社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

PrefixSpan 法の MapReduce 実装の改良

中田　秀基† 　井上　辰彦†,†† 　小川　宏高† 　工藤　知宏†

† 独立行政法人 産業技術総合研究所
〒 305–8568 茨城県つくば市梅園 1 − 1 − 1 中央第二
†† 株式会社創夢
〒 151–0072 東京都世田谷区幡ヶ谷 1-34-14
E-mail: †{hide-nakada,tatuhiko-inoue,h-ogawa,t.kudoh}@aist.go.jp

あらまし　分散キーバリューストアをベースとし、Owner Compute ルールで計算を実行することで、高速な繰り返し処理を可能とする MapReduce 処理系 SSS を開発している。この SSS の評価の一つとして、PrefixSpan 法による系列パターン抽出を実世界アプリケーションとして利用して来た。しかし、既存の手法では大規模なデータに対しては十分な絶対性能が得られていなかった。本稿では PrefixSpan を MapReduce で実装するための新たな手法を提案する。提案手法ではデータの流れを見直すことによって、これまで Reduce で行なっていた処理を、Map に移すことによって大幅な速度の向上を得た。具体的には 4M のソースコードのデータに対して SSS で約 60 倍、Hadoop で約 3 倍の高速化を実現した。

キーワード　MapReduce, 分散並列計算, キーバリューストア, 系列パターンマイニング, PrefixSpan

Improvement of MapReduce implementation of PrefixSpan Method

Hidemoto NAKADA†, Tatsuhiko INOUE†,††, Hirotaka OGAWA†, and Tomohiro KUDOH†

† National Institute of Advanced Industrial Science and Technology (AIST)
Tsukuba Central 2, 1-1-1 Umezono, Tsukuba, Ibaraki 305-8568, JAPAN
†† Soum Corporation,
1-34-14 Hatagaya, Setagaya-ku, Tokyo, 151-0072, JAPAN
E-mail: †{hide-nakada,tatuhiko-inoue,h-ogawa,t.kudoh}@aist.go.jp

Abstract　We have been implementing a Key-Value Store based MapReduce System, called SSS, which enables quick MapReduce iteration by employing Owner-Compute Rule. We have been employing sequential pattern mining using PrefixSpan method as an evaluation target. The performance so far was not satisfactory for large sized data, however. This paper proposes a new implementation technique for PrefixSpan method over MapReduce. In the proposed method, we moved bound operation from Reducer to Mapper, eliminating data transfer cost between Mapper and Reducer. As a result we confirmed that the new technique showed 60 times speedup for SSS and 3 times speedup for Hadoop.

Key words　MapReduce, Distributed Parallel Execution, Key Value Store, Sequential Pattern Mining, PrefixSpan

1. はじめに

近年広く用いられつつある MapReduce は、大規模データ処理に適したプログラミングフレームワークである。われわれは、分散キーバリューストアをベースとし、Owner Compute ルールで計算を実行することで、高速な繰り返し処理を可能とする MapReduce 処理系 SSS [1] [2] を開発している。

MapReduce による大規模データ処理応用の一つとして、PrefixSpan 法による系列データからのバターン抽出がある ([3])。これは、膨大な量のイベント列の中から、頻出している生起順序のイベント列を抽出するもので、実世界に幅広い応用があることが知られている。われわれは、この PrefixSpan 法を用いて、SSS を評価するとともに、PrefixSpan 法の効率的な MapReduce 実装を進めてきた [4] [5]。

本稿では、われわれが新たに開発した、より効率の良い PrefixSpan 法の MapReduce 実装について述べる。この実装では、従来 Reducer で行なっていた処理を Mapper で行うことにより、ボトルネックとなっていた Mapper・Reducer 間でやり取

りされるデータの書き出しと読み込みを省いた。これにより、大幅な高速化を実現することができた。

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 SSS について述べる。3. 節で、PrefixSpan 法による系列パターン分析手法と既存の MapReduce 上での実装について述べる。4. 節で新たな MapReduce 上の実装の提案を行う。5. 節で提案手法と既存の手法を比較する。6. 節はまとめである。

2. MapReduce と SSS

2.1 MapReduce

MapReduce [6] 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。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。

