情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-HPC-124 No.14
2010/2/23

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

小川 宏高[†1] 中田 秀基[†1] 美田 晃伸[†1,†2]
広渕 崇宏[†1] 高野 了成[†1] 工藤 知宏[†1]

大規模なデータインテンシブアプリケーションの高性能実行の必要性から，大規模データ処理に特化された分散処理を実現する，Data-Intensive Scalable Computing (DISC) が注目されている．MapReduce は，そうした DISC を実現するシステムのひとつであるが，近年利用可能になってきた，10Gbit/sec クラスの読み書き性能を持つ高速な SSD (Solid State Drive) との組み合わせでは，ソフトウェアオーバヘッドが課題となり，十分な性能が得られない危惧がある．そこで我々は，10Gbit/sec クラスの読み書き性能を持つフラッシュメモリストレージに適した，MapReduce システムの実現を目指し，分散キーバリューストアを基盤とする MapReduce システムのプロトタイプ実装を行った．本稿では，我々のプロトタイプ実装の概要を示すとともに，その基盤となるキーバリューストアの既存実装の性能評価を行う．

Preliminary Evaluation of Fast Flash Memory Oriented Key Value Stores

Hirotaka Ogawa,[†1] Hidemoto Nakada,[†1]
Akinobu Mita,[†1,†2] Takahiro Hirofuchi,[†1]
Ryousei Takano[†1] and Tomohiro Kudoh [†1]

The practical needs of efficient execution of large-scale data-intensive applications propel the research and development of Data-Intensive Scalable Computing (DISC) systems, which manage, process, and store massive data-sets in a distributed manner. MapReduce is a representative of such DISC systems. On the other hand, today, HPC community is going to be able to utilize very fast SSDs (Solid State Drives) with 10 Gbit/sec-class read/write performance. However, coupling such very fast storage devices with MapReduce systems, much of the benefits of devices can easily be lost because of software overhead incurred by MapReduce systems themselves. To resolve these problems, we are aiming to design and implement a novel DISC system called "SSS", which can effectively exploit the I/O performance of clusters with 10 Gbit/sec-class flash memories. In this paper, we first outline our prototype MapReduce system which utilizes distributed key-value store. And we perform an extensive benchmark for evaluating existing open-source implementations of key-value stores.

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高性能実行の必要性から，大規模データ処理に特化された分散処理を実現する，Data-Intensive Scalable Computing (DISC) が注目されている．MapReduce[1)] は，そうした DISC を実現するシステムのひとつである．

MapReduce は，本質的には分散したキーバリューペアの集合への大域的かつ統一的な操作を可能にする．しかしながら，キーバリューペアの入出力に Google File System (GFS)[2)] 上に格納されたテキストファイルやシリアライズされた構造データを用いることを前提にしており，実行時の読み書き性能を重視した設計を採っていない．これは，MapReduce の「典型的な」ワークロードが大容量データのランダムアクセスを必要とせず，かつ多数のストレージノードの I/O 性能の総計だけに依存するものに限定されているためである．

一方で今日，高速かつ低消費電力な外部記憶媒体として NAND 型フラッシュメモリを使用する SSD (Solid State Drive) が HPC システムの重要な構成要素として認識されてきている．特に Fusion-io 社の ioDrive[TM3)] duo に代表されるように，PCI-Express をインタフェースとして利用するハイエンドの SSD は，約 10Gbit/sec の読み書き性能を達成している．これは主記憶のアクセス速度の約 1/10 倍，ハードディスクの約 10 倍にも相当する．

我々は，こうした 10Gbit/sec クラスの読み書き性能を持つフラッシュメモリストレージに適した MapReduce システムの実現を目指し，分散キーバリューストアを基盤とする MapReduce システムのプロトタイプ実装を行った．

本稿では，プロトタイプ実装の概要を示すとともに，その基盤となるキーバリューストアの既存実装の性能評価を行う．

†1 産業技術総合研究所 / National Institute of Advanced Industrial Science and Technology (AIST)
†2 （株）フィックスターズ / Fixstars Corporation

