高速ストレージ/ネットワークによるHadoop MapReduceのベンチマーク

及川 一樹[1,a)] 本庄 利守[1,b)]

概要：ビッグデータの処理基盤として Google により提案された MapReduce と呼ばれる分散処理フレームワークが広く利用されている．特に，オープンソースソフトウェア実装である Hadoop は MapReduce のデファクトスタンダードとなりつつある．現行のコモディティサーバー上での MapReduce の実行では、ディスクやネットワークなどの I/O がボトルネックとなり、性能が律速されるケースが多かったが，ストレージやネットワークなどのハードウェアの進化が進むことで，I/O が高速になると，今後はボトルネックは CPU に移行することが考えられる．そこで，本論文では高速なストレージ／ネットワークを利用した MapReduce のベンチマークを行い，ボトルネックが CPU に移行することを示す．さらに，MapReduce 内部の各種基本的な処理に関して個別にベンチマークを実施し，現行の CPU が処理できるデータの処理速度を明らかにした．

Benchmark of Hadoop MapReduce using High Speed Storage and Network

Kazuki Oikawa[1,a)] Toshimori Honjo[1,b)]

1. はじめに

近年，ビッグデータの処理基盤として Google により提案された MapReduce[1] と呼ばれる分散処理フレームワークが広く利用されるようになってきた．特に，オープンソースソフトウェア実装である Hadoop[2] は MapReduce のデファクトスタンダードとなりつつある．現在の Hadoop MapReduce は Google により MapReduce が提案された 2004 年や Hadoop の開発がスタートした 2005 年におけるコモディティサーバとネットワークを想定した設計・実装となっており，ハードディスクやネットワークの I/O 性能がボトルネックとなることを想定したアーキテクチャとなっている．一方で，ストレージやネットワークなどのハードウェアの進化も進んでおり，高速なストレージである SSD(Solid State Drive) や高速なネットワークである 10G イーサネット・Infiniband などがコモディティ化されると，MapReduce におけるボトルネックは I/O から CPU へ移行することが予想される．

そこで，本論文では，高速なストレージ／ネットワーク環境で MapReduce ベンチマークを行い，ボトルネックが I/O から CPU に移行することを示す．そして，I/O がボトルネックとならない条件下で MapReduce 内部の各種基本的な処理に関して個別にベンチマークを実施し、現行の CPU が処理できるデータの処理速度の限界を明らかにする．

2. 課題

2.1 MapReduce の概要

MapReduce における処理の流れを図 **1** に示す．MapReduce は大きく分けて Map と Reduce と呼ばれる 2 つの処理から構成されている．Map 処理では分散ファイルシステムからデータを分割して読み込み，読み込んだデータに対して任意の処理を行うことで，Key と Value のペアを出力する．Reduce 処理では，Map 処理の出力より同じ Key を持つ Value を束ねることで，Key と Value 集合のペアに対

[1] 日本電信電話株式会社 NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center
[a)] oikawa.kazuki@lab.ntt.co.jp
[b)] honjo.toshimori@lab.ntt.co.jp

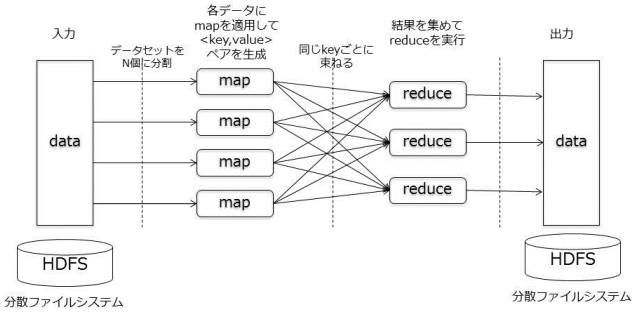

図 1 MapReduce の概要

して処理を行い，結果を分散ファイルシステムに書き出す．

Map と Reduce 処理内の処理シーケンスを図 **2** に示す．Map 処理では分散ファイルシステム上に蓄積されたデータを読み出し，そのデータをデシリアライズ・パースすることでプログラムが扱える形式に変換後，任意の処理を適用し，Key・Value ペアを生成する．生成した Key・Value ペアはメモリ上に蓄積され，一定量蓄積後に Key によるソートを実施し，ソート済みの Key・Value ペアのリストを Spill ファイルとしてローカルストレージに書き出す．以上を繰り返してデータの終端に達すると，メモリ上の Key・Value ペアをソートし，Spill ファイルとメモリ上データを Key 順でマージし，1 つのソート済みファイルを生成する．

Reduce 処理では各 Map 処理が出力したソート済みファイルをネットワーク越しに収集しローカルストレージに保存する．そして，それらのファイルを Key 順にマージすることで同じ Key に対応する全 Value を入手し，Key と Value 集合に対して集約演算等の任意の処理を行い，結果を分散ファイルシステムに書き出す．

以上の様な処理シーケンスにより，大規模なデータに対するフィルタ処理や集約演算などのさまざまなデータ処理を分散で実施することを可能としている．また，中間ファイルをローカルストレージに保存することによって，サーバが故障した場合でも，その故障したサーバが受け持っていたデータのみを再処理することで全体の処理を継続することができ，耐故障性も高く，コモディティサーバを大量に使った大規模データ処理に適している．

