Vol.2009-HPC-121 No.42
2009/8/6

高速フラッシュメモリ向け MapReduce フレームワークの実現に向けて

小川宏高[†1] 中田秀基[†1] 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．DISC は，安価なハードディスクと Gigabit Ethernet を用いたクラスタシステムを用いて，大容量データをストリーム的に扱うワークロードを実行するシステムとして非常に有効である．一方で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高速なフラッシュメモリを利用したストレージクラスタに適した，DISC システム SSS を設計・実装することを目指している．本稿では，既存の DISC システムの問題点を考察した上で，我々がプロトタイプ設計している SSS フレームワークのアーキテクチャを説明する．

Towards Fast Flash Memory Based MapReduce Framework

Hirotaka Ogawa,[†1] Hidemoto Nakada,[†1]
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. DISC systems are quite effective, especially when executing large-scale streaming data workload on an inexpensive cluster with hard drives and gigabit ethernet. 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, instead of cheaper and slower hard drives. However, coupling such very fast storage devices with DISC systems, much of the benefits of devices can easily be lost because of software overhead incurred by DISC 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 identify the problems of existing DISC systems and describe the architecture of "SSS" framework.

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．Google 社の Dean らによって開発された大規模データ処理のための分散処理プログラミングフレームワークである 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 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．しかしながら，key-value 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．また，定型的な分散処理しか許さないために，データの入出力と計算を重ね合わせによるオーバーヘッドの隠蔽に代表される，HPC システムでは常套手段となっているような最適化を施す余地も限定的なものとなっている．

これは，DISC システムの「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．逆の言い方をすると，DISC システムを構成するソフトウェアスタックがこのようなワークロードに特化してスクラッチから実装されてきたためである，と言ってもよい．

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．とりわけ，Fusion-io 社の ioDrive™[3] duo のように PCI-Express をインタフェースとして利用する SSD は，約 10Gbit/sec の読み書き性能を達成しており，これは主記憶

†1 産業技術総合研究所 / National Institute of Advanced Industrial Science and Technology (AIST)

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．つまり，我々はハードディスクと主記憶の中間に位置するメモリ階層を新たにコモディティとして利用できるようになりつつある．

このようなフラッシュメモリを DISC システムに適用することで得られるメリットは自明かつ重大である．まず，ハードディスクをフラッシュメモリに置き換えることで消費電力の低減の恩恵が受けられる．さらに一定の I/O 性能を得るのに必要なストレージノード数が少なくて済むため，機器自体の個数を 1/10 程度にまで圧縮することができる．

しかしながら，単純に置き換えるだけでは十分なメリットが得られないこともまた自明である．これまではハードディスクのアクセス速度がボトルネックとなっていたため，上述のように DISC システムが読み書き性能に関して十分な最適化を施していなかったとしても問題とはならなかった．これに対して，フラッシュメモリを利用する場合には，ハードウェア的なボトルネックが解消する反面，ソフトウェアスタックによるオーバーヘッドが顕在化することになる．

このような観点から，我々は 10Gbit/sec クラスの読み書き性能を持つ高速なフラッシュメモリを利用したストレージクラスタに適した，DISC システムを設計・実装することを目指している．

本稿では，既存の DISC システムの問題点を考察した上で，我々がプロトタイプ設計している SSS フレームワークのアーキテクチャの概要を説明する．

2. DISC システム

Data-Intensive Scalable Computing (DISC) システムは，大規模なデータインテンシブアプリケーションの実行を支援するフレームワークである．DISC システムでは，アプリケーションは主に共有ストレージシステムに格納された大量のデータセットに対する並列処理を特徴とする．このようなフレームワークとしては，MapReduce[1], Hadoop[4], Dryad[5] が広く知られており，いずれも大量の計算を多数のタスクに分割した上で，共有ストレージシステム上の入力データのローカリティを考慮して計算ノードに割り付けて実行する機能を提供している．