© 2010 Information Processing Society of Japan

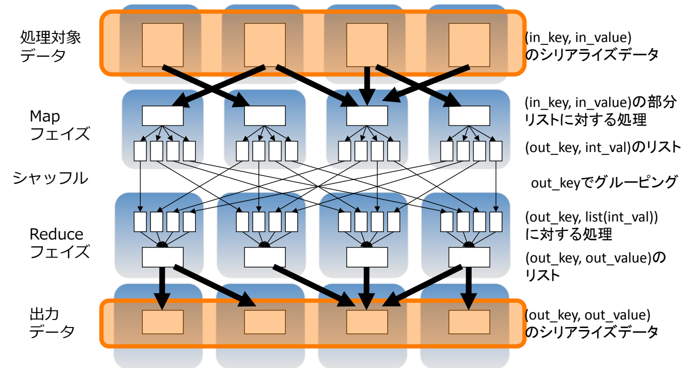

図 1 MapReduce

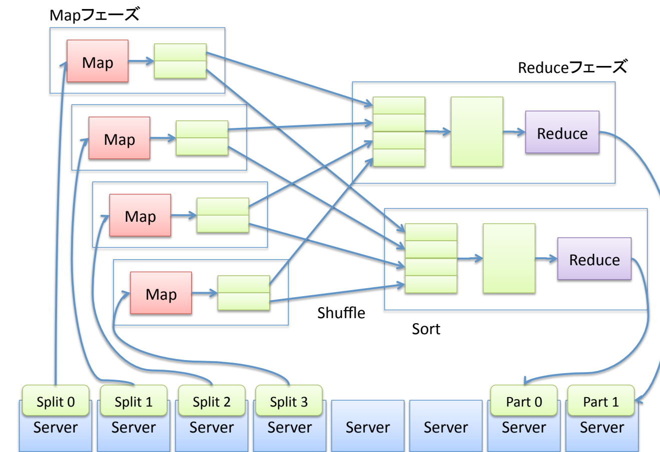

図 2 MapReduce Execution

2. キーバリューストアを基盤とした MapReduce 処理系の実現

2.1 MapReduce の実行モデル

MapReduce は一般に，キーバリューペアのリストデータを処理するためのプログラミングモデルとその分散実装を指す．

MapReduce のプログラミングモデルでは，データ処理のプロセスを Map，Shuffle & Sort，Reduce の 3 フェーズに分解して実行する．まず，Map フェーズでは各キーバリューペアから中間データを生成する．中間データはキーバリューペアのリストとする．Shuffle & Sort フェーズではキーが同じ中間データをまとめて，キーと値のリストのペアのリストを生成する．Reduce フェーズでは，各キーと値のリストのペアから出力キーバリューペアを生成する（図 1）．

このプログラミングモデルは，本質的には分散した key-value ペアの集合への大域的かつ統一的な操作を可能にする一方，分散実装においては，Google File System (GFS)[2] や HDFS (Hadoop Distributed File System)[4] に代表される分散ファイルシステム上に格納されたテキストファイルやシリアライズされた構造データを取り扱う．

MapReduce の分散実装では，まず入力データを M 個に分割し，分割されたデータの各レコードに対して Map 処理を行う．この M 個の処理は Map タスクと呼ばれ，複数のノード上で並列に実行される．次に Map タスクが出力した中間データをキーごとにソート，マージした上で R 個に分割し，分割された中間データの各キーに対して Reduce 処理を行う．この R 個の処理は Reduce タスクと呼ばれ，やはり複数のノード上で並列に実行される（図 2）．

Google の実装[1] の構成要素は，単一のマスターと多数のワーカーである．マスターはクライアントからの MapReduce ジョブの受付や Map/Reduce タスクのワーカーへのスケジューリングなどを受け持ち，ワーカーは各 Map/Reduce タスクを実行する．

MapReduce のオープンソース実装である Hadoop MapReduce[5] はほぼ Google の実装に準じた機能を提供する．Hadoop では，マスター，ワーカーをそれぞれ JobTracker，TaskTracker と呼んでいる．

どちらの実装も入出力データの格納場所として分散ファイルシステムを利用することを

© 2010 Information Processing Society of Japan

Vol.2010-HPC-124 No.14
2010/2/23

前提としており，チャンク（ブロック）のローカリティを考慮したタスクの割り当てを行う．Hadoop の実装では，入力となるチャンク（ブロック）を保持しているチャンクサーバ（DataNode）から物理的に近いワーカー（TaskTracker）に Map タスクを割り当てることで，通信コストを抑制するなどの最適化が施されている．

2.2 キーバリューストアを基盤とした MapReduce 実装

MapReduce は本質的にキーバリュー集合の操作を行うことを目的とした処理系であるにも関わらず，GFS/HDFS はキーバリューデータの入出力にテキストファイルやシリアライズされた構造データを用いることを強制する．つまり，プログラミングモデルと分散実装との間にセマンティックギャップがある．

この問題を解決するために，我々は共有ストレージシステムとして分散キーバリューストアを利用した実装を実現する*1.

