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に向けたKVSの改良

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　広く用いられている Hadoop と比較して、より広い範囲に適用可能な MapReduce 処理系 SSS を開発している。SSS はオープンソースのキーバリューストア（KVS）である Tokyo Cabinet とそのネットワークインターフェイスである Tokyo Tyrant をストレージとして用いる。しかし、SSS のアクセスパターンはバルクでの読み出し、書き込み、削除が主で、Tokyo Tyrant が本来想定している細粒度のアクセスとはかけ離れており、Tokyo Tyrant が用意する API を利用するだけでは、Tokyo Cabinet/Tokyo Tyrant 本来の性能を引き出すことが出来なかった。本稿ではわれわれが行った、Tokyo Cabinet に対する改変について報告する。SSS が行う範囲指定処理を Tokyo Cabinet および Tokyo Tyrant に追加し、さらに内部でのロックを最適化した。その結果、Tokyo Cabinet を直接用いるマイクロベンチマークで、読み込み時 5 倍書き込み時 2 倍の速度向上を確認した。また、SSS 全体としてのマクロベンチマークでも読み込み時 3 倍、書き込み時で 1.3 倍の速度向上を確認した。
キーワード　MapReduce, 分散並列計算, KVS

KVS improvement for MapReduce System SSS

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 developing a MapReduce system called SSS, which could be utilized for broader applications than Hadoop. We employed Tokyo Cabinet, an open sourced key value database, as storage layer of SSS, and found that Tokyo Cabinetwas not suited for the bulk read/write/delete accesses, which is common in SSS usage. This paper describes modification for Tokyo Cabinetand Tokyo Tyrantwe made. We have added range operations to Tokyo Cabinetand optimized internal data lock operations for bulk accesses. We confirmed that this modification caused substantial speedup; Tokyo Cabinetlevel evaluation showed x5 speed up for read, x2 speedup for write, and SSS level evaluation showed x3 speedup for read, x1.3 speedup for write.
Key words　MapReduce, Distributed Parallel Execution, KVS

1. はじめに

広く用いられている Hadoop [1] と比較して、より広い範囲に適用可能な MapReduce 処理系 SSS [2] [3] を開発している。SSS はオープンソースのキーバリューストア（KVS）である Tokyo Cabinet [4] をネットワークインターフェイス Tokyo Tyrant [5] を介して呼び出し、ストレージとして用いる。しかし、Tokyo Cabinet/Tokyo Tyrant がもともと 1 つのキーバリューペア単位での読み出し書き出しを想定して設計されているのに対して、SSS のアクセスパターンではレンジを指定しての一括読み出し、削除、およびソートされたデータの一括書き込みを行うのみで、API の設計に大きな乖離があった。このため、Tokyo Cabinet/Tokyo Tyrant が用意する API を利用すると Tokyo Cabinet 本来の性能を引き出すことが出来なかった。

本稿ではわれわれが行った、Tokyo Cabinet および Tokyo Tyrant に対する改変について報告する。われわれは、SSS が行うバルクでのデータ読み出し、書き込み機能を Tokyo Cabinet に追加し、Tokyo Cabinet 内部でのデータ構造のロックを最小化した。また Tokyo Tyrant に API を追加し、バルクコマンドをネットワーク越しに利用できるよう改変をおこなった。

