reduce で限定を行う。MapReduce の入力となるデータは系列データそのものである。アルゴリズム上は、射影は限定後に限定されたサブパターンのみに対してのみ行えばよい。しかし、この手法では射影を先に行うため、あと

で限定操作の際に捨ててしまうサブパターンに対しても、射影操作を行うことになる。また、限定前に射影を行ったデータが Map Reduce 間で受け渡されるため、Map Reduce 間のデータ転送量が大きい。一方で、アルゴリズム上の 1 反復が 1 回の MapReduce で行われるため、MapReduce 起動オーバヘッドは小さい。図 4 上段に、s-EB 法におけるデータ入出力量の模式図を示す。青がキー部分を、赤がバリュー部分を示す。s-EB 法では Map の出力が非常に大きいことが特徴となる。

3.2.2 s-BE

この手法は、アルゴリズム上の 1 反復を 2 段の MapReduce で行うことで余分なデータ処理・転送を削減する。1 段目の MapReduce で列挙と限定のみを行う。つまりある系列が指定された回数以上登場するかどうかのみを判定する。Map Reduce 間で転送されるキーバリューのバリューは整数値 1 つのみで、s-EB と比較するとデータ量が格段に小さい。2 段目の MapReduce では、限定された系列のみに対して射影を行う。このため、計算量を低く抑えることができる。一方 MapReduce の回数はアルゴリズム上の反復回数の 2 倍となるため、MapReduce 起動のオーバヘッドが大きい。図 4 中段に、s-BE 法におけるデータ入出力量の模式図を示す。2 回の MapReduce を行うがトータルのデータ入出力量は S-EB 法にくらべて小さい。

3.2.3 s-EB PBI

s-EB 法、s-BE 法はともに、初期系列データの一部をコピーして持ちまわるように実装されている。しかし、初期系列データの集合はメモリに乗る程度に十分に小さいため、コピーを行う代わりに系列番号と系列内でのインデックスを持ちまわれば、処理に十分な情報が得られる。

この知見に基づき、s-EB-PBI（Projection by Index）法を実装した。これは S-EB の亜種で基本的な動作は S-EB 法と同じであるが、系列データの一部をコピーする代わりに、系列番号と系列内インデックスを出力する。初期系列データは、MapReduce の入出力ではなく、サイドデータとして引き渡す。図 4 下段に、s-EB PBI 法におけるデータ入出力量の模式図を示す。動作としては s-EB 法とほぼ同じで各 Map、Reduce の生成するキーバリューペアの個数もおなじであるが、バリューのサイズが小さいため、入出力データ量は常に小さい。

3.3 PrefixSpan 法によるプログラムソースコードからのパターン抽出

PrefixSpan 法の具体的な応用として、プログラムソースコードからのコーディングパターンの抽出を用いた[14]。コーディングパターンとはイディオム的に用いられる一連の処理で

