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.

MapReduce処理系SSSの実アプリケーションによる評価

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
〒 305–8568 茨城県つくば市梅園 1 − 1 − 1 中央第二
E-mail: †{hide-nakada,h-ogawa,t.kudoh}@aist.go.jp

あらまし　高速で高機能な MapReduce 処理系の実装を目指し、分散キーバリューストアを用いて Owner Computes ルールに基づいた計算を行う軽量な MapReduce 処理系 SSS の開発を行なった。これまで、合成ベンチマークや簡単なアプリケーションのコアを用いたベンチマークによる評価の結果、Hadoop に対して大幅な高速化を確認したが、実世界のアプリケーションでの性能の検証は行われていなかった。本稿では PrefixSpan 法による系列パターン抽出を対象として SSS の評価を行った。この結果、SSS は全般に Hadoop と比較して高速であり、さらにノード数増大による速度向上の余地があることが確認できた。

キーワード　MapReduce, 分散並列計算, 系列パターン抽出

An Evaluation of a Mapreduce System SSS Using a Real-World Application

Hidemoto NAKADA†, 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
E-mail: †{hide-nakada,h-ogawa,t.kudoh}@aist.go.jp

Abstract　We have been implementing a MapReduce system called SSS aiming at highspeed and highly capable MapReduce framework. SSS is implemented on distributed Key-Value store and allocate computation based on Owner Compute model. So far, we have evaluated SSS using synthetic benchmarks and application core benchmark and confirmed that it is fater than Hadoop for most situations, but the performance for real-world applications are not proven. In this paper, we evaluated SSS using a real-world application to mine sequential patterns from large dataset using PrefixSpan method. The results showed that SSS outperforms Hadoop in general and SSS still has room to improve performance by adding worker nodes while Hadoop does not have.

Key words　MapReduce, Distributed Parallel Execution, Sequential Pattern Mining

1. はじめに

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として MapReduce [1] が注目されている。特に、MapReduce 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], [4] を開発している。SSS は分散 Key Value ストア（KVS）を基盤とすることで I/O と計算をパイプライン的に実行し、高速化を実現している。

われわれはこれまでに剛性ベンチーマーク [5]、およびクラスタリングアルゴリズム K-means を用いた評価 [4] を行い、SSS が Hadoop と比較して高速であることを確認した。しかし、実アプリケーションでの性能は明らかになっていない。

本稿では、実アプリケーションである PrefixSpan 法による系列パターン抽出を SSS で実装し、その性能を Hadoop と比較する。PrefixSpan 法は MapReduce の繰り返しで構成されるため、

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 PrefixSpan 法による評価と Hadoop との比較を示す。5. 節はまとめである。

2. MapReduce と Hadoop、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 [3] 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。

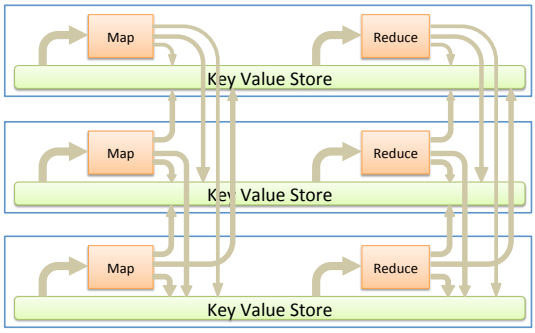

図 1　SSS におけるデータの流れ

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。

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 [6] 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

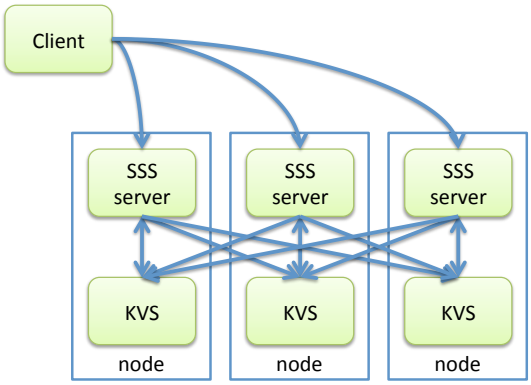

図 2　SSS の構成

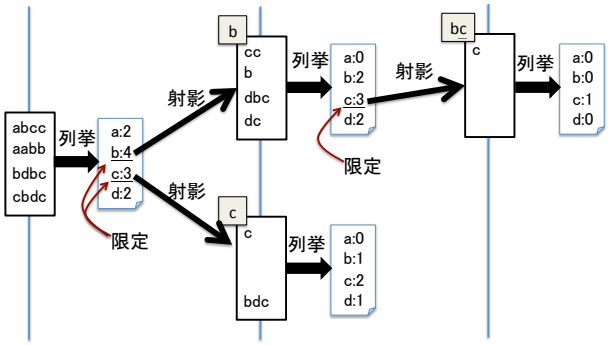

図 3　PrefixSpan 法の概要

Tyrant [7] を、クライアント側に jTokyoTyrant [8] 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 [9] 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 法では、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。このアルゴリズムでは、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、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 [10] 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。