この改変の影響を Tokyo Cabinet レベル、Tokyo Tyrant レベル、SSS レベルでそれぞれ評価した結果、いずれの場合にも

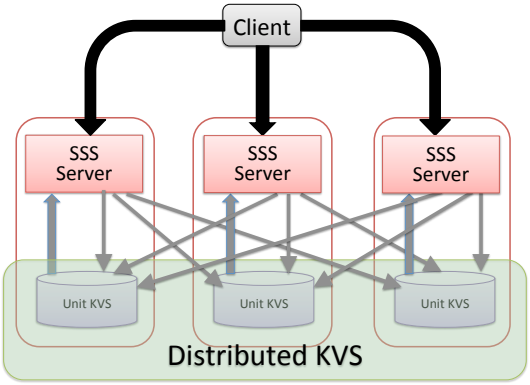

図 1　SSS の構成

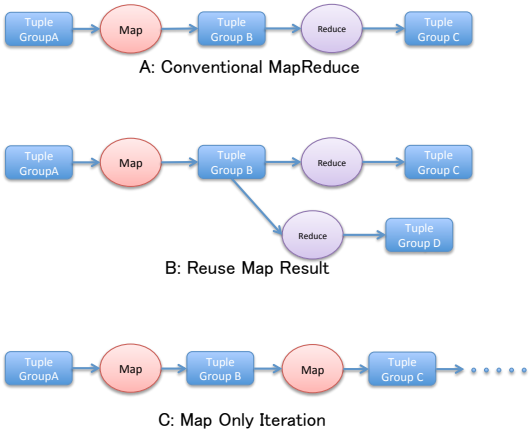

図 2　タプルグループと Map/Reduce

大きな速度向上を確認することができた。

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その実装である SSS について概説する。3. 節で SSS で利用した KVS である Tokyo Cabinet およびそのネットワークインターフェイスである Tokyo Tyrant を概説する。4. 節で SSS のアクセスパターンを踏まえた Tokyo Tyrant の改良について述べる。5. 節で提案の変更による性能向上の評価を行う。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。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　SSS

SSS [2] 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 (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。

2.2.1　タプルグループ

SSS はデータ空間を**タプルグループ**とよぶサブ空間に分割する。個々の Mapper/Reducer は、特定のタプルグループからタプル（キーバリューペア）を読み込み、特定の 0 個以上のタプルグループに対して出力を行う。複数の Mapper/Reducer がタプルグループを共有することも可能である。

図 2:A に、一般的なの MapReduce の様子を示す。Mapper/Reducer はそれぞれ 1 つのタプルグループから読み込み、1 つのタプルグループに対して書き出している。図 2:B では、Mapper の出力を再利用している。図 2:C は、Mapper のみによる繰り返し演算を示している。B,C のようなワークフローは Hadoop では実現できない。

2.2.2　分散 KVS の実装

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。

要素 KVS として Tokyo Cabinet を用い、Tokyo Cabinet へのリモートインターフェイスとして Tokyo Tyrant を用いていた。また、SSS からの Tokyo Tyrant へのブリッジとしては Tokyo Tyrant の Java Binding [7] を用いた。これは、Tokyo

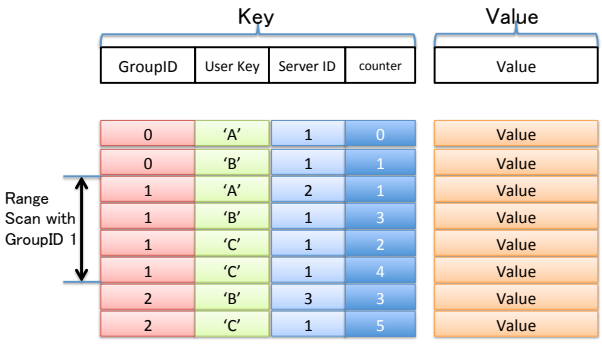

図 3　タプルグループの実装

Tyrant の C クライアントを Java でラップしたものである。

2.2.3　タプルグループの実装

Tokyo Cabinet はタプルグループのような名前空間をサポートしていないので、われわれはこの機能をキーのエンコーディングで実現した (図 3)。Tokyo Cabinet のキーに、ユーザの指定したキー（ユーザ・キー）だけでなく、タプルグループの ID や SSS サーバ番号、カウンタをエンコードする。

このようにエンコードすることで、特定のタプルグループに属するすべてのタプルを取得するには、プレフィックスを指定したスキャンが出来れば良いことになる。これは、Tokyo Cabinet で容易に実現できる。

3.　Tokyo Cabinet と Tokyo Tyrant

Tokyo Cabinet [4] は、Fal Labs の平林幹雄氏が開発したキーとバリューからなるペアを格納するデータベース、いわゆるキーバリューストア (KVS) である。Tokyo Tyrant [5] は、やはり平林氏が開発した Tokyo Cabinet に対するネットワークインターフェイス層で、リモートからの Tokyo Cabinet へのアクセス機能を提供する。Tokyo Cabinet、Tokyo Tyrant は Tokyo シリーズと呼ばれる一連の DB 関連製品群の一部である(注1)。

3.1　Tokyo Cabinet の構造

Tokyo Cabinet は、ハッシュ、B+ tree、固定長データベース、テーブル型データベースなどさまざまな内部データ構造を取ることができるが、われわれの用途では、キー順に整列した構造であることが必要なため、B+ tree を用いた。