2.2 MapReduce の課題

MapReduce はコモディティサーバを前提に大規模データを効率的に処理するために考案されたフレームワークである．基本的な考え方として，現行 CPU の処理性能に比べて，ハードディスクの読み書き性能やネットワークの転送速度の方が遅いことを想定しており，コモディティサーバを大量に並べ，多くのハードディスクから並列に読み書きすることで，ディスク I/O 性能を稼ぐことを想定している．また，ディスク I/O に比べてネットワーク転送速度の方が遅いことや，ツリー型のネットワークトポロジを想定

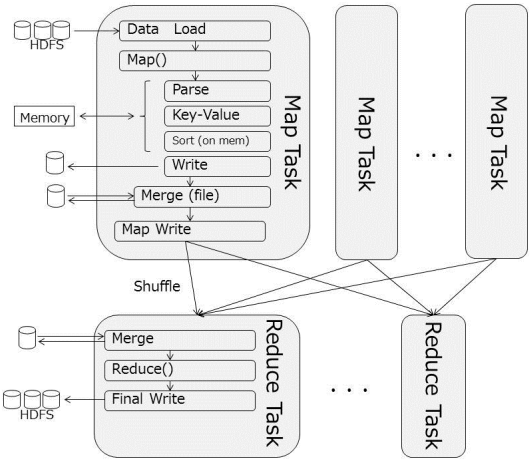

図 2 MapReduce の処理シーケンス

し，可能な限りローカルに保管されているデータを処理することを，分散ファイルシステムとの連携により実現している．

一方で，ストレージやネットワークの進化も進んでおり，SATA 接続の SSD は 1 台で 500MB/s の読み書きが可能となり，100MB/s 程度の性能しか出ない HDD と比べ 5 倍以上の性能を達成しているほか，サイズ・発熱・消費電力も少ないため HDD がよりも多くの台数を 1 台のサーバに設置することが出来る．また，SATA 接続の SSD の性能は SATA3.0 のインタフェース速度に律速されているため，PCI Express(PCIe) 接続の SSD も登場しており，数 GB/s の読み書き性能が可能になっている．ネットワークに関しても 10G イーサネットがコモディティ化しつつあるほか，Infiniband においては 56Gbps の通信が可能となってきている．

このように，最新のデバイスでは，従来に比べて性能が 1 桁以上向上しており，MapReduce のアーキテクチャが前提としてきた I/O ボトルネックが解消され，CPU がボトルネックとなる可能性がある．ボトルネックが CPU に移行したと仮定すると，前節にて述べた MapReduce を構成する各種処理のうち Map 処理におけるデータのデシリアライズやパーズ処理／ Key によるソート処理／ソート済みファイルのマージ処理や，Reduce 処理におけるデシリアライズ処理／ソート済みファイルのマージ処理などが CPU リソースを多く消費し，CPU ボトルネックとなって全体の性能を律速することが懸念される．

3. 高速 I/O 上の MapReduce ベンチマーク

第 2.1 章で概要を述べた MapReduce を，現時点でコモディティ化しつつある，高速なストレージ・ネットワーク上で動作させ，MapReduce のベンチマークとして著名な TeraSort を用いることで，高速なストレージ・ネットワークの性能を活用可能か評価を行った．

表 1　マシン構成

CPU	Intel Xeon E5-2695 (2.9GHz，8-core)×2
Memory	128GB (DDR3-1600，16GB×8)
HDD(SATA)	7200rpm (1TB)
SSD(SATA)	Intel SSD 520 (480GB)
SSD(PCIe)	Intel SSD 910 (800GB)
Network(eth)	1000BASE-T (OnBoard)
Network(ib)	Mellanox ConnectX-3 (FDR Infiniband)
OS	CentOS 6.3 (x86_64)
Java VM	Oracle JDK 6 Update 38
Hadoop	CDH 4.1.2

表 2　ネットワークスイッチ

Ethernet	1000BASE-T (16-port)
Infiniband	Mellanox MIS5022Q-1BRR (8-port QDR)

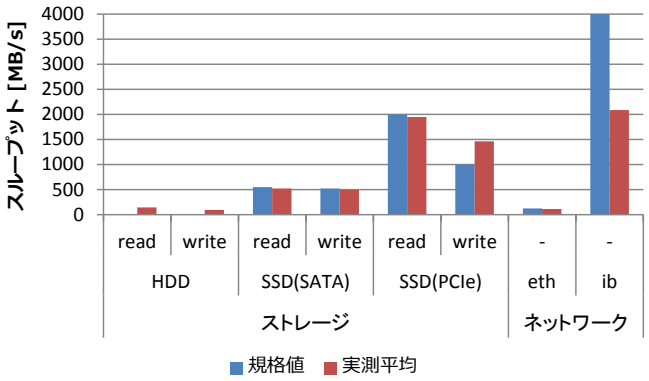

図 3　ストレージとネットワークの基本性能

3.1　評価環境

ベンチマークに利用したマシン構成を表 1，ネットワークのスイッチ構成を表 2 に示す．Infiniband は FDR 対応の小型スイッチが用意できなかったため，マシン側 Infiniband カードは FDR 対応だが QDR 動作で最大 32Gbps となる．従来のストレージ・ネットワークは HDD と Ethernet の組み合わせで，高速なストレージ・ネットワークは SSD(S-ATA，PCIe) と Infiniband を指す．