Map と Reduce の間には、シャッフルと呼ばれるフェイズがあり、Map が出力するキーバリューペア全てに対してキーごとに集約を行う。ここで、分散ノード間での通信が発生する。分散計算の立場から考えると、MapReduce は通信のパターンをシャッフルに限定することで、モデルを単純化し、記述性を向上させたモデルであると考えることができる。

2.2 SSS

SSS [1] は、われわれが開発中の MapReduce 処理系である。SSS は Hadoop [7] における 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 (図 1)。

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。

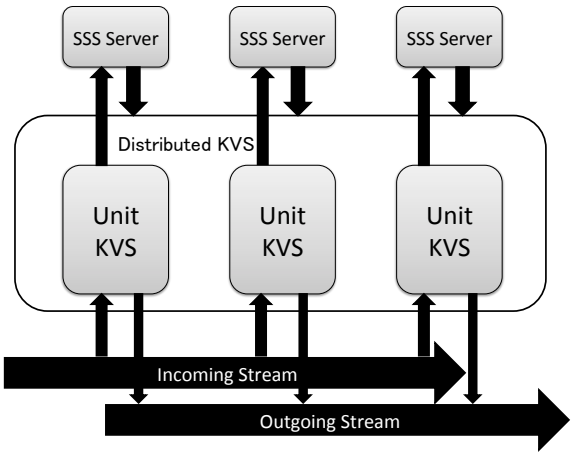

図 1　SSS の概要

SSS の分散 KVS は、各ワーカノードに設置される要素 KVS の集合から構成される。データの分散には、キーによる単純なハッシュを用いた。キーバリューペアを書き込む際、書き込みノードでキーのハッシュを取り、そのハッシュ値に基づいて書きこむノードを決定する。すべての SSS サーバがハッシュ関数を共有しているので、同じキーをもつキーバリューペアは同じ要素 KVS に書き込まれることが保証される。

3. PrefixSpan 法による系列パターン抽出

本節では、PrefixSpan 法 [3] 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 2 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、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。

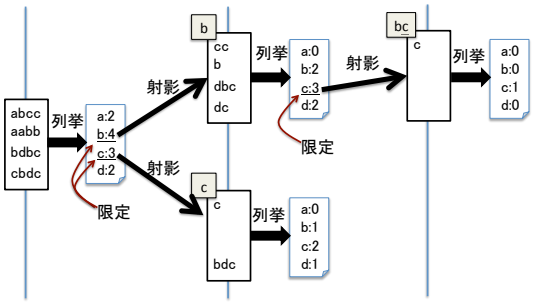

図 2　PrefixSpan 法の概要

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 [8]。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ある。本稿では文献 [8] 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。

3.3　PrefixSpan 法のパラメータと計算量

PrefixSpan 法の計算量、入出力データ量は、入力データセット以外に、主に 2 つのパラメータによって決定される。ひとつは、発見したい系列パターンの最低長である。もう一つは、最低出現頻度である。たとえば、長さ 4、頻度 10 の場合、長さ 4 以上で 10 回以上現れた系列パターンを出力する。

このうち、長さの計算に与える影響は比較的小さい。長さが影響するのは、各ステージで発見したパターンを出力する部分のみで、ステージ間でやり取りされるデータ量、計算量には影響はない。

一方、頻度の影響は非常に大きい。頻度は限定操作で用いられる。最低頻度が大きければ、限定操作の際に切り捨てられるデータ量が大きく、したがって生き残るデータ量が小さくなる。

本稿では、長さ 4、頻度 10 をパラメータとして用いる。

3.4　PrefixSpan 法の MapReduce による実装

井上ら [9] 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。

3.4.1　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 3 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。

3.4.2　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 3 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。このため、データセットが大きくなり、中間データが非常に大きい場合には、s-EB 法よりも高速である場合がある。

3.4.3　s-EB PBI

s-EB 法、s-BE 法はともに、初期特徴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。個々の系列データのサイズは小さいが、重複して持つことになるためデータ量は莫大になる。しかし、初期特徴列データ全体の集合はメモリに乗る程度に十分に小さい。このため、特徴列データのコピーを行う代わりに、特徴列番号と特徴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、我々は s-EB-PBI（Projection by Index）法を提案した ([4])。s-EB-PBI は S-EB の亜種で基本的な動作は S-EB 法と同じであるが、特徴列データの一部をコピーする代わりに、特徴列番号と特徴列内インデックスを出力する。初期特徴列データは、MapReduce の入出力ではなく、サイドデータとして引き渡す。