図 4 に、B+ tree を用いた場合のデータ構造を示す。B+ tree を用いた場合、Tokyo Cabinet はデータをリーフと呼ぶ単位で管理する。B+ tree には個々のキーバリューペアではなく、リーフが格納され、リーフには連続したキーを持つキーバリューペアが格納される。リーフはメモリ上に LRU でキャッシュされる。

3.2　Tokyo Tyrant の API

Tokyo Tyrant は Tokyo Cabinet の多彩なコマンドをサポートするが、キーバリューペアの連続アクセスに関連する API のみを抜き出したものを表 2 に示す。

(注1)：現在 Tokyo シリーズの開発は休止しており、作者である平林幹雄氏は後継システムである Kyoto シリーズへの移行を勧めているが、KyotoCabinet には SSS の実装に不可欠な範囲取得に利用できる機能が実装されていないため、SSS では TokyoTyrant を用いた。

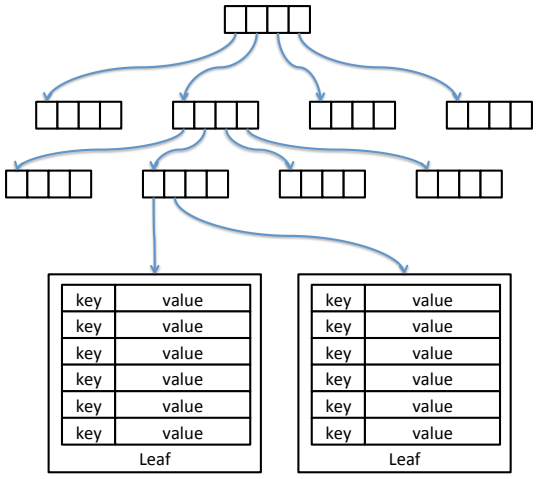

図 4　B+ tree を用いた場合の Tokyo Cabinet のデータ構造

4.　Tokyo Cabinet、Tokyo Tyrant の改変

4.1　SSS のアクセスパターン

SSS における KVS へのアクセスは以下の 3 つのパターンに限定される。

4.1.1　範囲指定のバルク読み出し

SSS の Map/Reduce は、特定のタプルグループに対して一括での処理を行う。タプルグループは前述のように、キーに対するプレフィックスで実装されているため、タプルグループからの一括読み出しは、特定のプレフィックスを持つキーに対するレンジ読み出しに帰着される。

4.1.2　同一プレフィックスデータのバルク書き込み

Map/Reduce は特定のタプルグループに対する書き込みを行う。この際、書きこみ対象となるユニット KVS はパーティション関数によって分散される。また、Tokyo Cabinet レベルでの書き込みを効率化する目的で、SSS 内部でバッファリングし、ソートしてから書き込みを行う。書き込まれるデータは、同一のプレフィックスを持ち、さらにソートされてはいるが、他のノードからの書き込みと同じ領域に対して書き込まれるため、必ずしも完全に連続した書き込みとはならないことに注意が必要である。

4.1.3　範囲指定のバルク消去

SSS ではタプルグループ単位での消去を行う。これは読み出しの際と同様に、特定のプレフィックスを持つ範囲に対する一括した消去となる。

4.2　オリジナルの Tokyo Cabinet/ Tokyo Tyrant での実装

表 2 に示したとおり、オリジナルの Tokyo Tyrant には直接あるキーの範囲のデータを取得、消去するコマンドは用意されていない。たとえばある範囲のデータを読み出すには fwmkeys でキーの配列を取り出し、そのキー配列に対して、getlist コマンドによって読み込みを行う事になる。削除の時にも同様に fwmkeys で取得したキー配列に対して削除を行う必要がある。この場合、不要なキー配列の転送が発生するため、効率がよく

表 1　Tokyo Tyrant の主要データ入出力コマンド

コマンド名	意味	引数	出力
put	キーバリューペアの書き込み	キーバリュー	なし
get	キーバリューペアの読み出し	キー	バリュー
out	キーバリューペアの読み出し	キー	なし
fwmkeys	キーの順方向読み出し	開始点、数	キーのリスト
putlist	一括書き込み	キーバリューペアのリスト	なし
getlist	一括読み出し	キーのリスト	キーバリューペアのリスト
outlist	一括削除	キーのリスト	なし