表 1 のマシンを 6 台用意し，1 台を NameNode などのマスター系プロセスを動作させるマスターサーバとし，残りの 5 台をスレーブサーバとした．

ストレージとネットワーク性能の規格値と実測値を図 3 に示す．ストレージの性能測定には fio(flexible I/O tester) を利用し，ネットワークの性能測定には NetPerf を用いた．SSD(PCIe) の書き込みスループットが規格値を超えているのは，Intel SSD 910 のハイパフォーマンスモードを有効にしているためである．Infiniband の実測平均値が規格値を大きく下回っているのは，TCP/IP スタックを経由しているためと思われるが，どのストレージよりも高速であるため本ベンチマークでは問題にならないと考える．

また，各ベンチマーク共通の主な設定値を表 3 に示す．Map タスク数は CPU の論理コア数より DataNode と TaskTracker 用に 2 スレッド減算した値をスレーブサーバの台数で掛けたものを設定した．Reduce タスク数は HyperThreading を考慮し，物理コア数を 1.5 倍した後，DataNode/TaskTracker 用に 2 スレッド分を減算し，Map 処理と Reduce 処理の重なりを考慮し，その値のおおよそ半分を設定した．

表 3　ベンチマーク共通の設定値

パラメータ名	設定値
Map タスク数	150
Reduce タスク数	50
中間ファイル圧縮	有効 (Snappy)

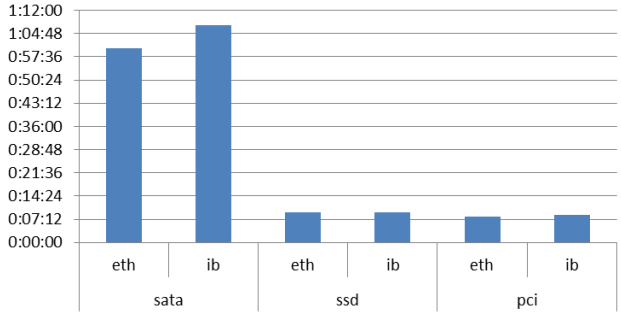

図 4　TeraSort の測定結果 (実行時間)

3.2　TeraSort によるベンチマーク

予め TeraGen を用いて生成した 400GB(40 億レコード) のデータに対する TeraSort の測定結果を図 4 に示す．

ネットワークの帯域幅には影響を受けず，HDD と SSD の性能差が大きく影響する結果となった．

TeraSort 実行時の CPU 利用率を図 5，図 6，図 7 に示す．HDD を利用した場合 (図 5) は I/O wait が CPU 時間の大半を占めており HDD がボトルネックになっていることが判る．それに対し，SSD を利用した場合 (図 6，図 7) は，CPU 時間の大半をユーザ空間で消費しており，CPU にボトルネックが移行していることが確認できる．SSD(SATA) を利用している場合は Reduce 処理にて I/O wait が発生しているが，SSD(PCIe) を利用している場合は，ほとんど I/O wait が発生していない．SSD(PCIe) の読み書きスループット (図 8) を確認しても，SSD(PCIe) の基本性能 (図 3) に達しておらず，SSD(PCIe) を利用した場合は TeraSort の全処理において CPU がボトルネックになっていると言える．

以上から，高速なストレージやネットワークを用いた MapReduce においては I/O ではなく CPU がボトルネックになることがベンチマーク実験により示された．

4.　MapReduce 内部の各基本処理に対するベンチマーク

本章では，2.1 章にて述べた，MapReduce 内部の各基本処理に対して，個別にベンチマークを実施し，CPU による

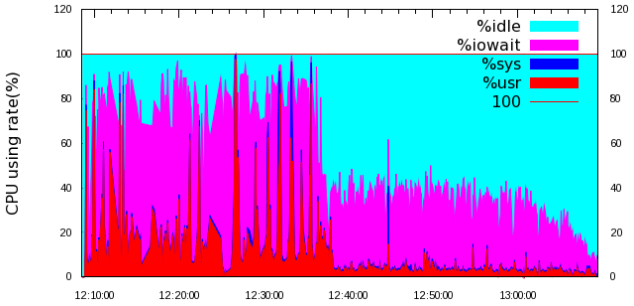

図 5　TeraSort 実行時の CPU 利用率 (HDD+eth)

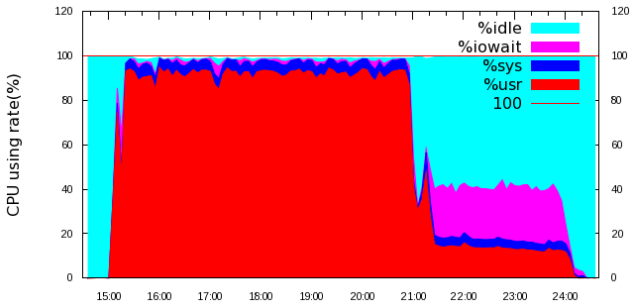

図 6　TeraSort 実行時の CPU 利用率 (SSD(SATA)+eth)