Map,Reduce の入出力となるキーバリューペアは、キーが候補系列（整数のリスト）、バリューが特徴列番号（整数値）と特徴列内インデックス（整数値）から構成されるタプルとなる。

図 3 下段に、s-EB PBI 法におけるデータ入出力量の模式図

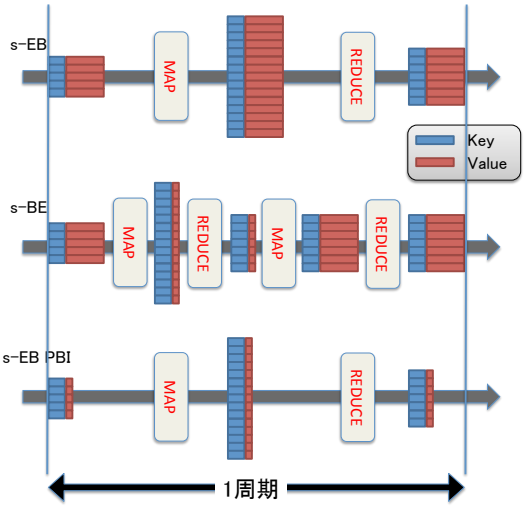

図 3　PrefixSpan の実装

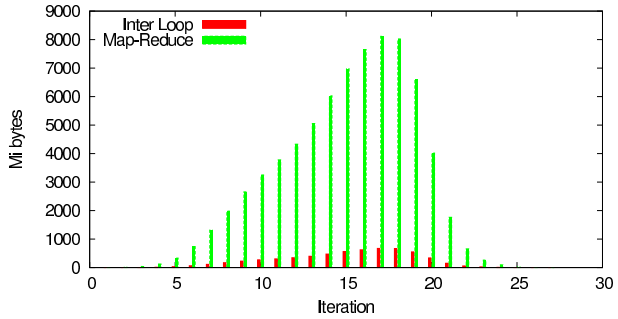

図 4　s-EB 法によるデータ量

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。

3.4.4　各手法でのデータ量

各手法でソースサイズ 3M のデータを処理した場合のデータ量を図 4, 図 5, 図 6 に示す。横軸は繰り返しのループを表している。ループ回数はいずれの場合も 27 回で終了しているが、初期と末期はデータ量が極端に小さくなるのでプロットされていない。縦軸はデータ量で Mbyte 単位である。まず、いずれの場合も Map-Reduce 間のデータ転送がドミナントであることがわかる。

最もナイーブな s-EB 法 (図 4) では、最大となる 17 ループ目においては 8Gbyte 程度のデータが入出力されている。これに対して s-BE 法 (図 5) では、ピーク時でも 1.2GByte 程度とデータ量が抑えられている。一方 MapReduce の回数が 2 倍となっているため、入出力されるデータ量の総計にはそれほど大きな差は無い。

一方 s-EB PBI 法 (図 6) は、グラフの形としては s-EB 法 (図 4) と酷似しているが、絶対値は大きく低減されている。ピーク時で 1.7Gbyte と 4 分の 1 以下となっている。