ない。

また、putlist, getlist、outlist の実装には性能上の問題もあった。これは、これらのコマンドが 1 キーバリューごとに対象リーフに対するロックを取得・解放するためである。このような構造になっているのは、Tokyo Cabinet 上で複数のスレッドが同時に動作する際に、特定のスレッドが長時間ロックすることによって、他のスレッドの動作に影響が出ることを避けるためだと思われる。

4.3　Tokyo Cabinet/ Tokyo Tyrant への変更

fwmkeys と getlist/outlist の組み合わせによるアクセスでは、キー配列の転送が余分に行われ、さらにクライアント側にキー配列を一時的に保持する空間を保持する必要があり、メモリ管理のコストが生じる。このため、あらたに、range/rangeout コマンドを実装した。これらのコマンドでは、始点キーと終点キーを指定し、その間の範囲のキーをもつキーバリューペアに対して一括して操作を行う。

また、SSS での利用においては、バルク操作時のスループットの向上が他のスレッドへの影響よりも重要であるため、対象リーフに対するロックを解除せずに連続してアクセスするよう改変した。

5.　評　　価

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。今回の実験では、すべて ioDrive を用いている。

5.2　ioDrive の性能

まず基礎データとして ioDrive 単体の性能を測定した。計測には [8] を使用した。ジョブ数は、SSS で利用する場合のスレッド数とあわせて 16 とした。計測は、READ, RANDOM READ, WRITE, RANDOM WRITE で行った。

図 5 に計測結果を示す。X 軸はブロックサイズ、縦軸はスループットである。READ と RANDOM READ, WRITE と RANDOM WRITE はほぼ同じ傾向を示している。これは ioDrive ではヘッドのシークが生じないためである。

5.3　Tokyo Cabinet、Tokyo Tyrant レベルでの性能

Tokyo Cabinet/Tokyo Tyrant レベルでの性能を評価した。

表 3　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

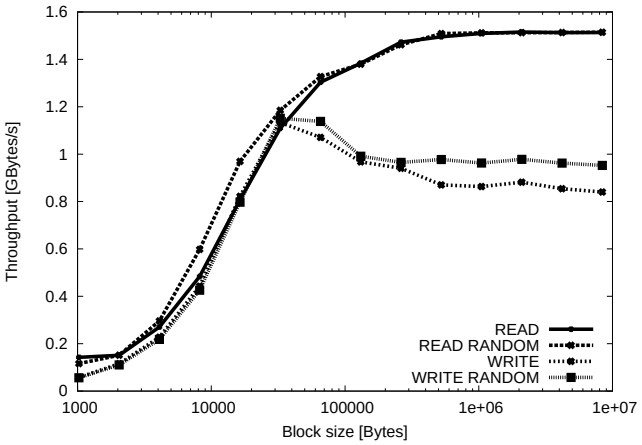

図 5　ioDrive 単体でのベンチマーク結果

評価においては、書き込み（PUT）、読み出し（GET）、消去（OUT）をそれぞれオリジナルのバージョンおよび変更したバージョンで実行した。また実験は、Tokyo Cabinet、Tokyo Tyrant ローカル、Tokyo Tyrant リモートの 3 つの環境で行った (図 6)。スレッド数は 16 とした。バルク操作するキーバリューペアの数であるバッチサイズを 128, 1Ki, 8Ki と変化させた。

Tokyo Cabinet 内のキャッシュ、およびディスクキャッシュの影響を排除するため、書き込み後に一度 Tokyo Cabinet プロセスを停止し、さらにディスクキャッシュをフラッシュしている。

図 7 に Tokyo Cabinet レベルでのベンチマーク結果を示す。バッチサイズ 1Ki の場合で比較すると、読み込み時で 0.19GB/s から 0.99GB/s へと、5 倍以上の性能向上、書き込み時には 0.17GB/s から 0.37GB/s へと 2 倍以上の性能向上が見られる。

図 8、図 9 に Tokyo Tyrant レベルでのベンチマーク結果

表 2　Tokyo Tyrant の主要データ入出力コマンド

コマンド名	意味	引数	出力
putlist_atomic	一括アトミック書き込み	キーバリューペアのリスト	なし
range_atomic	範囲指定読み出し	始点キー、終点キー	キーバリューペアのリスト
rangeout_atomic	範囲指定削除	始点キー、終点キー	なし