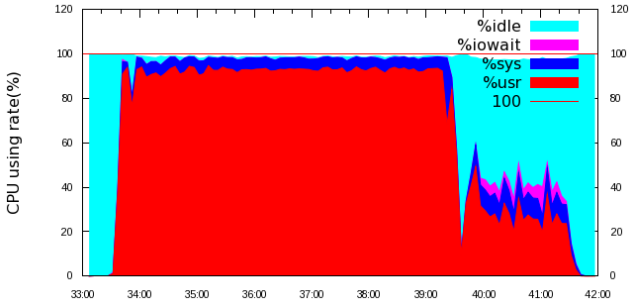

図 7　TeraSort 実行時の CPU 利用率 (SSD(PCIe)+eth)

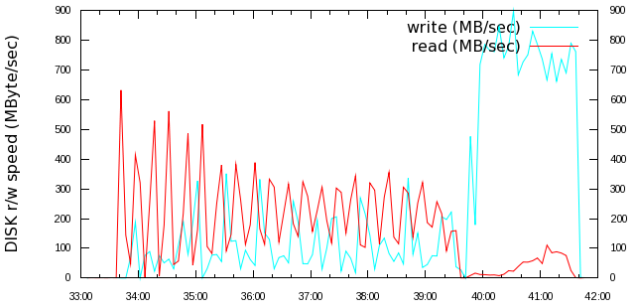

図 8　TeraSort 実行時の SSD(PCIe) 読み書きスループット

データ処理性能を評価し、MapReduce 内部の CPU ボトルネック箇所を明らかにする．

4.1　データ読み込み処理

Map 処理における分散ファイルシステムからの入力データ読み込みのほかにも，ソート済みファイルの読み書きなど，Map/Reduce 処理内部の複数箇所で行われるファイル読み書き処理のうち，CPU (1 コア) が処理することが出来る最大スループットを評価する．

4.1.1　評価方法

ファイルアクセスの方法として以下の 5 種類を利用し評価を行った．

- Java を利用しない方法
  (1) cat (GNU coreutils 8.4-19)
- Java 標準の方法
  (2) java.io.FileInputStream クラス
  (3) java.nio.channels.FileChannel クラス (NIO)
- Hadoop ファイルシステム抽象化レイヤを通した方法
  (4) org.apache.hadoop.fs.local.LocalFs クラス
  (5) org.apache.hadoop.fs.Hdfs クラス

複数の方法を使って評価することで，評価に用いたマシンでの限界性能と，性能が律速されるレイヤの特定も同時に行う．

評価では I/O がボトルネックとならないようにファイルをメモリ上のファイルシステムである tmpfs 上に配置したほか，HDFS 経由でアクセスする場合も I/O がボトルネックとならないようにローカルホスト上に DataNode を起動させ，チャンクの保管場所として tmpfs を指定した．また，メモリ上に保存される tmpfs とのデータのやり取りであるため，書き込み性能については評価せず，読み込み性能のみを評価した．

4.1.2　評価結果

評価結果を図 **9** に示す．図の横軸の番号は上記のアクセス方法の番号を表す．評価マシンが搭載しているメモリは DDR3-1600 であるためチャンネルあたり最大 12.8GB/s の帯域幅を持つが，tmpfs のオーバーヘッド等もあり，cat では 5.5GB/s となった．この値を基準とすると，NIO の FileChannel を使った方法では Java 上でのメモリコピーが削減されているため，cat よりもやや低い 4.9GB/s にとどまっている．FileInputStream や LocalFs を用いた場合は NIO を使っていないためメモリコピーが発生し，FileChannel よりも低い結果となった．HDFS を経由した場合は，Java 内部でのメモリコピーのほかにも，Loopback とはいえネットワーク通信に伴うメモリコピーも発生するため，大きくスループットを落とし 1.5GB/s となった．

この評価プログラム実行中の CPU 利用率を図 **10** に示す．全ての方法において評価プログラムのプロセスは CPU 1 コアを 100%専有したほか，HDFS 経由の場合は DataNode のプロセスも CPU 1 コアを 100%占有している．HDFS から 1.5GB/s のスループットでデータを読み込む場合はそれだけで CPU 2 コアを占有するので，高速なストレージ上で HDFS を利用する場合はファイル読み込みだけでも CPU リソースを多く消費する事になると思われる．

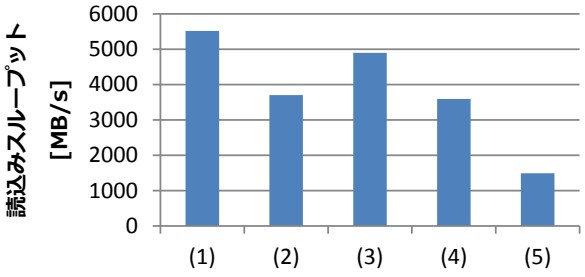

図 9　ファイル読み込み性能

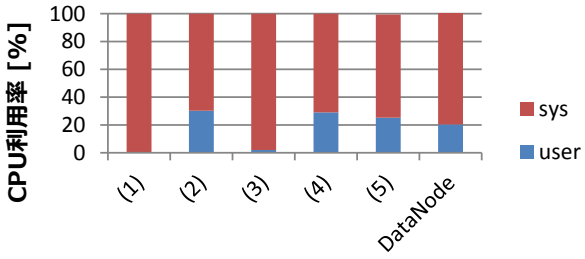

図 10　ファイル読み込み時の CPU 利用率

また，CPU 利用率のグラフ (図 10) を見るとユーザ空間よりもカーネル空間が占める割合が大きいため，数 GB/s というスループットになるとカーネル内でのメモリコピー等の処理がボトルネックとなっていると考えられる．