また DISC システムは，実アプリケーションの利便を図るため，共有ストレージ・並列処理フレームワークだけでなく，包括的なソフトウェアスタックを提供している．例えば，Google や Yahoo は自身のインターネットサービスを，定型的な分散処理を実現するスタック (MapReduce, Hadoop MapReduce)，計算処理をプログラムするためのスタック (Sawzall, Pig)，構造データをストアするスタック (Bigtable[6], HBase[7]) 上に構築している．また，これらのスタックはそれぞれ GFS (Google File System), HDFS (Hadoop Distributed File System)[8] と呼ばれる分散共有ファイルシステム上に実装されている．

ここではこれらのソフトウェアスタックのうち，GFS/HDFS と Google MapReduce/Hadoop MapReduce について概要を述べた上で，DISC システムの問題点を述べる．

2.1 GFS/HDFS

GFS は，多数のストレージノード上に構築される分散ファイルシステムの一種であり，大容量，スケーラビリティ，耐障害性などの特徴を持つ．

GFS では，ファイルを複数のチャンク (デフォルトのサイズは 64MB) に分割してストレージノードのローカルディスクに格納し，これらのチャンクを仮想的に統合することで巨大なファイルを実現している．

GFS の構成要素は，チャンクを格納するチャンクサーバとそれらを管理するマスターである．チャンクサーバはローカルディスク上に格納された個々のチャンクを管理する一方，マスターはファイルのディレクトリ構造，ファイルを構成するチャンクの情報，チャンクとチャンクサーバのマッピング情報などのメタ情報を管理している．

GFS ではチャンクを格納する際にチャンクのレプリカを複数のチャンクサーバにコピーして保持する．この多重化は耐故障性と性能の両方に寄与する．チャンクサーバの一部が障害を起こしても他のチャンクサーバから復元できるだけでなく，物理的に近いチャンクサーバからチャンクを取得することで通信量を削減できる．

GFS のオープンソース実装である HDFS はほぼ GFS に準じた機能を提供する．HDFS では，チャンク，チャンクサーバ，マスターをそれぞれブロック，DataNode，NameNode と呼んでいる．

GFS/HDFS の HPC 向けの並列ファイルシステム (IBM GPFS[9], Panasas PanFS[10], PVFS2[11], Lustre[12] など) との際立った差異は，POSIX に準拠した API を提供しないことである．特に GFS/HDFS では，チャンク単位でのデータの読み書きしかできない．また，並列ファイルシステムが専用のストレージサーバ上に round-robin などの固定的なポリシーにしたがって暗黙にレプリカを割り当てるのに対して，GFS/HDFS はラックを考慮してランダムに計算ノードにレプリカを割り当て，その割り当て情報をユーザや上位のミドルウェアに開示する．

こうした設計が採られているのは，読み書きスループットの最大化，チャンク (ブロック) のローカリティを利用した計算の最適化，ノード全体でのデータのアクセスと計算のロード

バランスなどを実現するためである．

2.2 Google MapReduce/Hadoop MapReduce

MapReduce は，key-value 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 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，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 key-value ペアから中間データを生成する．中間データは key-value ペアのリストとする．Shuffle 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 key-value ペアのリストを生成する．

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．

Google の実装の構成要素は，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 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 GFS/HDFS を利用することを前提としており，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リ当てを行う．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，通信コストを抑制している．

2.3 DISC システムの問題点

ここまで述べてきたように，MapReduce 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，その入出力に GFS/HDFS 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になる．

これにはいくつかの問題がある．ひとつはこのようなデータはチャンクの先頭から読むことしかできないことであり，もうひとつは入出力のたびにデータのマーシャリング・アンマーシャリングのコストを支払わねばならないことである．実際，Hadoop の実装ではチャンク単位でのランダムアクセスしか許しておらず，(MapReduce 操作を繰り返すなどして) 同一チャンクに複数回アクセスする場合でもクライアントサイドでキャッシュするなどしない限り，マーシャリング・アンマーシャリングのコストを回避する方法がない．