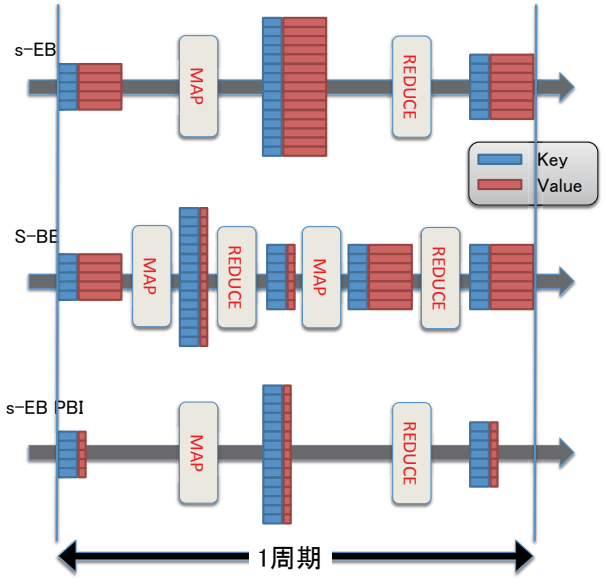

図 4 PrefixSpan の実装

ある。本稿では文献[14] に従う。

まず、解析対象のソースコードをメソッド単位に分割し、正規化ルールを適用することで、メソッドを要素列に変換する。正規化することで、while ループと for ループなどの機能的に等価な構文を同一のものとして扱うことができる。メソッドを要素列とすることで、ソースコード全体から一連の正規化された要素列のデータベースを得る。このようにして得られた要素列データベースに対して PrefixSpan 法を適用することで、コーディングパターンの抽出を行う。

4. 疎行列ベクトル積

疎行列に対してベクトルをかけ合わせベクトルを得る疎行列ベクトル積は応用が広い。例えば Web ページの重要性をページ間のリンクの重みで定義する PageRank[15] は疎行列ベクトル積をベクトルが収束するまで行うことで計算できることが知られている。

非常に大規模な行列を対象とした MapReduce での疎行列ベクトル積の手法としては、行列を column に分割したものと対応するベクタの要素を合わせたものを入力とし、Map で

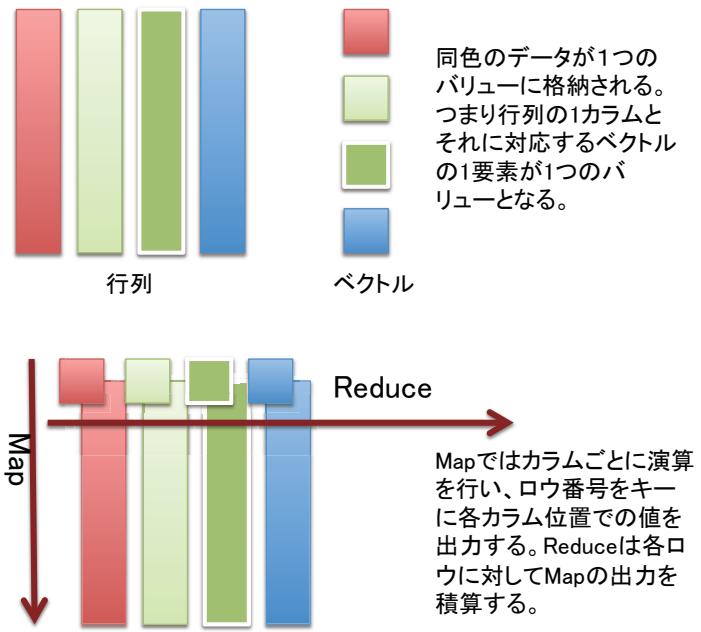

図 5 超大規模疎行列ベクトル積

各要素に対する乗算を行い row ごとの値を出力、Reduce で raw ごとの集計を行う手法が知られている[16](図 5)。行列の要素は Reduce 計算には必要ではないが、収束までを連続的に行う場合、次のイタレーションに行列の要素を引き継がなければならないため、ベクタの要素とあわせて Reduce に引き継ぎ、Reduce から出力する。このため、入出力データ量が大きくなる問題がある。

これに対して、ベクトルのサイズが Map 時にメモリに乗る程度のサイズである場合には入出力データ量を抑えた手法が考えられる。行列は row 方向に分割し、これを Map の入力とする。ベクトルはサイドデータとして入力する (図 6)。この手法では、入力データサイズは前述の手法とほぼ同じであるが、行列を出力する必要がないため出力データサイズは小さくなる。さらに、Map で乗算と加算の双方を実行することができるため、Reduce フェイズでは演算を行う必要はない。本稿ではこの手法を用いて評価を行う。