4.　s-EB IMR 法

4.1　In-Mapper Reducer 法の提案

3.4.4 で述べたように、s-EB PBI 法では、Map-Reduce 間でやりとりされるデータは s-EB 法よりは小さくなったものの

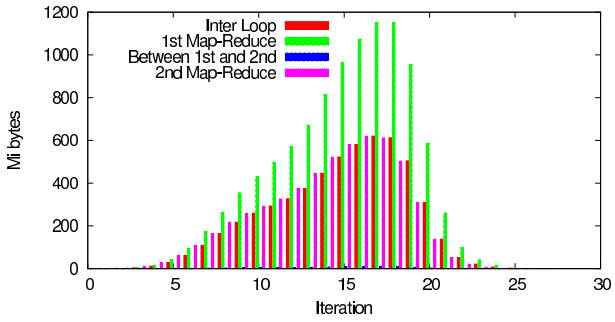

図 5　s-BE 法によるデータ量

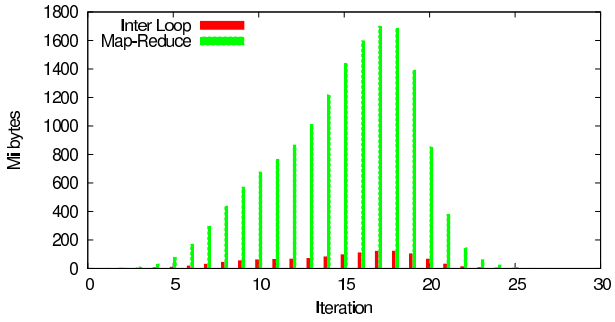

図 6　s-EB PBI 法によるデータ量

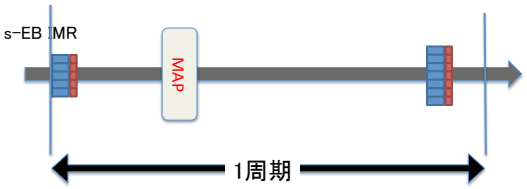

図 7　s-EB IMR

まだまだ大きく、この入出力が律速となることが予想される。

われわれは、このデータ量を削減すべく、s-EB PBI 法の計算とデータの流れを詳細に再検討した結果、Reducer で行なっている処理を Mapper に移すことが可能であることに気がついた。

例として、特徴は *a*,*b*,*c* が存在するとして、特徴列候補 *abc* を処理することを考える。s-EB PBI 法では、Map 内で *abca*, *abcb*, *abcc* の 3 つの候補系列とそのインデックスを見つけ、出力し、Reduce 内でそれぞれの個数を数え、限定操作を行う。しかし、*abca*, *abcb*, *abcc* の系列はいずれも *abc* を処理している Map 以外で生成されることはありえない。したがって限定操作を Map 内で実行しても、結果が変わることはない。問題があるとすれば、Map で出力せずメモリ上に貯めこんで限定操作するため、メモリが余分に必要になることであるが、4.4 で見るように、実質的には問題にならない。

我々はこの手法を s-EB IMR （In-Mapper Reduce）法と名付けた。この手法を実現するためには、Map にある特徴列候補のインデックス情報が同時にすべてわたらなければならないため若干データ構造を変更する必要がある。具体的にはキーバリューペアのキーは候補特徴列で同じであるが、バリューにすべてのインデックス情報のリストを持つ。

この手法の動作を図 7 に示す。1 周期が 1 度の Mapper で実現されている。

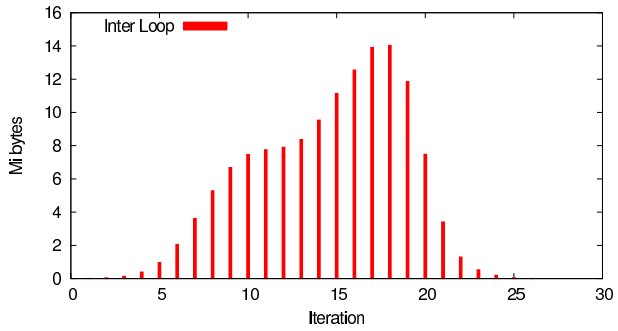

図 8　s-EB IMR 法によるデータ量

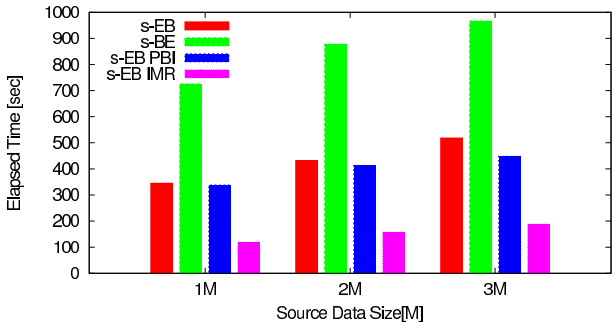

図 9　Hadoop による実行結果

4.2　s-EB IMR 法によるデータ量

図 8 に、入力データサイズ 3 M とした場合の、s-EB IMR 法動作時にやり取りされるデータ量を示す。s-EB IMR 法では、Mapper のみが動作するため Map-Reduce 間の通信が存在しない。また、ループ間でやり取りされるデータ量も非常に小さい事がわかる。