4.2　データのデシリアライズ処理

MapReduce を使ったアプリケーションを開発する場合，大抵の場合はバイナリデータをそのまま扱うのではなく，数値や文字列といった型に変換してから様々な処理を行う．Hadoop では数値や文字列といった基本的な型について，シリアライズ可能なクラスを提供しており，通常は Hadoop が提供するシリアライズ可能クラスを用いるため，本節では，基本的な型のシリアライズ／デシリアライズ性能を評価する．

4.2.1　評価方法

整数や浮動小数点型，可変長バイナリ型は 1GB 分のデータのシリアライズ／デシリアライズを行うことで評価を行い，文字列型については ASCII と UTF-8 符号化方式の 2 種類について 1GB 分の文字列の符号化／復号を行うことで評価を行った．可変長バイナリ型については要素サイズを 8B から 1KB の間で変化させたほか，文字列型では UTF-8 にて符号化／復号する対象データを ASCII コードの範囲の文字列と日本語からなる文字列の 2 種類について評価した．

4.2.2　評価結果

評価結果を**図 11**，**図 12**，**図 13** に示す．整数型等のシリアライズ性能は Bool や Byte 型は少し特殊だが，それ以外は要素サイズに関わらずおよそ秒間 40M 回のシリアライズが可能となっている．しかし，要素サイズが小さいためバイト数でのスループットに換算すると 8 バイトと最大

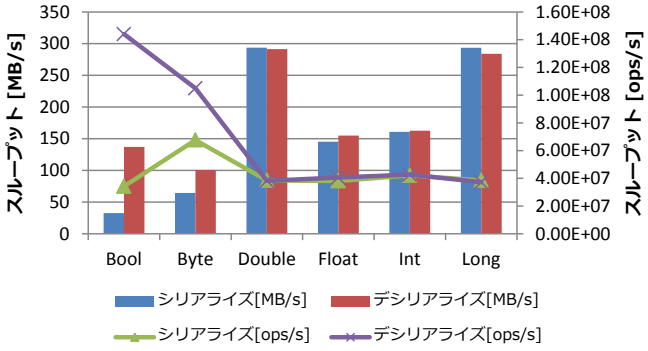

図 11　シリアライズ性能

の要素サイズである Long/Double 型であっても 290MB/s と CPU 1 コアあたりのスループットはそれほど高くはない．これはバイナリ型にも同じ事が言え，要素サイズが大きくなれば 7GB/s に達するが小さい場合だと 290MB/s にとどまる．秒間 40M 回のシリアライズを行なっているという事は，評価マシン CPU の動作周波数 (ターボブースト時 3.8GHz) より，1 要素のシリアライズに 90 クロックも消費していることになり，整数値からバイナリへ変換するのに必要な命令数を考慮しても，必要以上にクロックを消費している様に思える．

文字列の符号化性能については，Java は内部的に UTF-16 で保持しているため ASCII 符号方式であっても単純なメモリコピーでは変換できず 1.5GB/s となている．UTF-8 の場合は対象の文字列が ASCII コードの範囲内であれば，1.0〜1.5GB/s のスループットで符号化／復号できるが，日本語など符号化の結果がマルチバイトになるようなコードポイントを含むと 230MB/s と大きくスループットが低下する．ASCII 符号化方式の復号が UTF-8 よりも遅く，半分程度のスループットとなる原因は不明だが，本来であれば UTF-8 より遅くなることは無いと思われる．

MapReduce を使ったアプリケーションにおいて文字列操作を行う場合は，一度可変長バイナリ型 (Text) に変換後，String 型に変換を行う．64B の英数字を Key または Value として扱う場合はデシリアライズ処理と文字列の復号処理を行うため，スループットは 800MB/s 程度となり I/O 性能を活用できそうだが，日本語等を含む場合は 200MB/s となるため，数 GB/s の I/O 性能を活用するためには，テキストの読み込みだけで 10 コア以上を必要とする計算となる．

4.3　データのパース処理

MapReduce を使ってテキストログ分析などを行う場合，通常はログファイルを 1 行ずつ読み込み，読み込んだ 1 行文字列を Map 処理の入力とする．続いて Map 処理内では 1 行文字列を区切り文字等で区切り，日付やログレベルなど様々な情報に分割し，フィルタリング等を行う．

本節では前述のようなパース処理を行うのに必要な行単

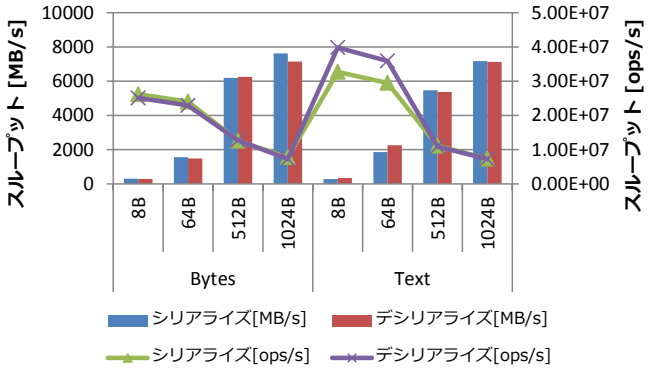

図 12 シリアライズ性能 (バイナリ型)