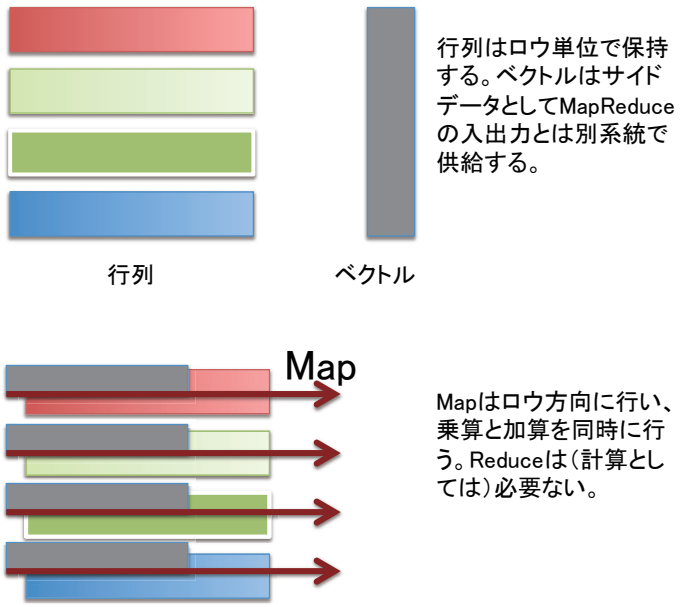

図 6 ベクトルがメモリに乗る場合の大規模疎行列ベクトル積

4.1 SSS での実装

前述のように、この手法では Reduce フェイズが不要となる。SSS では Map のみのジョブが構成できるので、Reduce の存在しないワークフローとして実装を行った。図 7 にワークフローを示す。Map からの出力は、次段の Map の入力ではなくサイドデータとなる。Map の入力は常に行列である。

4.2 Hadoop での実装

Hadoop では Reduce のないジョブを構成することができない。このため、今回の実装では Reduce でデータ量の削減を行う。

Map からの出力では、ベクトルをインデックス (int) と値 (double) のペアで表現する。このため 1 要素あたり 12 バイトが必要となる。しかしインデックスが昇順であることを前提にすれば、各要素にインデックスを付加する必要はなく、1 要素あたり 8 バイトで表現する事が可能となる。本実装ではベクトルを 64 のフラグメントに分割する。Map からの出力時に、どのフラグメントに属するべきかを計算し、フラグメント ID をキーとし、バリュー

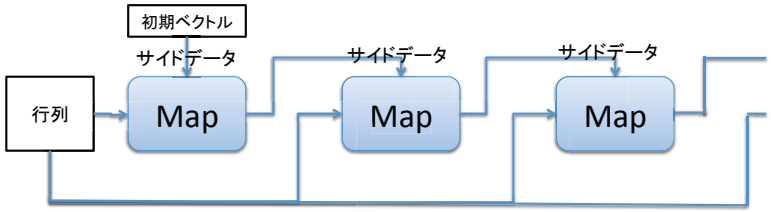

図 7 疎行列ベクトル積のワークフロー

表 1 Benchmarking Environment

# nodes	17 (*)
CPU	Intel(R) Xeon(R) W5590 3.33GHz
# CPU per node	2
# 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

にインデックスと値を格納する。Reduce では Map からの出力をもちいてフラグメントを構成し、出力する。

5. 評　　価

5.1 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 を備えているが、基本的に全ての計測は ioDrive を用いて行った。Hadoop のバージョンは 0.20.2、レプリカ数は 1 としている。

5.2 PrefixSpan

s-EB 法、s-BE 法、s-EB PBI 法をそれぞれ SSS, Hadoop で実行した結果を表 2 に示す。対象データとしては、総計約 1M バイト、2M バイト、3M バイト、4M バイトのソースコード集合を用い、これを正規化、記号化した列を入力とした。記号化された入力データのバイト数はそれぞれ 72K バイト, 136K バイト, 200K バイト、277K バイトである。s-EB 法では 4M のデータに対しては中間データの量がストレージ容量をオーバーし、実行を完了することができなかった。