4.3　In-Mapper Combining との関連

Combiner の機能を Mapper 内部に実装し、Mapper からの出力を減らす手法は、In-Mapper Combining 法として知られている [10]。ここで用いた手法は、この手法の変形であると考えることができる。つまり、他の Mapper から同一のキーが出力されることはないというアプリケーションドメインの知識を用いて、In-Mapper Combiner で Reducer を置き換えている。

4.4　メモリ上のデータ量に関する考察

s-EB IMR 法では、1 つの候補系列に関する情報を 1 つのキーバリューペアで表現する。このため、バリューが大きくなる可能性がある。また、Mapper のなかで次の候補系列群の列挙と限定を連続的に行うため、列挙した候補系列群をオンメモリで持つ必要がある。本項ではこれらのデータ量について考察する。

まずバリューの最大値について考える。バリューは、候補系列の次のインデックスの集合である。候補系列は、1 つの特徴列につき最大 1 つであるため、候補系列集合の最大サイズは、特徴列の数に等しい。また、インデックスは 2 つの整数（特徴列番号と、列中の位置）で表現できる。したがって、バリューの最大サイズは、特徴列の数 * 整数のサイズ * 2 となる。次節で用いる最大のデータサイズである 4M のケースでは特徴列は 6160 個あるため、$6160 * 4bytes * 2 = 49280bytes$ となる。

次にオンメモリで持つ候補系列群のサイズについて考える。Mapper は候補系列の次の末尾値となる特徴 ID ごとに、上で考えたバリューを保持する。したがって、候補系列群サイズの最大値は 特徴 ID の種類の数*特徴列の数*整数のサイズ*2 となる。上述の 4M のケースでは、特徴 ID の種類は 5226 通りとなるため、$5226*6160*4bytes*2 = 257,537,280bytes \approx 246Mibytes$ となる。これは最悪のケースで通常はこれよりもはるかに小さくなる。また、最悪のケースであっても十分に小さい。

5.　評　　価

5.1　評 価 環 境

評価には、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。CPU は Interl Xeon(R) W5590 x 2, ノードあたりのメモリ量は 48GB となっている。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。今回の実験では、すべて ioDrive を用いている。

評価は、SSS と Hadoop で行った。Hadoop は CDH3 Update5 に含まれる hadoop-0.20.2+923.418 を用い、各ノードでの reducer 数は 8 としている。

5.2　対象データ

対象データとしては、総計約 1Mbyte、2Mbyte、3Mbyte、4Mbyte 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 byte 数はそれぞれ 64Kbyte, 128Kbyte, 194Kbyte, 270Kbyte であった。

また、発見対象のデータは長さ 4 以上、頻度 10 以上とした。この設定では、出力結果が膨大になり、これをシステムから取り出すのに時間がかかるため、以下の結果では、出力結果をシステムから取り出す時間は含めていない。

5.3　結　　果

1M から 3M の場合の結果を示す。図 9 は Hadoop、図 10 は SSS である。いずれの場合も提案手法である s-EB IMR 法が突出して高速であることがわかる。

また、SSS と Hadoop を比較すると、とくに SSS での高速性が目を引く。これは、データの処理コストが低減されたため、繰り返し処理の際のジョブ起動コストがドミナントになったためであると考えられる。Hadoop では 1 度の起動につき 10 秒程度かかるのに対して、SSS では 1 秒未満である。このため SSS が有利になったと考えられる

4M のデータを加えたグラフを図 11 に示す。レンジが広く読み取りづらいので表 1 にも同じデータをまとめた。ここでは s-EB, s-BE 法は掲載していない。これらの手法では 4M のデータを処理すると非常に時間がかかり、計測が難しかったためである。