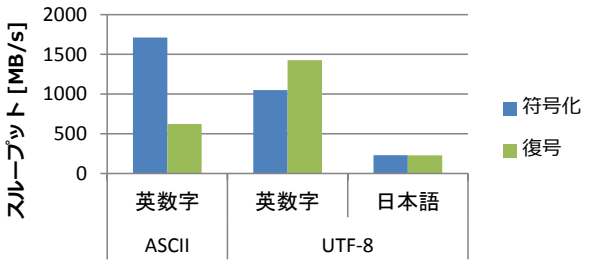

図 13 文字列の符号化／復号性能

位読み込みと，区切り文字分割のスループットを評価する．

4.3.1 評価方法

Wikipedia 英語版のダンプデータより本文のみを抽出し，1 行が 256 バイト以下となるように整形したテキスト 4GB 分を tmpfs に保存し，tmpfs 上のファイルに対して行読み込み／スペースを区切り文字とした分割スループットの評価を行った．

行読み込み／分割方法は以下の 6 種類用意した．

- 行読み込み方法
  - (1) java.io.BufferedReader クラス
  - (2) Hadoop の TextInputFormat クラス (バイナリ)
  - (3) Hadoop の TextInputFormat クラス (文字列)
- 区切り文字分割方法
  - (4) java.util.StringTokenizer クラス
- 行読み込みと分割の組み合わせ
  - (5) BufferedReader + StringTokenizer
  - (6) TextInputFormat + StringTokenizer

StringTokenizer 単体の評価は Java の制約により 4GB の文字列が確保できなかったことと，256MB など大きめの文字列を指定すると異常に性能が劣化したため，行読み込みと組み合わせた時と同じ条件である 256B の文字列に対する処理を 4GB 分繰り返すことで評価した．また，TextInputFormat は LocalFs クラスを利用して tmpfs からファイルを読み込んだ．

4.3.2 評価結果

評価結果を図 **14** に示す．横軸の数字は評価方法にて列挙した 6 種類の方法を表す．

最も性能が良い BufferedReader や TextInputFormat(バイナリ) であっても，秒間 400MB/s 程度と，4.1 節にて評価したファイル読み込みスループットと比較すると大きく低下している．TextInputFormat(バイナリ) はファイルからデータを読み込み，バイナリデータに対して直接，行の終端を検索し 1 行分のバイナリデータのコピーを行なっている．4.1 節の結果よりファイル読み込み性能を除外し，TextInputFormat 単体のスループットを計算すると 500MB/s となるため，行の終端検索と行単位のメモリコピーが律速となっていると考えられる．

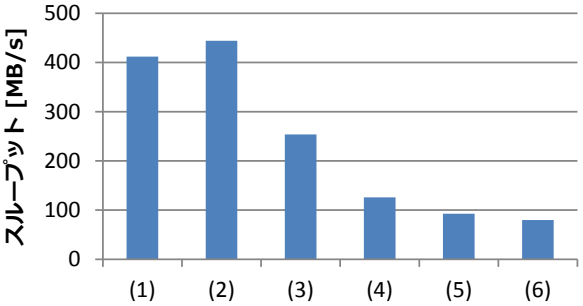

図 14 行読み込み／区切り文字分割性能

また，StringTokenizer 単体による空白文字による単語分割も 125MB/s 程度のスループットに留まったため，行読み込みと単語分割を組み合わせにおいても，80MB/s 程度と低いスループットとなった．

4.4 ソート／マージ処理

Map 処理の結果をソート済みの中間ファイルとして出力する箇所や，Reduce 処理の入力となる複数のソート済みファイルを Key 順にマージしながら読み込む箇所に利用される，ソート及びソート済みデータのマージスループットについて評価する．

4.4.1 評価方法

Hadoop より Map 処理の出力をソートするクラス (MapOutputBuffer) を抜き出し，MapReduce を通してではなく，ソート及びマージの単体性能の評価を行う．

ソート対象となるデータは，小さい Key(4 バイト) ／大きめの Key(TeraSort の Key と同じ 10 バイト) と，小さな Value(4 バイト) ／大きめの Value(TeraSort の Value と同じ 90 バイト) を組み合わせたものを利用する．なお小さな Key としては比較が高速と思われる 32bit 整数型と Text 型 (内部はバイナリ) の 2 種類，大きな Key は Text 型のみとし，Key は乱数により生成した．

Spill ファイルや最終結果となるソート済みファイルの出力先は tmpfs を利用し，データサイズは 8GB とした．

4.4.2 評価結果

評価結果を図 **15** と図 **16** に示す．図 15 はソートとマージのスループットを分けたもので，図 16 はソートとマージの合計スループットとなっている．