また，MapReduce が定型的な分散処理しか許さないために，データの入出力と計算を重ね合わせによるオーバーヘッドの隠蔽に代表されるような，HPC システムでは常套手段となっているような最適化を施す余地も限定的になるという問題も指摘し得るであろう．

他にも Hadoop MapReduce のチャンク (ブロック) のローカリティを考慮したタスクの割り当てが，不完全なデータ・アフィニティの実現に留まっている点も指摘できる．

これらの問題は，DISC システムの安易な前提条件，すなわち，

- DISC システムの「典型的な」ワークロードが大容量データをストリーム的に扱うものに限定されており，かつ
- DISC システムの「典型的な」実行環境が安価なハードディスクと Gigabit Ethernet を用いたクラスタシステムに限定されている

条件下ではほとんど無視し得るとも言える．

このようなワークロードにおいては，クライアントキャッシュは無意味であり，またストレージやネットワークの I/O バンド幅がボトルネックになっている状況では，処理系の性能は多数のストレージノードの I/O バンド幅の総計だけにほとんど依存することになるからである．当然，マーシャリング・アンマーシャリングのコストを抑制できたり，データの入出力と計算を重ね合わせることができたりしたとしても得られる速度向上はごく小さいものとなる．

また，ハードディスクで構成されたストレージノードと Gigabit Ethernet の組み合わせでは，ストレージとネットワークのスループットが比較的均衡しているため，ワーカー (TaskTracker) とチャンクサーバ (DataNode) 間の通信においてラック間通信を抑制できさえすれば実用上は十分であると言ってもよい．

しかしながら今日，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^{TM}[3] duo のように PCI-Express をインタフェースとして利用する SSD は，約 10Gbit/sec の読み書き性能 (実効で Read 700MB/sec, Write 600MB/sec) を達成しており，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．

つまり，我々はハードディスクと主記憶の中間に位置するメモリ階層を新たにコモディティとして利用できるようになりつつある．このことは，上記の安易な前提条件が成立しなくなることを意味しており，したがって DISC システムの非効率さが顕著になると我々は考える．

3. フラッシュメモリを用いた高性能 DISC システム

我々は，10Gbit/sec クラスの読み書き性能を持つ高速なフラッシュメモリに見合った，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．これにより，2.3 で述べたソフトウェアスタックの問題を解決するだけに留まらず，HPC を含めたより広範なアプリケーション分野への DISC システムの適用を可能にすることを目論んでいる．

以下では，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「SSS」の設計について述べる．

3.1 設 計 指 針

SSS では，以下に示す設計指針を採る．

3.1.1 既存の DISC システムのレイヤリングの踏襲

既存の DISC システムが与えるような包括的なソフトウェアスタックを実現することで実アプリケーションの利便を図る．これには，共有ファイルシステムに相当するスタック，分散処理を実現するスタック，計算処理をプログラムするためのスタック，構造データをストアするスタックを含む (図 1)．

以降は，これらのうち，分散ファイルシステムに相当するスタック，分散処理を実現するスタックについてのみ検討する．

3.1.2 分散 key-value ストアとしての共有ストレージシステムの実現

MapReduce は本質的に key-value ストア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 key-value ペア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，両者のセマンティックギャップがソフトウェアオーバーヘッドの原因になっていると言ってよい．

この問題を解決するために，SSS は共有ストレージシステムとして分散 key-value ストアを実現し，アプリケーションから直接的に key-value ペアにアクセスするインタフェースを提供する．