いずれの場合でも s-EB IMR 法が高速であることが確認できる。また、SSS と Hadoop を比較すると、ほぼすべてのケー

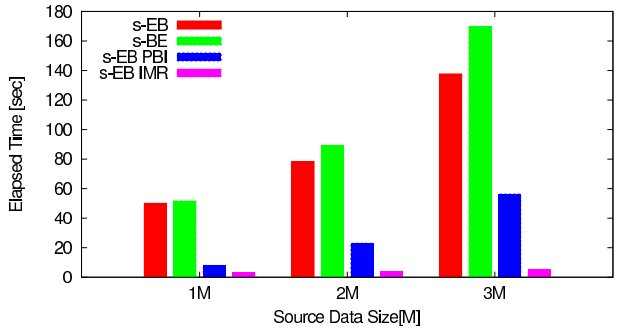

図 10　SSS による実行結果

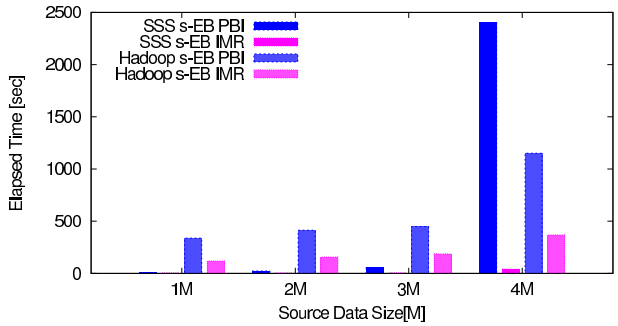

図 11　4M を含めた実行結果

表 1　実行時間 (s)

	1M	2M	3M	4M
SSS s-EB PBI	8	23	56.3	2402.3
SSS s-EB IMR	3	3.7	5.3	40.7
Hadoop s-EB PBI	338	413.3	448.3	1151.3
Hadoop s-EB IMR	119	157.3	188	368.3

スで SSS が高速であるが、s-EB PBI 法の 4M の場合にのみ、SSS 2402 秒、Hadoop 1151 秒と大きく遅れを取っている。これは、非常に細粒度のデータが大量に読み書きされることになるためである。S-EB IMR 法では、データの総量が大きく減っていることに加え、キーに対してインデックスのリストをバリューとしてもたせることでデータが粗粒度となっている。このため、SSS が非常に高速となっているのであると思われる。

6. おわりに

本稿では PrefixSpan を MapReduce で実装するための新たな手法を提案した。これまで Reduce で行なっていた処理を、Map に移すことによって、4M のソースコードのデータに対して SSS で約 60 倍、Hadoop で約 3 倍の高速化を実現した。

今後の課題としては、実世界で意味のある系列データ抽出への適用が挙げられる。本稿で示した結果は、 長さ 4 頻度 10 の系列データである。しかし、4Mbyte のソースコードにのパラメータを適用すると、4300 万通りの系列パターンが発見されてしまう。これは、データマイニングのコンセプトから考えると、明らかに妥当ではない。より大規模なデータセットに対して、より少量の結果を出力するパラメータを用いた評価を行う必要がある。

また、本稿で述べた手法では、本質的に幅優先探索を探索領域を区切って行なっていることになり、繰り返し処理の間で情報を交換するのは負荷の均等な分散のためにすぎない。このように考えると、PrefixSpan の 1 イタレーションを、1 回の MapReduce にマップする必要は無いのではないかとも考えられる。たとえば、1 回の MapReduce の間に複数回の PrefixSpan イタレーションを行うスタイルの実装が考えられる。Hadoop のように起動コストが大きい処理系では効果が高いのではないかと考えられる。

さらに、Haloop [11] のように、ループ処理に特化した処理系が存在する。このような処理系での本手法の評価も今後の課題である。

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．

文　　献

[1] 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).

[2] 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).

[3] 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).

[4] 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).

[5] 中田秀基, 小川宏高, 工藤知宏: MapReduce 処理系 SSS の PrefixSpan 法による評価, 情報処理学会研究報告 2011-HPC-133 , 2012 (2012).

[6] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (2004).

[7] : Hadoop, `http://hadoop.apache.org/`.

[8] 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).

[9] 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).

[10] Lin, J. and Dyer, C.: *Data-Intensive Text Processing With MapReduce*, Morgan and Claypool Publishers (2010).

[11] Bu, Y., Howe, B., Balazinska, M. and Ernst, M. D.: HaLoop: Efficient Iterative Data Processing on Large Clusters, *Proc. of VLDB'10: The 36th International Conference on Very Large Data Bases*, pp. 24–30 (2010).