2.2.1 分散キーバリューストア

キーバリューストアとはキーと値の組を保持することができるデータストアであり，memcached の実装が広く用いられている．分散キーバリューストアは，こうしたキーバリューストアを多数並列に並べることで，アクセス性能，容量，耐故障性などデータストアに求められる要求に応えるものであり，実際の大規模な Web アプリケーションにおいても利用されるようになってきている．

分散キーバリューストアに多様な実装があるが，我々は最終的にはデータインテンシブ計算への特化やフラッシュメモリストレージの読み書き性能の活用を目的としているため，比較的単純な実装を前提とする．すなわち，(1) データをオンメモリではなく二次記憶装置に記録し永続化すること，(2) 分散キーのハッシュ値を用いたコンシステントハッシングを行い，クライアントもしくは Map, Reduce タスクからゼロホップで担当キーバリューストアサーバにアクセスできること，(3) レプリケーションは行わないこと，である．

図 3 に示すように，フラッシュメモリストレージを備えたキーバリューストアサーバが複数用意してあるものとする．取り扱うキーバリューデータは，(*distribution-key*, *local-key*, *value*) のタプルからなるものとする．*distribution-key*, *local-key* はそれぞれ，ハッシュ値から担当サーバを決定するためのキー，単一キーバリューストア内のキーバリューを区別するためのキーとして用いられる．

*1 本稿執筆の時点では，Python によるプロトタイプ実装が完成している．が，実験環境が整わないためベンチマークは割愛する．

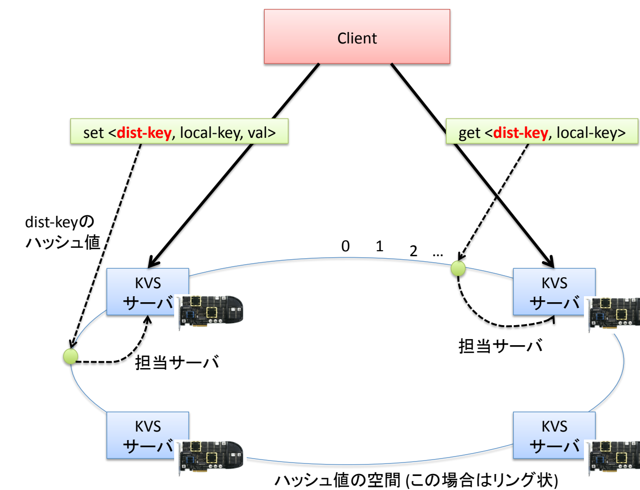

図 3 分散キーバリューストア

クライアントは *distribution-key* のハッシュ値から担当キーバリューストアサーバ (以降，サーバ) を決定し，担当サーバに対してリクエストを行う一方，各サーバはリクエストに応じて (*distribution-key*, *local-key*, *value*) のタプルをロード・ストアする機能を提供する．また，*distribution-key* から対応する *local-key* のリストを得る機能も提供する．

2.2.2 MapReduce の実現

我々の MapReduce 実装は上記で述べた分散キーバリューストアを基盤とする．具体的には，クライアントは入力として *distribution-key* のリストを与え，出力として *distribution-key* のリストを受け取る．入出力データの本体は，分散キーバリューストア内に格納される．

MapReduce 処理系の構成は次のような単純なものにできる (図 4).

- Map フェーズ
  Map タスクは，クライアントが与えた *distribution-key* ごとに生成される．*distribution-key* に関連付けられたすべてのデータを分散キーバリューストアから取得し，取得データを用いて計算を行う．計算結果の中間データを分散キーバリューストアに格納し，その *distribution-key* のリストのみを返す．

© 2010 Information Processing Society of Japan

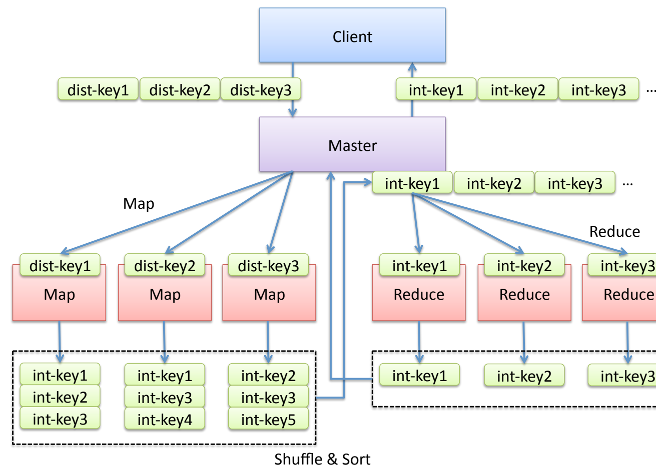

図 4 KVS をバックエンドとして用いる MapReduce

- Shuffle & Sort フェーズ
  すべての Map タスクの生成した *distribution-key* リストをマスターに回収し，重複を除いた *distribution-key* リストを生成する．
- Reduce フェーズ
  Reduce タスクは，Shuffle フェーズで生成された *distribution-key* ごとに生成される．*distribution-key* に関連付けられたすべてのデータを分散キーバリューストアから取得し，取得データを用いて計算を行う．計算結果を分散キーバリューストアに格納し，その *distribution-key* のリストのみを返す．ただし，一般的に Reduce タスクは計算結果のみを生成するため，返値は Reduce タスクの入力 *distribution-key* のみを要素として含むリストとなる．