図 15 より，1 秒間で処理できる Key 数をソートとマー

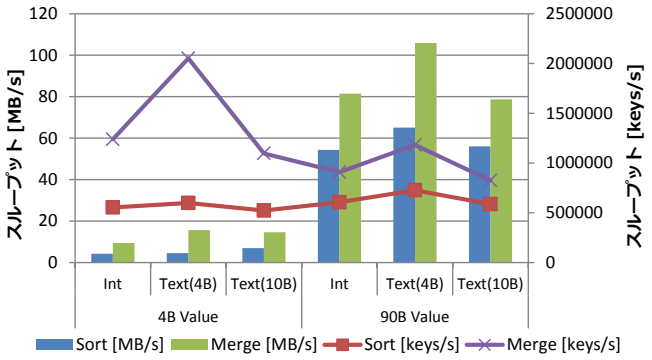

図 15　ソート／マージ性能

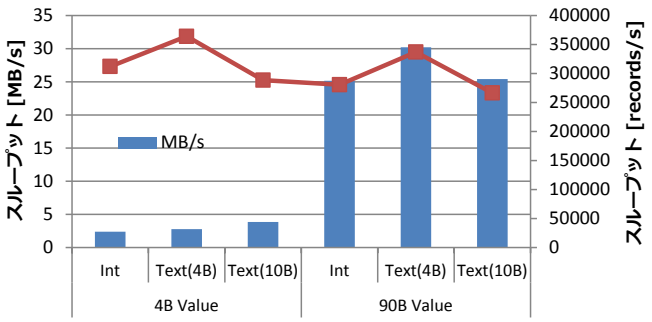

図 16　ソート／マージ合計性能

ジで比較すると，Value サイズが 4 バイトの時はマージのほうが 2 倍程度高速となっている．Value サイズが 90 バイトの時に，ソートとマージの性能差が縮まっている理由は，ソートは Value サイズに依存しないのに対し，マージは Key ／ Value ペアのコピーを行うため Value サイズにも依存しているからである．今回の評価では Key を乱数によって生成したため Key サイズが大きくなっても比較処理自体は Key 全体を比較すること無く終わってしまう可能性が高く，4 バイトと 10 バイトで大きな差は現れなかったが，当初の想定に反し，同じ Key サイズであれば Int 型よりも Text 型の方がソート／マージともに高い性能を示した．Hadoop のソート用メモリ領域はシリアライズされた形式で保持されているため，内部表現がバイナリである Text はシリアライズされた形式のまま直接バイナリ同士で比較ができるが，Int 型の場合はシフト演算等を用いてバイナリから整数型に変換してから比較する必要があるため，比較のたびに発生するデシリアライズがオーバーヘッドとなった可能性が考えられる．

1 秒間でソートできる Key 数は Value サイズに依存しないため，1 秒間でソートできるデータサイズは Value サイズに大きく依存したものとなる．WordCount など入力データを細かく分割したものを Key とし小さな Value を付与するアプリケーションでは，単位時間あたりで処理できるデータサイズは非常に小さいものとなるが，TeraSort など入力データを分割し小さな Key と大きな Value とする場合は，ソートやマージの段階でもスループットを維持できる可能性がある．しかし TeraSort の Value サイズではソートは 60MB/s，マージは 105MB/s であるため，Map 処理の出力スループットは図 16 により 30MB/s に留まる．

4.5　基本処理に対するベンチマークのまとめ

4.1〜4.4 節での評価結果をまとめると，ソート／マージ処理のコストが最も高く，続いて区切り文字分割や行単位読み込みといった文字列のパース処理，小さいサイズのシリアライズ処理，日本語等のマルチバイト文字の符号化処理のコストも高い結果となった．

これらコストの高い処理を全て利用する WordCount では，CPU のコアあたり最大 2.8MB/s 程度のスループットであると推定されるため，現行 CPU のコア数・ソケット数では SATA 接続の SSD の性能を活用できず，CPU がボトルネックとなる．入力したデータの大半をフィルタリングしてしまうようなテキストログ分析を想定しても，ソートのコストは無視できるが，入力データのデシリアライズ・パースのコストが高いため，39MB/s 程度のスループットに留まり，PCIe 接続の SSD 性能を活用する為には Map 処理だけで 50 コア以上が必要という計算になる．

5.　考察

4 章の評価結果を元に，3.2 章で評価した TeraSort の結果を考察する．TeraSort は以下のような処理によって構成されている．

(1) ファイルを HDFS 上から読み込む
(2) 読み込んだデータを固定長 Text でデシリアライズし Key(10B)/Value(90B) ペアを作成
(3) 上記 Key/Value をそのまま Map 処理の出力としソート及びマージを行い，ソート済みファイルを作成 (シリアライズ+ソート・マージ)
(4) Reduce 処理側では各サーバに点在しているソート済みファイルをローカルにコピー
(5) ローカルにコピーしたソート済みファイルをマージし結果を HDFS 上に書き出す (マージ+ファイル出力)