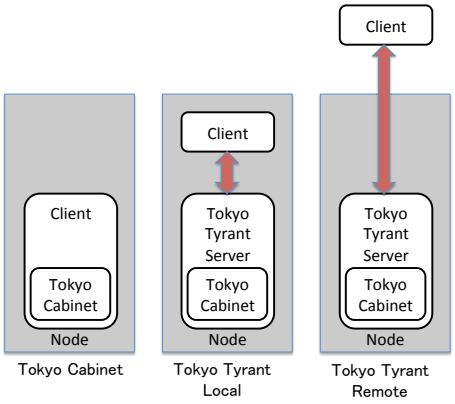

図 6　Tokyo Cabinet,Tokyo Tyrant 評価環境

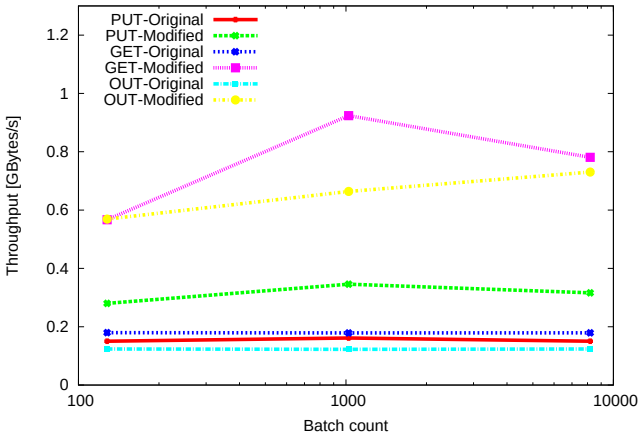

図 7　Tokyo Cabinet レベルでのベンチマーク結果

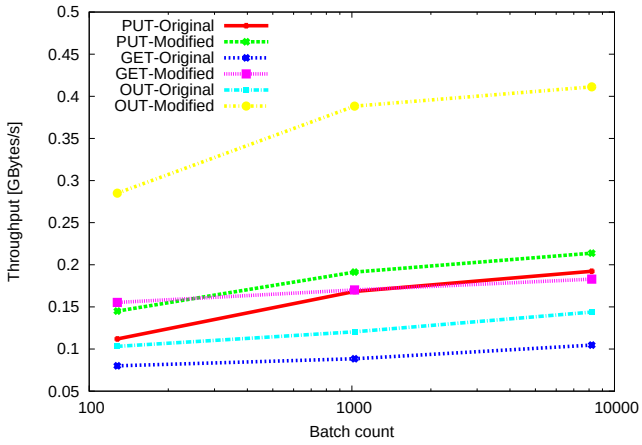

図 8　Tokyo Tyrant ローカルベンチマーク結果

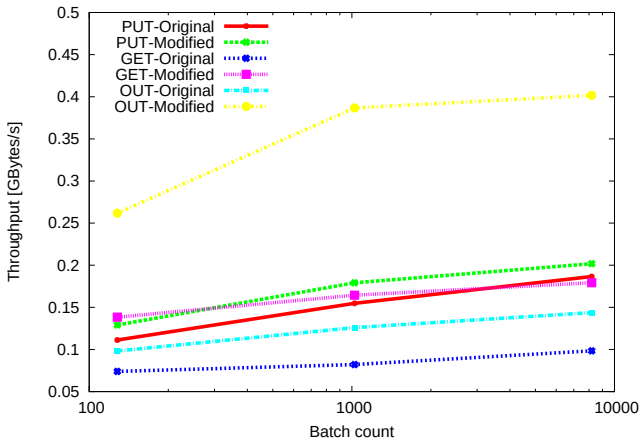

図 9　Tokyo Tyrant リモートベンチマーク結果

を示す。削除の性能は Tokyo Cabinet とそれほど変わらないが、書き込み、読み出しとも大きく性能が低下している事がわかる。更新後で、書き込みスループットが 0.37GB/s から 0.21GB/S へと 0.55 倍、読み込みスループットが 0.99GB/s から 0.18GB/s へと 0.18 倍となっている。

これはプロセス間データ転送を行うため、データコピーが多発するためであると思われる。削除の場合の性能が Tokyo Cabinet と Tokyo Tyrant とで殆ど変わらないのは、削除の際にはデータの転送がないためである。