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 MapReduce 間で受け渡されるため、データ転送量が大きい。

一方で他の 2 つの手法と比較すると、アルゴリズム上の 1 反復を 1 回の MapReduce で行うことができるため、オーバヘッドが小さい。

3.2.2　p-BE

この手法は他の 2 つとは全く異なり、存在する可能性のある長さ 1 のパターンを列挙したものを入力データとする。つまり、各 map では、特定のパターンにのみ着目して系列データを処理する。アルゴリズムの入力となる系列データは、サイドデータとして供給される。

利点は各 map 処理の負荷が完全に均等化されることである。一方、欠点も多い。まず、mapper がすべての系列データを読みこむため、長さ 1 のパターンの個数を n とすると、読み込むデータ量の総計が n 倍となる。また、map タスク、reduce タスクの個数が常に n 個となり、n が大きい場合には、タスク管理のオーバヘッドが大きくなる。

3.2.3　s-BE

この手法は、s-EB 法と類似するが、アルゴリズム上の 1 反復を 2 段の MapReduce で行う。1 段目の MapReduce で列挙と限定のみを行い、2 段目の MapReduce で射影を行う。

この手法の利点は、余分な射影操作を行わないため、計算負荷が比較的小さいこと、不必要なデータの入出力がないことである。一方 MapReduce の回数は反復回数の 2 倍となるため、オーバヘッドが大きい。

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

3.2.4　手法の選択

本稿では *s-EB* 法に基づいた MapReduce 化を行った。これは、文献 [10] において、この手法がもっとも高速であることが報告されているからである。

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 [11]。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 [11] 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.　評　　価

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。今回の実験では、すべて ioDrive を用いている。

Hadoop のバージョンは 0.20.2、レプリカ数は 1 としている。利用ノード数に対する性質を見るために、ノード数を 4,8,16 と変化させた。(注1)

4.2　対象データ

対象データとしては、総計約 1M バイト、2M バイト、3M 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 バイトである。

(注1)：Hadoop では、全体のワーカノード数を固定の 16 としたまま、map ワーカ数、reduce ワーカ数を 4,8,16 とした。

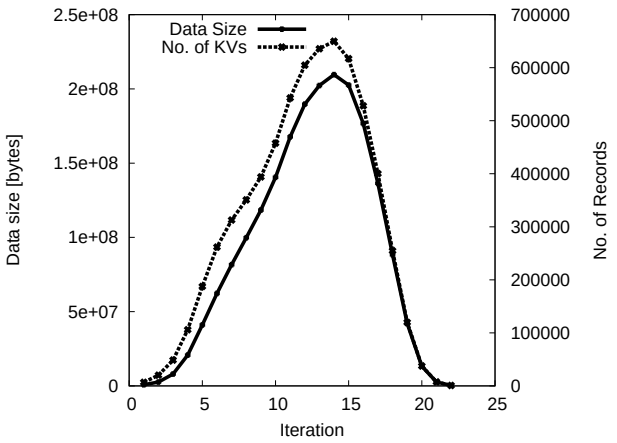

図 4　中間データ (1M)

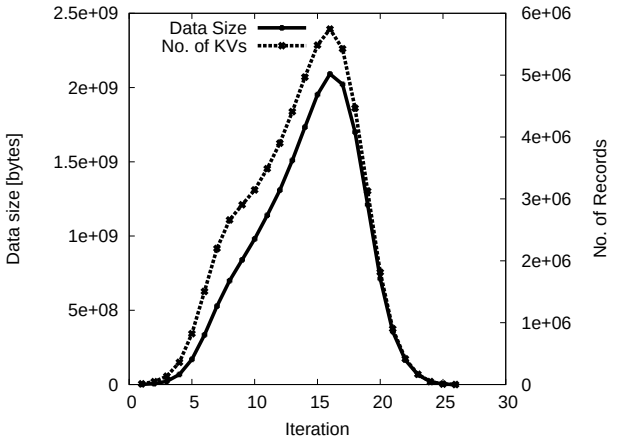

図 5　中間データ (2M)