2.3 例: ワードカウント

以下では，ワードカウントを行うアプリケーションでの実行のステップを説明する．

まず，分散キーバリューストアに複数のドキュメントが行単位で格納されているものとする．

```
<’file01’, ’1’, ’Hello World Bye World’>
<’file02’, ’1’, ’Hello SSS Goodbye SSS’>
```

クライアントは，<’file01’, ’file02’>というキーを指定してマスターに処理を要求する．マスターはキーごとに Map タスクを生成する．

distribution-key ’file01’ を担当する Map タスクは，

```
<’file01’, ’1’, ’Hello World Bye World’>
```

というただ 1 つのタプル[*1]を分散キーバリューストアから読み出し，ワードごとに分割して下のようなタプルを生成する．*local-key* には Map タスクの ID と生成タプルのシリアル番号からなるユニークな文字列とする．

```
<’Hello’, ’map0-word0’, 1>
<’World’, ’map0-word1’, 1>
<’Bye’,   ’map0-word2’, 1>
<’World’, ’map0-word3’, 1>
```

その上で生成された中間データの *distribution-key* をマスターに返値する．

```
<’Hello’, ’World’, ’Bye’, ’World’>
```

同様に，*distribution-key* ’file02’ を担当する Map タスクは以下のタプルを生成し，

```
<’Hello’,   ’map1-word0’, 1>
<’SSS’,     ’map1-word1’, 1>
<’Goodbye’, ’map1-word2’, 1>
<’SSS’,     ’map1-word3’, 1>
```

中間データの *distribution-key* をマスターに返値する．

```
<’Hello’, ’SSS’, ’Goodbye’, ’SSS’>
```

マスターは，Map タスクの返値をマージして *distribution-key* のリストを得る．

```
<’Hello’, ’World’, ’Bye’, ’SSS’, ’Goodbye’>
```

この各キーに対して Reduce タスクを生成する．

distribution-key ’Hello’ を担当する Reduce タスクは，

```
<’Hello’, ’map0-word0’, 1>
<’Hello’, ’map1-word0’, 1>
```

⋆1 この場合は，たまたま file01, file02 とも一行のみからなるファイルであったとする．複数行のファイルの場合は複数の該当タプルを分散キーバリューストアから読み出し，処理を行う．

© 2010 Information Processing Society of Japan

Vol.2010-HPC-124 No.14
2010/2/23

という 2 つのタプルを分散キーバリューストアから読み出し，集計して下のタプルを生成し，

```
<'Hello', 'count', 2>
```

distribution-key をマスターに返値する．

```
<'Hello'>
```

他の Reduce タスクも同様に処理を行うことで，下記のタプルが生成され，

```
<'Hello',   'count', 2>
<'World',   'count', 2>
<'Bye',     'count', 1>
<'SSS',     'count', 2>
<'Goodbye', 'count', 1>
```

マスターは Reduce タスクの返値をマージして *distribution-key* のリストを得る．

```
<'Hello', 'World', 'Bye', 'SSS', 'Goodbye'>
```

このリストをクライアントに返値する．

2.4 議　　論

分散キーバリューストアを基盤とした MapReduce 実装を上記のような構成で実現することには，メリットとデメリットがある．

メリットとしては以下のようなものがある．

まず，Map タスクならびに Reduce タスクをデータを保持しているサーバに割り当てることが容易にできることである．ある *distribution-key* が与えられたとき，そのキーを持つキーバリュー集合がどのサーバに格納されているかは，コンシステントハッシングによって一意に決定できるためである．

もうひとつは，Shuffle & Sort フェーズが大幅に簡略化できることである．我々の実装では，Map タスクが中間データを分散キーバリューストアに格納した時点でグルーピングが完了しているためである．これに対して，Google や Hadoop の実装では，Map タスクを実行した全ワーカー (TaskTracker) ノードで中間データをキーごとにソートしておき，それをキーごとに Reduce タスクを実行するワーカー (TaskTracker) ノードに渡し，そのデータをキーごとにグルーピングする必要がある．

もうひとつは，Map タスクと Reduce タスクの処理は本質的に同一の操作になることである．このことはただシステムの実装を容易にするだけでなく，Map と Reduce という 2 フェーズからなる大域的なデータ処理という計算モデルに制約されないデータインテンシブ計算が可能になることを示唆している．

逆に，想定されるデメリットもいくつかある．

まず，キーバリューストアへのアクセスが二次記憶装置へのランダムアクセスを必要とするため，GFS/HDFS が仮定するファイルチャンクへのシーケンシャルなアクセスに比べて性能が劣る可能性がある．しかし，一般的なフラッシュメモリストレージではランダム読み込み性能は問題とならず，また，ioDrive™ などのハイエンドの製品では (制限付きながら) ランダム書き込みもランダム読み込みに匹敵する性能を示すことが知られている．後述の 3 では，ioDrive を用いたキーバリューストア実装の性能評価を行い，性能の問題がないことを実際に確認している．

distribution-key による分散は粒度が小さくて非効率になるのではないかという指摘もあり得る．しかし，それは Google や Hadoop の実装でも同様で，小容量のファイルを多数処理したい場合にはタスク制御のオーバーヘッドが顕在化し得る．