一方で，POSIX ライクなファイルシステムとしての機能は設けず，必要であれば上位レイヤーで実現する．

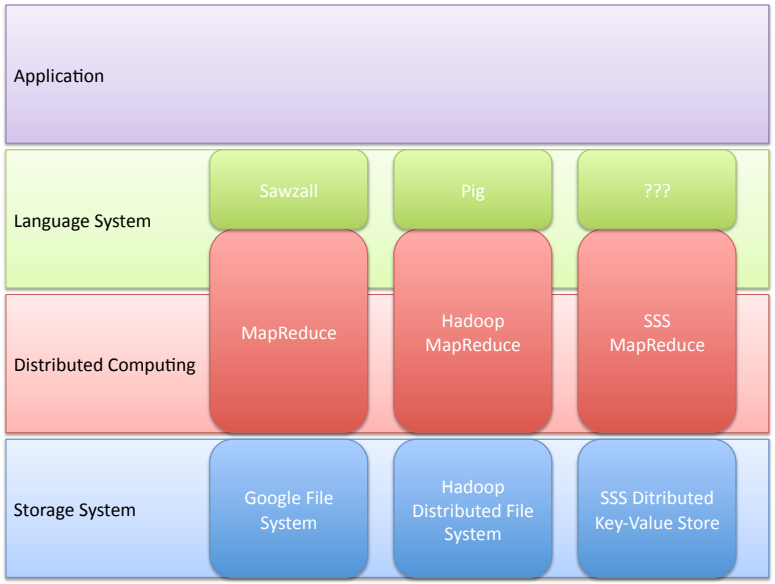

図 1 SSS DISC フレームワーク

3.1.3 データ・アフィニティの実現

Google や Hadoop の MapReduce の実装では，データのローカリティを考慮したタスク割り当てが行われる．具体的には，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) から極力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．しかしながら，高速なフラッシュメモリが利用できる条件下では，チャンクが存在するノードに確実にタスクが割り当てられなければ，性能低下は免れない．

SSS では，チャンクサーバノードへの Map/Reduce タスク割り当て状況・負荷やレプリカの参照状況に応じて，新たにレプリカを追加することで，データ・アフィニティを実現する．

3.2 設　　計

以下では，SSS で実現しようとしている (1) 分散 key-value ストアの全体アーキテクチャ，

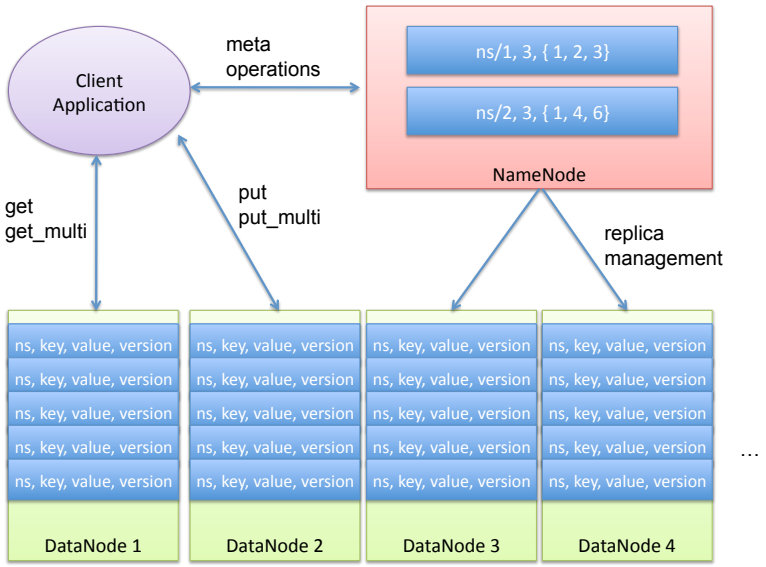

図 2 SSS 分散 key-value ストア

(2) 分散 key-value ストアの DataNode のストレージアクセス，(3) MapReduce，についてそれぞれ説明する．

3.2.1 SSS 分散 key-value ストア

SSS 分散 key-value ストアでは，(namespace, key, value, version) の 4 つ組をロード・ストアする分散 key-value ストアを実現する．実現する分散 key-value ストアの全体像を図 2 に示す．

namespace は key-value ペアのセットを指定するグローバルな識別子である．これは階層型ファイルシステムにおけるファイルの識別子 (ファイル名) と見なすこともできるが，我々の分散 key-value ストアではディレクトリ階層に応じたアクセス制御などは考慮せず，namespace ごとにオーナーやアクセス権を設定することで代替するものとする．また，version はレプリカの同期管理を行うために用意されているもので，GFS[2] における version と同様の役割を果たす．