4.3　中間データ量

本プログラムにおいては、各回における MapReduce の MapReduce 間で受け渡されるデータ量が大きく、システムに負荷を与えることが予想される。プログラムの挙動を理解するために、まずそれぞれのデータサイズにおいて、中間データの量を、バイト数とキーバリューペアの数の側面で測定した。図 4,5,6 に結果を示す。横軸は MapReduce の繰り返し段数である。

いずれの場合もほぼ結果は一致しており、中盤に中間データ量が爆発的に増加することが見て取れる。中間データバイト数は、入力データ量と強い相関があり、3M の場合には最大となる 17 回目では、2246 万キーバリューペア、データ量 8G バイト弱のデータが入出力されている。

このように中盤にデータ量が増大するのは、個々の候補パターンに対してそれぞれ射影時に入力系列の一部をコピーするため、候補パターンが豊富な前半において、指数級数的にキーバリューペア数が増大するためである。一方後半においては、新たに射影するべき入力系列が尽きてくるため、急速に減衰する。

4.4　実 行 時 間

実行時間を図 7,8,9 に示す。横軸は使用したワーカノード数である。いずれの場合も SSS の実行時間が Hadoop のそれを

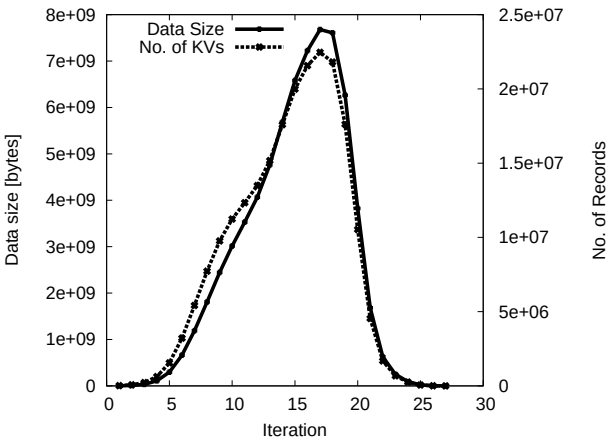

図 6　中間データ (3M)

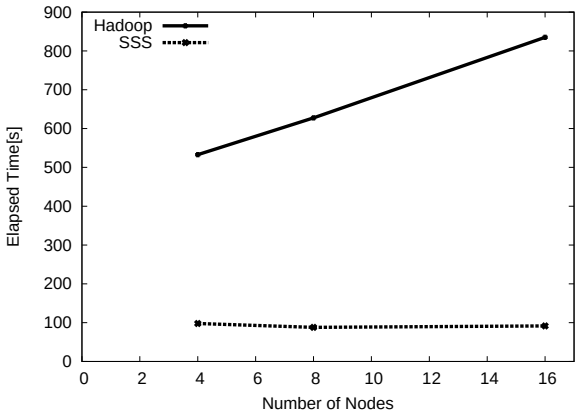

図 7　実行時間 (1M)

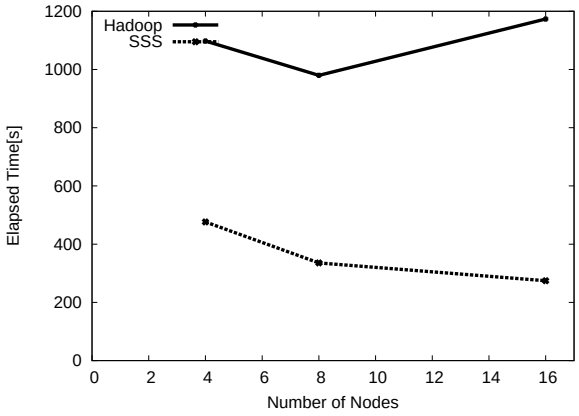

図 8　実行時間 (2M)

大きく下回っている。

Hadoop に着目すると、ノード数の増大が実行時間の低減に寄与していない。1M, 2M では 16 ノードがもっとも遅い。3M でも 8 ノード時の性能が 16 ノードを上回る結果となっている。これは、prefixSpan の処理がおもに I/O インテンシブであり、ノードの追加による計算処理性能の増大が必ずしも性能の向上に直結しないことによると思われる。