また，既存の MapReduce のオープンソース実装と互換性がないという問題も指摘できる．この点に関しては，我々はより上位のプログラミング言語処理系を提供することで差異を吸収すべきと考えており，そのための取り組みも行っている．

3. キーバリューストアの性能評価

2 で述べてきた分散キーバリューストアを基盤とした MapReduce 処理系が現実的になるためには，バックエンドとして用いるキーバリューストアが十分な性能を達成することを実証しなくてはならない．

既存のキーバリューストアの評価の多くが小容量のデータのリクエスト処理性能に主眼を置いてきたのに対して，我々は大容量のデータの読み書き性能と複数のクライアント (Map/Reduce タスク) からの並列アクセス性能も重視する必要がある．特に，ハイエンドのフラッシュメモリストレージや 10Gb Ethernet の性能を損なわないだけの性能が達成できることが肝要である．

このような観点から我々は，Fusion-io 社 ioDrive と Myrinet 社 Myri-10G を用いたごく小規模な実験環境を構築し，その上で二次記憶装置にキーバリューデータを永続化できるキーバリューストアのオープンソース実装 (MemcacheDB[6]，Tokyo Tyrant[7]，Hail Cloud Computing Project[8] の Chunkd) の性能評価を行った．

3.1 評 価 環 境

表 1 に示すスペックを持つクライアント，サーバを各一台用意した．両者の 10GbE NIC

© 2010 Information Processing Society of Japan

表 1 マシンスペック

CPU	Intel Xeon E5430 @ 2.66GHz x 2
Memory	8GB (DDR2-667)
10GbE NIC	Myricom Myri-10G

表 2 ioDrive 160GB Catalog Spec

NAND Type	Single Level Cell (SLC)
Write Bandwidth	670MB/s (32K packet size)
Read Bandwidth	750MB/s (32K packet size)
IOPS	116,046 (4K read packet size) 93,199 (75/25 r/w mix 4K packet size)
Access Latency	26 μ s Read
Bus Interface	PCI-Express x4

は直結している．サーバには Fusion-io 社の ioDrive 160GB(表 2) が搭載されている．

クライアント，サーバとも CentOS 5.4 がインストールされており，カーネルバージョンはそれぞれ 2.6.32，2.6.30，Myri-10G ドライバは 1.5.1-1.451，1.4.4-1.401，ioDrive のドライバは 1.2.7.2 である．カーネルならびに Myri-10G ドライバのバージョンが両者で異なるのは，ioDrive のドライバが 2.6.30 を要求するためである．

3.2 評価対象のキーバリューストア実装

評価対象のキーバリューストア実装は，MemcacheDB，Tokyo Tyrant，Chunkd の三種である．これらは二次記憶装置にキーバリューデータを永続化するキーバリューストアのオープンソース実装である．それぞれ二次記憶装置への格納形式が異なり，MemcacheDB は BerkeleyDB，Tokyo Tyrant は Tokyo Cabinet と呼ばれる独自のデータベース形式，Chunkd はファイルシステム上のファイル形式 (ファイル名がキー，中身が値) でキーバリューデータを保存する．

それぞれのバージョンは下記の通りである．

- MemcacheDB
  - memcachedb: svn revision 98
  - db-4.8.26
  - libevent-1.4.13-stable
- Tokyo Tyrant
  - tokyotyrant-1.1.39
  - tokyocabinet-1.4.41
- chunkd
  - cld: fcfc100c53c0169378e9fb7d1b2bf2e1133a6431
  - chunkd: 6f868dcf280c5b42874cd2920b332ed9980b1299
  - chunkd は複数スレッドで動作させた場合にメモリ破壊が起きるバグがあるため，非公式なパッチを当ててある．

3 種類ともスレッド数 16 で起動している．

3.3 ベンチマークプログラム

ベンチマークプログラムは，クライアントのスレッド数とキーバリューデータの Value サイズを変えながら一定量のデータを set/get するのに用する時間を測定するものを用意した．

- Key サイズ
  16 バイト固定 (スレッドごとに異なる)
- Value サイズ
  100 | 1,000 | 10,000 | 100,000 | 1,000,000 | 10,000,000 | 100,000,000 | 200,000,000 | 400,000,000 バイト
- スレッド数
  1 | 2 | 4 | 8 | 16
- リクエスト数
  6,400,000,000 / (Value サイズ * スレッド数)