SSS 分散 key-value ストアの構成要素は，HDFS と同様，単一の NameNode，多数の DataNode，そしてクライアントである．NameNode は namespace を管理するとともにクライアントからのアクセスを制御する．一方，DataNode はストレージノードごとに存在し，それぞれのストレージを管理する．SSS 分散 key-value ストアでは，GFS 等と同様に各 namespace は複数のブロックに分割され，各ブロックは DataNode に割り当てられる．

NameNode は，namespace に対するグローバルな操作を実現するとともに，namespace の DataNode への割り当てを決定する．namespace に対するグローバルな操作としては以下のものを用意する:

- create
- rename
- remove
- replicate

一方の DataNode は，担当 namespace ブロックに対するクライアントからのリクエストを適切に処理する責任を持つ．クライアントからのリクエストとしては以下のものを用意する:

- put
- put_multi
- get
- get_multi
- update
- remove

また，NameNode からのリクエスト (create, rename, remove, replicate) に応じて，自分が担当する namespace ブロックに対する操作を行う．

3.2.2 DataNode のストレージアクセス

図 3 に DataNode のアーキテクチャを示す．

各 DataNode は，クライアントや NameNode からのリクエストを処理する RPC サービスを提供するとともに，(namespace, key, value, version) の 4 つ組のセットを自身のストレージで管理する．クライアントから DataNode へのリクエストは，クライアント側のランタイムライブラリでカプセル化することで，直接的なストレージアクセスか RPC コールかを選択的に実行する．

内部的には，DataNode が管理するストレージは，(namespace, key, version) の 3 つ組をユニーク ID (もしくは extent の先頭アドレスとバイト数) にマップする Index Store と，value を格納する可変長の extent のセットで構成できる．

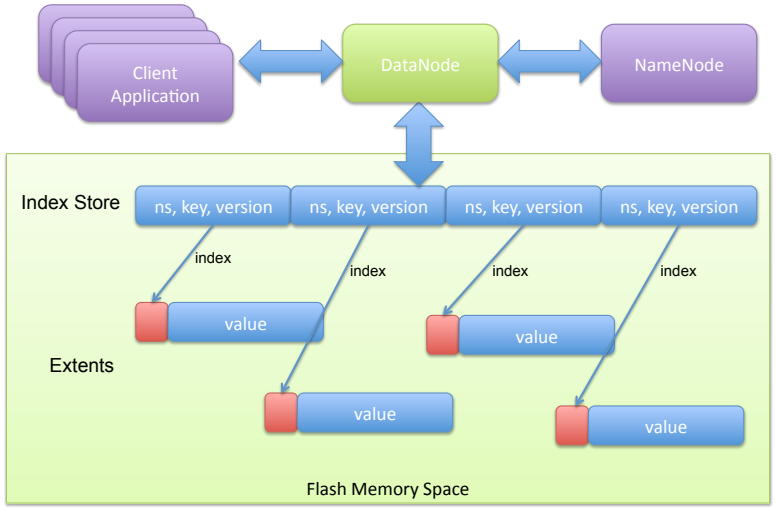

図 3 DataNode のストレージアクセス

実際の実装は本稿の時点では検討中である．(1) BerkeleyDB (BDB) の B-tree データベースにすべてを詰め込む，つまり extent を BLOB として格納する，(2) Index Store を BDB で実現して，extent 自体は buddy memory allocator[13] で割り当てる，などさまざま実現方法があり，少なくとも (1) は (2) に比べて実装は容易だがスペースの利用効率や速度の点では不利になり，アプリケーションからゼロコピーで extent にダイレクトアクセス可能にするためには (2) の方法を採るしかない．