表 2 PrefixSpan の実行時間 [s]

	s-EB		s-BE		s-EB PBI	
	SSS	Hadoop	SSS	Hadoop	SSS	Hadoop
1M	41.8	835.0	74.2	844.6	11.9	278.6
2M	109.6	1173.5	150.4	1068.1	44.2	351.5
3M	339.4	1632.4	327.5	1341.6	123.2	456.9
4M	N/A	N/A	8888.5	12372.4	4460.5	*

(* 本稿執筆時点でデータの計測が間に合わず)

この表から、手法としては s-EB PBI が Hadoop、SSS をとわず常に高速であることが確認できる。これは入出力されるデータ量が少ないからである。s-EB 法と s-BE 法を比較すると、対象データ量が小さい場合には s-EB 法が有利だが、大きい場合には s-BE 法が有利であることがわかる。これは s-EB 法はジョブ起動回数が少ないため、ジョブ起動によるコストがドミナントな領域では優位だが、データ入出力量は大きくなる傾向にあるため、対象データ量が大きくデータ入出力による時間がドミナントな領域では不利となるためである。

また、Hadoop と SSS を比較すると、常に SSS が高速である。特にデータサイズが小さい際にはその差が顕著であるが、その差は主にジョブ起動のオーバヘッドからくるものである。たとえば、s-EB PBI 法の 1M データでは 25 倍程度の差が付いているように見えるが、MapReduce ジョブは 22 回起動されており、それぞれに 12 秒程度がかかることを考えると、278.6 秒の実行時間のほとんどがジョブ起動に費やされていることがわかる。このことから、SSS の軽量なジョブ起動が処理時間の短縮に貢献していることが確認できる。

5.3 疎行列ベクトル積

疎行列の次元数を 4Mi, 16Mi, 64Mi, 256Mi と変化させ、計測を行った。行列はランダムに生成し、各コラム/ロウあたりの非ゼロ要素は 10 とした。計測は複数回のループで行い、ループの 1 反復の平均時間を実行時間とした。1 ループ目は他のループと大きく挙動が異なるため除外した。図 8,9 にそれぞれ SSS を用いた実装と Hadoop を用いた実装の結果を示す。図中の赤はサイドデータとしてのベクトルの読み出しにかかる時間、緑はループの最後に前回の出力ベクトルを消去するのにかかった時間である。青はそれ以外の時間を表す。

256Mi 次元の場合で比較すると、SSS での 198 秒に対して Hadoop では 418 秒以上の時間がかかっている。SSS では、サイドデータでのベクトルの入力時間が 129 秒と、全実行

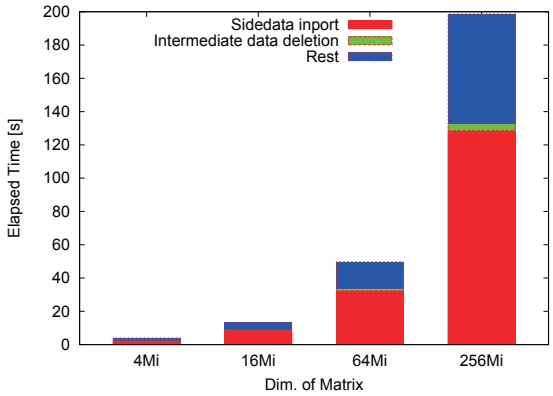

図 8 SSS での実行時間

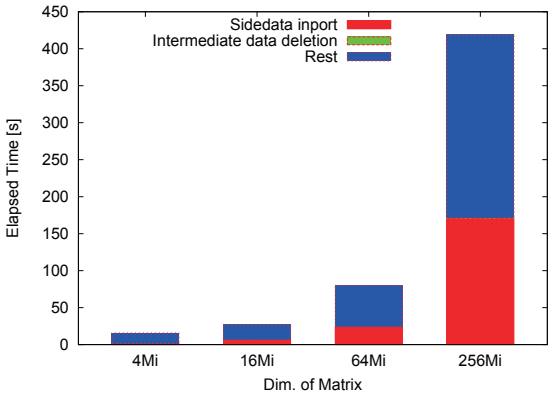

図 9 Hadoop での実行時間

時間 198 秒の過半を占める。これはサイドデータが複数のノードに分散しており、ローカルに読み出すことができないためであると考えられる。Hadoop でもサイドデータの入力時間が 171 秒と非常に大きい。