4 章の評価結果を上記の処理にあてはめると，
(1) 1490MB/s　(2) 342MB/s　(3) 280MB/s，30MB/s
(4) 3600MB/s　(5) 106MB/s，1490MB/s となる．今回の TeraSort の評価では Snappy による圧縮を有効にしたので，Snappy による圧縮と展開を 250MB/s，500MB/s とした [3]．また，4 章の評価では CPU 1 コアをターゲットとしたベンチマークを行ったため，評価マシンの CPU はターボブーストが働き動作周波数が定格よりも 1.31 倍上昇している．3.2 章で評価した際は Map や Reduce 処理を複数プロセスで同時に実行しているため，ターボブーストは働かない．以上の点を考慮すると，4 章の評価結果より算出される Map 処理のスループットは 15.1MB/s，Reduce 処理のスループットは 61.6MB/s，Map・Reduce 処理を合わせた一連のスループットは 12.1MB/s となる．

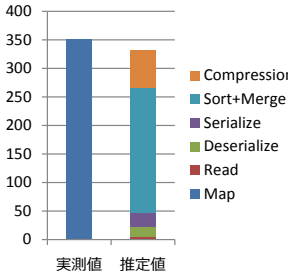

図 17　Map 処理時間内訳

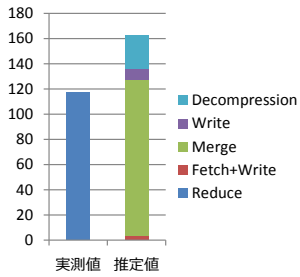

図 18　Reduce 処理時間内訳

HDFS への読み書きやソート部などは複数スレッドを利用する点と，図 8 によると，Reduce 処理は CPU を半分しか消費していない点を考慮し，Map 処理では CPU 物理コア数，Reduce 処理では CPU 物理コア数の半分を，上述の予測値に適用すると Map 処理および Reduce 処理時間の内訳は**図 17** と**図 18** となる．Reduce 処理側は 2 割程度の違いがあるが，Map 処理側は概ね実測値と同じ値となり，4 章にて求めた基礎性能をもとに積み上げることによって実際のプログラムの性能を大まかに推定できた．

以上より，4 章での評価の結果，Map 処理・Reduce 処理ともにソート及びマージ部が処理時間の大半を占めており，ここがボトルネックとなっていることがわかった．ストレージやネットワークが高速化した場合はここを何らかの方法で解決しなければ，I/O 性能に見合った MapReduce の処理時間短縮は望めない．

6. 関連研究

Hadoop のベンチマークに関する研究は多数行われているが，内部まで踏み込んだ解析を行なっているものは少ない．文献 [4] では，MapReduce における CPU 利用率に着目したベンチマークを実施しており，Map 処理においては行単位でパースするコストとソートのコストについて分析がなされており，ソート処理に多くの CPU 時間が割かれていることが指摘されているが，行単位のパースコスト分析ではパース処理単体での分析を行わなかったため，他の部分の影響により無視できるものとしてしまっている．また，文献 [6] では RDD(Resilient Distributed Datasets) と呼ばれる分散共有メモリを用いた MapReduce の高速化を提案している．本論文の中で，デシリアライゼーションのオーバーヘッドが思いのほか大きいことを指摘している．

7. まとめ

本論文では，ストレージやネットワークなどの I/O が高速化すると，ボトルネックが CPU に移行することを示した．加えて MapReduce 内部の各種の基本的な処理に関して個別にベンチマークを実施し，現行の CPU が処理することの出来る，ファイルの読み込みや，シリアライズ・パース処理，ソート・マージの性能 (スループット) 限界を明らかにした．

MapReduce 内部の個々の処理の性能分析の結果，ソートやマージのコストが非常に高いことや，パースや分割，文字列の符号化・復号のコストも高いことがわかった．前述の処理を全て使う，ログ分析といった行単位・区切り文字分割を利用するアプリケーションの場合，数 GB/s の性能が出る非常に高速なストレージでなくとも S-ATA 接続の SSD 程度の I/O 性能で CPU がボトルネックとなる可能性が高いことを示した．そして，個々の処理性能の積み重ねと Map ／ Reduce の処理時間と比較することで，Map ／ Reduce 処理の大半がソート及びマージに割かれている事を示した．

今後の課題としては TeraSort だけでなくシリアライズを多用するワークロードでも個々の処理性能から推定される性能と近い結果となるかの評価や，今回明らかになったボトルネックに関して，データの処理方法などの工夫を通じて現行の CPU 上での性能改善手法の検討などがあげられる．

参考文献

[1] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, Proc. Sixth Symposium on Operating System Design and Implementation (OSDI’04), pp.137-150 (2004).

[2] Welcome to Apache Hadoop! (online), http://hadoop.apache.org.

[3] snappy - A fast compressor/decompressor (online), http://code.google.com/p/snappy/.

[4] Mazur, E. Li, B. Diao, Y. and Shenoy, P.: Towards Scalable One-Pass Analytics Using MapReduce, Proc. IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW ’11), IEEE, pp. 1102-1111 (2011).

[5] Jiang, D. Ooi, B.C. Shi, L. and Wu, S.: The performance of MapReduce: an in-depth study, Proc. VLDB Endowment, Volume 3 Issue 1-2, September 2010, pp. 472-483, (2010).

[6] Zaharia, M. Chowdhury, M. Das, T. Dave, A. Ma, J. McCauley, M. Franklin, M. J. Shenker, S. and Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, Proc. 9th USENIX conference on Networked Systems Design and Implementation (NSDI’12), 2-2, (2012).