3.4 実 験 結 果

Value サイズごとに，処理の完了に要する時間をプロットしたものを末尾の「付録」にまとめて示した．

Value サイズごとに比較した場合，スレッド数が変わっても全スレッドで発行されるリクエストの総数は同じなので，スレッド数を増やしても実行時間が短くなれば (つまり，グラフが右肩下がりならば)，スレッド数に対してスケールしていると言える．概ねどの Value サイズ，実装でもスケールしていることが分かる．

特に Tokyo Tyrant の get リクエストのスループットをプロットしたものを図 5 に示す．100KB 程度の Value の場合にはほぼネットワークの限界性能までスレッド数に応じてスケールすることが分かる．

実装ごとの比較では，概ねどのケースも Tokyo Tyrant が優れているが，Value サイズが大きくなると Chunkd が速い．Chunkd は Value サイズが小さい領域では他の実装に比べて遅い．これは Chunkd が値をファイルに格納する方式であり，set，get いずれの場合も

© 2010 Information Processing Society of Japan

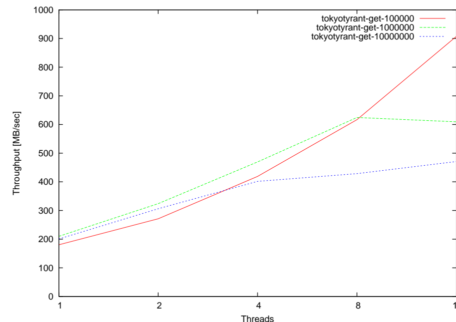

図 5 “Get” Throughput of Tokyo Tyrant (Value 1,000,000-100,000,000)

ファイルを新たに open する必要があるためと推定される．逆に Value サイズが大きい領域ではそうしたシステムコールのオーバーヘッドは無視できるようになるのに加え，値本体を取り出すコストが他実装に比べて小さいためと思われる．

4. 関連研究

油井らによる gridool[9] は，ドキュメンテーションがないため詳細は不明だが，分散ハッシュテーブル上で MapReduce を実行する機能を提供する．gridool が主に解決するのは容量に対するスケーラビリティと可用性・耐故障性である．本研究で示したプロトタイプ実装では耐故障性などは一切考慮していないため，今後の設計・開発において参考にしたいと考えている．

5. まとめ

我々は，10Gbit/sec クラスの読み書き性能を持つフラッシュメモリストレージに見合った，スケーラブルなデータインテンシブアプリケーション実行環境の実現を目指している．

本稿では，その実現に向けて，我々がプロトタイプ実装した分散キーバリューストアを基盤とした MapReduce システムの概要を説明した．また，その基盤となるキーバリューストアの既存実装の性能評価を行った．

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) Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *Communications of the ACM*, Vol.51, No.1, pp.107–113 (2008).
2) Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google file system, *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM, pp.29–43 (2003).
3) Fusion-io: Fusion-io :: Products, `http://www.fusionio.com/Products.aspx`.
4) Borthakur, D.: HDFS Architecture, `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.
5) Apache Hadoop Project: Hadoop, `http://hadoop.apache.org/`.
6) Chu, S.: MemcacheDB, `http://memcachedb.org/`.
7) Hirabayashi, M.: Tokyo Tyrant: network interface of Tokyo Cabinet, `http://1978th.net/tokyotyrant/`.
8) Garzik, J.: Hail Cloud Computing Wiki, `http://hail.wiki.kernel.org/index.php/Main_Page`.
9) Yui, M.: gridool: An Infrastructure of Parallel Job Execution on Grid, `http://code.google.com/p/gridool/`.

付録

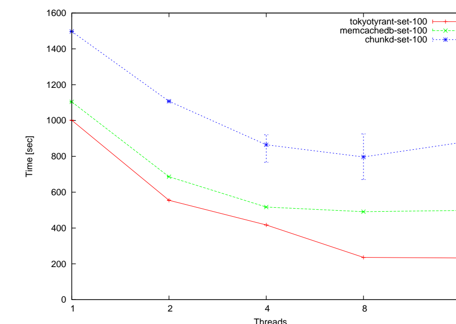

図 6 Benchmark result (Value 100 bytes, 6,400,000 sets)

© 2010 Information Processing Society of Japan

Vol.2010-HPC-124 No.14
2010/2/23

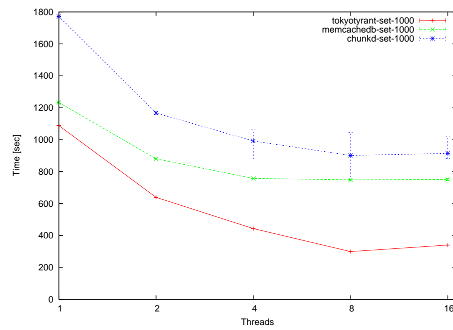

図 7 Benchmark result (Value 1,000 bytes, 6,400,000 sets)