また，フラッシュメモリデバイスへのアクセス方法についても多くのオプションがある．Fusion-io の ioDrive のような SSD としてのインタフェースしか持たないフラッシュメモリデバイスを利用することを前提にすると，(1) SSD 上のファイルシステム上の BDB データベースファイルに格納する，(2) SSD 上のファイルシステム上のファイルを mmap する，(3) SSD のブロックデバイスを直接メモリマッピングする，などの方法が考えられる．(1), (2) は namespace ブロックごとに別ファイルに格納することで Index Store を小さくでき，BDB やファイルシステムの提供する allocator を利用でき，レプリカの管理も容易になる．その反面，ファイルシステムによるオーバーヘッドがある．

一方で，ホストとフラッシュストレージ間のインタフェースは過渡的な段階にあり，将来

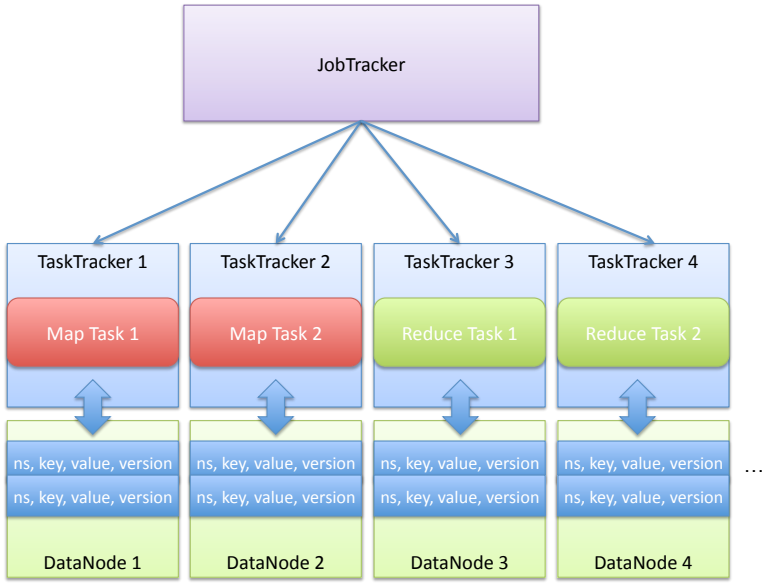

図 4 SSS MapReduce

的には NVMHCI (Non-Volatile Memory Host Controller Interface) のような汎用インタフェースが標準化されるなどして，Raw デバイスとしてアクセスできるフラッシュメモリデバイスが利用できるようになる可能性もある．その場合には，こうした key-value ストアに特化したウェアレベリングやトランザクションの実現を含めた challenging な研究・開発が可能になるだろう．

3.2.3 SSS MapReduce

SSS における MapReduce は，分散 key-value ストアのインタフェースを用いて実現されるという点を除いては，Hadoop MapReduce の実装を踏襲する (図 4)．

3.1.3 で述べたようにデータ・アフィニティを実現するために，分散 key-value ストアにおいてレプリカを必要に応じて追加する機能を用意するとともに，各 TaskTracker においてローカルの DataNode を利用して Map/Reduce タスクを実行できない場合にはレプリカ作成を NameNode に指示するオプションが必要になる．

4. 関連研究

Seltzer ら[14] は，Object-based Storage Device (OSD)[15] の利用を前提として，type/value ペアでファイルを特定してアクセスできるファイルシステム hFAD を試作している．我々の分散 key-value ストアは，DataNode の設計の点で彼らの試みに類似している．彼らはあくまでファイルシステムを拡張するアイディアとして hFAD を提案しているのに対して，我々はファイルシステムが通常提供するトランザクション機能を捨て，key-value ペアの単位でアトミックにストレージを操作する機能のみを提供することで，高速フラッシュメモリをより効率良く利用できるのではないかというアイディアに基づいている．