一方 SSS では、16 台の範囲ではノード数の増大にともない性能が向上している。

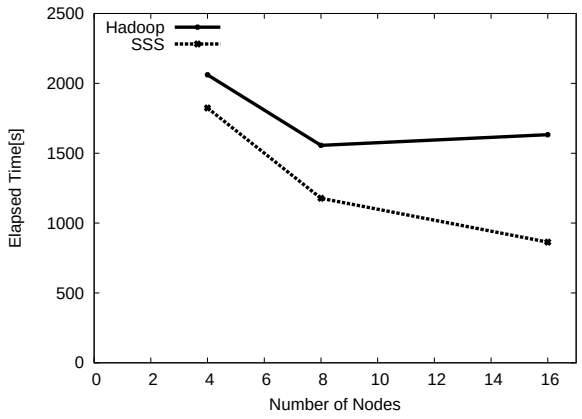

図 9　実行時間 (3M)

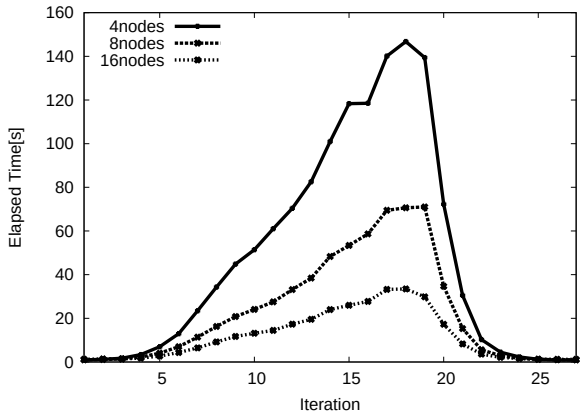

図 10　各繰り返しの実行時間 (3M)

図 10 に、入力データが 3M の際の各繰り返しでの SSS での実行時間を示す。中間データのサイズに応じて実行時間が変化していることがわかる。

4.5 議　　論

Hadoop と SSS の際だった相違点の 1 つは、MapReduce ジョブ起動のオーバヘッドである。Hadoop は空の MapReduce ジョブの起動にすら 20 秒弱程度を要する。これはタスクをノードに配置する構造上、タスクのスケジューリングなどを行うオーバヘッドによるものであると思われる。

一方 SSS は前述の通り、Owner Compute Rule で計算を行うため、1 ジョブに対して、各ノード上で 1 つずつのサブジョブが立ち上がり map や reduce を行う。従って、スケジューリングは非常に用意で、高速である。

この相違は、計算や IO にかかる時間が少ない場合にとくに顕著になる。例えば、ソースデータ量 1M の場合 (図 7) には、Hadoop と SSS の差が非常に大きくなっているがこれは主に繰り返し時の MapReduce ジョブ起動のオーバヘッドによるものであると思われる。この場合繰り返し回数は 22 回であるため、400 秒あまりが、MapReduce ジョブの起動で消費される計算になる。全実行時間が 550 秒程度なので、7 割以上の時間が MapReduce の繰り返しのために空費されたことになる。

一方ソースデータ量 3M の場合 (図 9) には、計算時間、I/O 時間がドミナントになるため、繰り返しオーバヘッドの寄与は

小さくなり、Hadoop と SSS の差が小さくなる。

5. おわりに

PrefixSpan 法による系列パターン検出をアプリケーションとして、開発中の MapReduce 処理系 SSS の性能を評価した。この結果、SSS は全般に Hadoop と比較して高速であり、さらにノード数増大による速度向上の余地があることが確認できた。

今後の課題は以下の通りである

- 本稿で示した Hadoop の実行結果は必ずしも Hadoop を十分にチューニングした結果ではない。レプリカ数やワーカ数などを調整し、アプリケーションに対して最適な設定を確立することが、SSS と比較する上で不可欠である。
- 3. 節で述べたとおり、本稿では文献 [10] で提案されている 3 つの MapReduce 化手法のうち *s-EB* を採用した。これは文献 [10] によれば、3 つの手法中もっとも高速であるためである。

しかしデータ量が大きい領域では *s-BE* のほうが高速であるとの報告もある(注2)。*s-BE* は限定と射影を異なる MapReduce で行うもので MapReduce の回数は 2 倍になる一方、中間データ量は削減される。*s-BE* を用いた実装を行い、比較することが今後の課題である。

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] 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] 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 '11 (to appear) (2011).

[5] 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).

[6] FAL Labs: Tokyo Cabinet: a modern implementation of DBM, `http://fallabs.com/tokyocabinet/index.html`.

[7] FAL Labs: Tokyo Tyrant: network interface of Tokyo Cabinet, `http://fallabs.com/tokyotyrant/`.

[8] : Java Binding of Tokyo Tyrant, `http://code.google.com/p/jtokyotyrant`.

[9] 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).

[10] 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).

[11] 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).

(注2)：HPC 研究会での当該論文の講演時の報告。