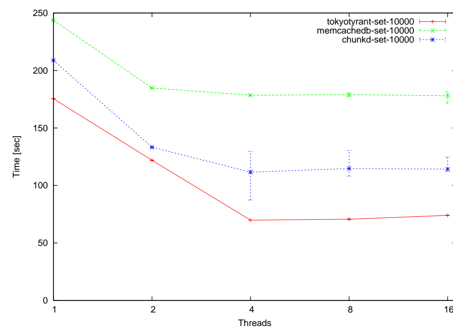

図 8 Benchmark result (Value 10,000 bytes, 640,000 sets)

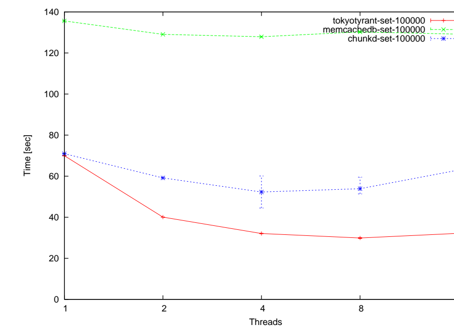

図 9 Benchmark result (Value 100,000 bytes, 64,000 sets)

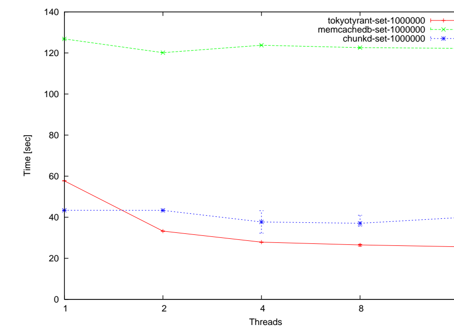

図 10 Benchmark result (Value 1,000,000 bytes, 6,400 sets)

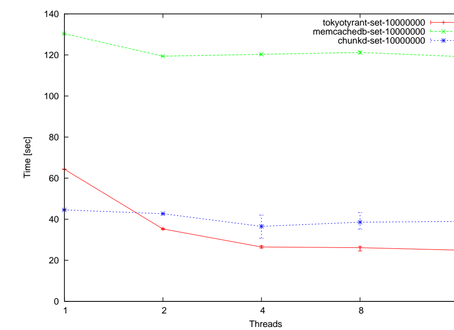

図 11 Benchmark result (Value 10,000,000 bytes, 640 sets)

© 2010 Information Processing Society of Japan

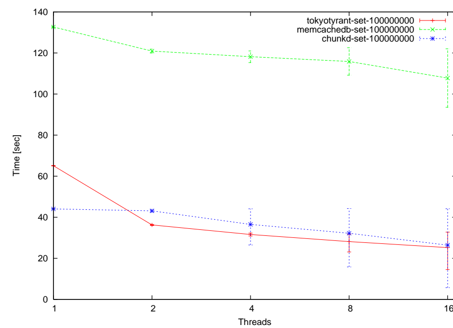

図 12 Benchmark result (Value 100,000,000 bytes, 64 sets)

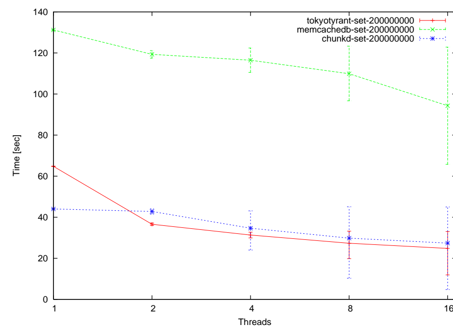

図 13 Benchmark result (Value 200,000,000 bytes, 32 sets)

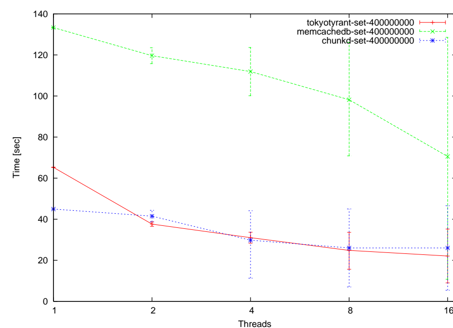

図 14 Benchmark result (Value 400,000,000 bytes, 16 sets)

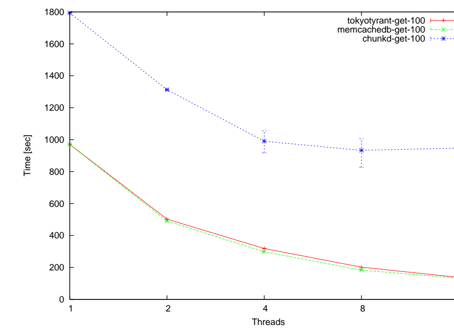

図 15 Benchmark result (Value 100 bytes, 6,400,000 gets)