また，分散 key-value ストアについてはたくさんのシステムが存在する．代表的なものを挙げると，Amazon Dynamo[16]，MemcacheDB，Project Voldemort，Scalaris，他にも Dynomite，Ringo，Kai などがある．また，DISC システムのソフトウェアスタックとして言及した BigTable[6] や HBase[7] もそうである．これらのシステムはミドルウェアとして実現されているのに対し，我々はより低位のストレージ抽象として分散 key-value ストアを設計・実装しようとしている点で異なる．しかしながら，設計の詳細を決定する段階ではこれら既存システムのアーキテクチャを大いに参考にする予定である．

SSD を並列ファイルシステムに用いる試みとしては，Lustre で SATA 接続の SSD を用いた場合の NASTRAN アプリケーションベンチマーク結果が[17] に記されている．I/O ボトルネックが減少した結果，高速化の効果が得られているが，より高速な 10Gbit/sec クラスの SSD デバイスでどのような振る舞いをするかについては不明である．

5. まとめ

我々は，10Gbit/sec クラスの読み書き性能を持つ高速なフラッシュメモリに見合った，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「SSS」の実現を目指している．これにより，既存の DISC システムの持つソフトウェアスタックのオーバーヘッドの問題を解決するだけに留まらず，HPC を含めたより広範なアプリケーション分野への DISC システムの適用を可能にすることを目論んでいる．

本稿では，既存の DISC システムの問題点を考察した上で，我々がプロトタイプ設計している「SSS」の概要と設計について説明した．今後は，より詳細な設計を行い，プロトタイプ実装を行う予定である．

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) Apache Hadoop Project: Hadoop, `http://hadoop.apache.org/`.
5) Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks, *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, New York, NY, USA, ACM, pp.59–72 (2007).
6) Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data, *ACM Transactions on Computer Systems (TOCS)*, Vol.26, No.2, pp.1–26 (2008).
7) Apache Hadoop Project: Hadoop HBase, `http://hadoop.apache.org/hbase/`.
8) Borthakur, D.: HDFS Architecture, `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.
9) Schmuck, F. and Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters, *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, USENIX Association, p.19 (2002).
10) Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J. and Zhou, B.: Scalable performance of the Panasas parallel file system, *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, USENIX Association, pp.1–17 (2008).
11) PVFS2: Parallel Virtual File System, Version 2, `http://www.pvfs.org/`.
12) Cochrane, S., Kutzer, K. and McIntosh, L.: Solving the HPC I/O Bottleneck: Sun Lustre Storage System, `http://wikis.sun.com/display/BluePrints/Solving+the+HPC+IO+Bottleneck+-+Sun+Lustre+Storage+System` (2009).
13) Knuth, D.: *The Art of Computer Programming Volume 1: Fundamental Algorithms (Second Edition)*, pp.435–455, Addison-Wesley (1997).
14) Seltzer, M. and Murphy, N.: Hierarchical File Systems Are Dead, *HotOS XII: 12th Workshop on Hot Topics in Operating Systems* (2009).
15) Nagle, D., Factor, M.E., Iren, S., Naor, D., Riedel, E., Rodeh, O. and Satran, J.: The ANSI T10 Object-based Storage Standard and Current Implementations, *IBM Journal of Research and Development*, Vol.52, No.4, pp.401–411 (2008).

Vol.2009-HPC-121 No.42
2009/8/6

16) DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's Highly Available Key-Value Store, *SOSP '07: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, New York, NY, USA, ACM, pp.205–220 (2007).
17) McIntosh, L. and Burke, M.: Solid State Drives in HPC - Reducing the IO Bottleneck, `http://wikis.sun.com/display/BluePrints/Solid+State+Drives+in+HPC+-+Reducing+the+IO+Bottleneck` (2009).