削除時間に関しては、SSS では最大 4.4 秒かかっているのに対して、Hadoop では 3 ミリ秒と遥かに小さくグラフには反映されないほどとなっている。これは、SSS では一連のキーバリューを一括して消していく操作となるが、Hadoop ではディレクトリのメタデータを書き換えるだけで削除できるためである。

サイドデータ入力時間、削除時間を省いた時間でも SSS が優位である。これは Reduce を省略した効果であると思われる。

6. おわりに

KVS を基盤とする MapReduce 処理系 SSS を、PrefixSpan と疎行列ベクトル積とで評価し、Hadoop と比較した。また、PrefixSpan に関しては中間データのサイズを削減する手法を提案した。この結果、PrefixSpan, 疎行列ベクトル積共に、SSS が Hadoop の性能を上回ることが確認できた。

今後の課題としては、より広範囲なリアルアプリケーションに対して SSS を適用し、SSS の特性を明らかにすることがあげられる。

謝　辞

PrefixSpan 法のプログラムおよび入力データは、大阪大学井上研究室ならびに萩原研究室の井上佑希氏、置田助教、萩原教授にご提供いただきました。深く感謝の意を表します。

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) Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H. and Murthy, R.: Hive - A Petabyte Scale Data Warehouse Using Hadoop, *ICDE 2010: 26th IEEE International Conference on Data Engineering* (2010).
4) : jaql: Query Language for JavaScript(r) Object Notation, `http://code.google.com/p/jaql/`.
5) Ogawa, H., Nakada, H., Takano, R. and Kudoh, T.: SSS: An Implementation of Key-value Store based MapReduce Framework, *Proceedings of 2nd IEEE In-*

ternational 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).
6) 中田秀基，小川宏高，工藤知宏：分散 KVS に基づく MapReduce 処理系 SSS，インターネットコンファレンス 2011 (IC2011) 論文集，pp.21–29 (2011).
7) 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).
8) 中田秀基，小川宏高，工藤知宏：MapReduce 処理系 SSS の実アプリケーションによる評価，信学技報, Vol.111, No.255, CPSY2011-25-CPSY2011-41，pp.55–60 (2011).
9) FAL Labs: Tokyo Cabinet: a modern implementation of DBM, `http://fallabs.com/tokyocabinet/index.html`.
10) FAL Labs: Tokyo Tyrant: network interface of Tokyo Cabinet, `http://fallabs.com/tokyotyrant/`.
11) : Java Binding of Tokyo Tyrant, `http://code.google.com/p/jtokyotyrant`.
12) Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.-C.: PrefixSpan Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth, *Proceedings of 17th International Conference on Data Engineering*, pp. 215–226 (2001).
13) 井上佑希，置田真生，萩原兼一：系列パターン抽出の MapReduce 実装におけるタスク分割方式の検討，情報処理学会研究報告　2011-HPC-130 (2011).
14) 伊達浩典，石尾　隆，井上克郎：オープンソースソフトウェアに対するコーディングパターン分析の適用，ソフトウェアエンジニアリング最前線 2009 (2009).
15) Page, L., Brin, S., Motwani, R. and Winograd, T.: The PageRank citation ranking: Bringing order to the Web, Technical Report Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University (1999).
16) Lin, J. and Dyer, C.: *Data-Intensive Text Processing With MapReduce*, Morgan and Claypool Publishers (2010).