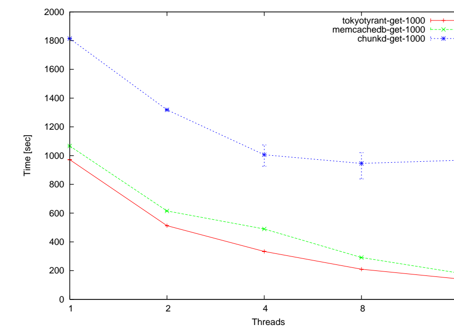

図 16 Benchmark result (Value 1,000 bytes, 6,400,000 gets)

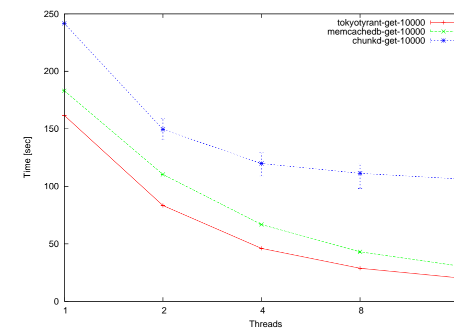

図 17 Benchmark result (Value 10,000 bytes, 640,000 gets)

© 2010 Information Processing Society of Japan

Vol.2010-HPC-124 No.14
2010/2/23

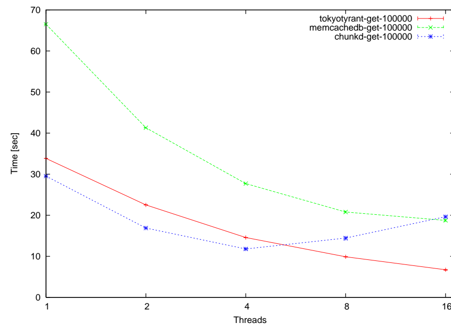

図 18 Benchmark result (Value 100,000 bytes, 64,000 gets)

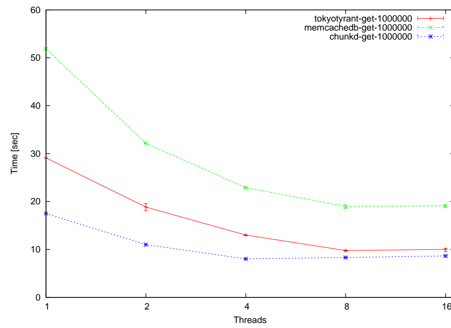

図 19 Benchmark result (Value 1,000,000 bytes, 6,400 gets)

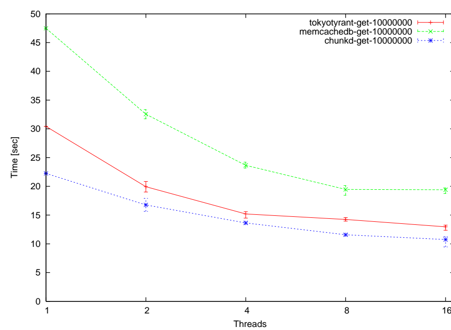

図 20 Benchmark result (Value 10,000,000 bytes, 640 gets)

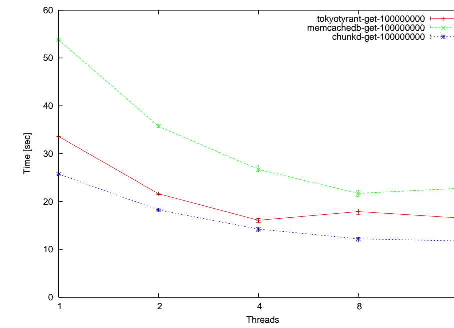

図 21 Benchmark result (Value 100,000,000 bytes, 64 gets)

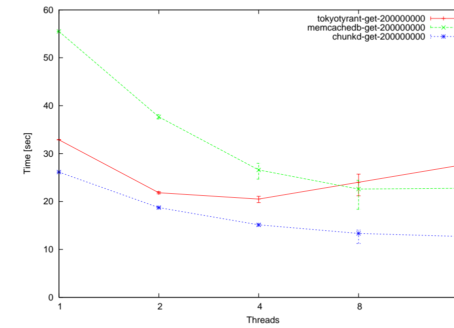

図 22 Benchmark result (Value 200,000,000 bytes, 32 gets)

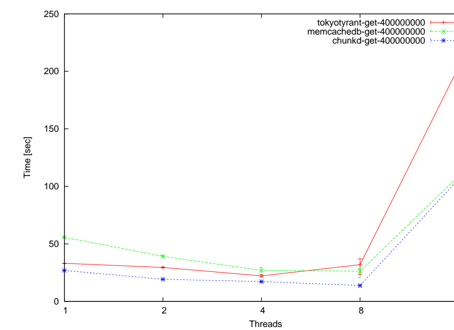

図 23 Benchmark result (Value 400,000,000 bytes, 16 gets)

© 2010 Information Processing Society of Japan