Tokyo Tyrant ローカルの場合の変更前と変更後を バッチサイズ 1Ki の場合で比較すると、読み込み時で 0.17GB/s から 0.37GB/s へ 2.1 倍、書き込み時で 0.18GB/s から 0.21GB/s へ 1.14 倍程度の速度向上が見られた。

また、Tokyo Tyrant ローカルと Tokyo Tyrant リモートの場合では、ほとんど性能に差はない。これは、ネットワークが 10GE と十分高速であるため、データコピーのオーバヘッドがドミナントになるためであると思われる。

5.4　SSS レベルでの性能

同様のベンチマークプログラムを SSS のレベルで実行した。SSS の Mapper 関数内で読みだし、書き出しを行い、その時間を計測しスループットを算出している。読み出し・書き込み時のバッチサイズを 128, 1Ki, 8Ki と変化させている。タプルのキー、バリューのサイズはそれぞれ 1KiB、レコード数は 1Mi 個、総データ量は 2G である。この評価では、Tokyo Tyrant サーバの再起動やディスクキャッシュのクリアを行なっていないため、キャッシュの効果を含めての評価となっている。

結果を図 10 に示す。性能はバッチサイズによらずほぼ一定である。バッチサイズ 1Ki の場合で比較すると、書き込み時で約 1.3 倍、読み込み時で約 3 倍のスループット向上が得られている。

6.　おわりに

MapReduce 処理系 SSS のアクセスパターンに対応するため

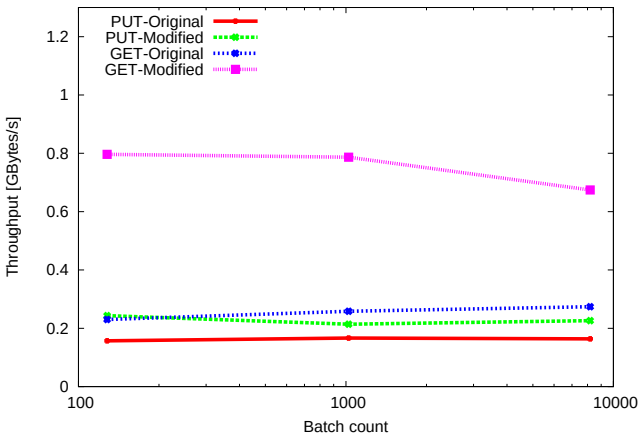

図 10　SSS レイヤでのベンチマーク結果

の、KVS Tokyo Cabinet の改変について述べた。SSS が行うバルク処理を Tokyo Cabinet に追加し、内部でのロックを最適化することで、Tokyo Cabinet を直接用いるマイクロベンチマークで、読み込み時 5 倍書き込み時 2 倍の速度向上を確認した。また、SSS 全体としてのマクロベンチマークでも読み込み時 3 倍、書き込み時で 1.3 倍の速度向上を確認した。

今回の改変によって、SSS のアクセスパターンに関しては大幅な性能向上を確認することができたが、依然として Tokyo Tyrant を介した場合の性能は、Tokyo Cabinet を直接利用する場合と比較して大幅に低い。これは Tokyo Tyrant のネットワーク上のプロトコルがバルク転送に最適化されておらず、オーバヘッドとなっているためであると思われる。このプロトコルをバルク転送に最適化することでさらなる高速化を行うことが今後の課題である。

また、範囲を指定した消去コマンドである rangeout でもかなりの時間がかかることが明らかになった。これは、消去の際にも一旦ディスクからメモリ上に全データを取り込む Tokyo Cabinet の構造上の制約からくるものである。SSS でワークフローを処理する場合、大規模な範囲指定消去が多発するため、今後何らかの方法で最適化していく必要がある。

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．

文　　献

[1] : Hadoop, `http://hadoop.apache.org/`.
[2] 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).
[3] 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).
[4] FAL Labs: Tokyo Cabinet: a modern implementation of DBM, `http://fallabs.com/tokyocabinet/index.html`.
[5] FAL Labs: Tokyo Tyrant: network interface of Tokyo Cabinet, `http://fallabs.com/tokyotyrant/`.
[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] : Java Binding of Tokyo Tyrant, `http://code.google.com/p/jtokyotyrant`.
[8] : fio, `http://freecode.com/projects/fio`.