Vol.2012-HPC-135 No.6
2012/8/1

MapReduce処理系の「京」での実装

松田 元彦[1]　丸山 直也[1]

概要：K 上で MapReduce を使って HPC 計算を行うための高性能な処理系 KMR の実現について述べる．高並列環境では，ノード数が多いので通信には低オーバーヘッドが要求される．また，多数ノードがファイルシステムを共有するのでファイル I/O を効率的に行うことも簡単ではなくなる．そのため，高並列環境に適応した MapReduce として KMR を実装している．K では MPI 拡張としてリモートメモリアクセス（RDMA）が提供されているので，その活用も重要である．まず，基本情報として K のファイル I/O と RDMA の通信オーバーヘッド特性を報告し，それに続いて KMR の実装を紹介する．KMR では，MapReduce の shuffle 操作に必要になる全対全通信に scatter-gather を組合わせた $\log(N)$ ステップのアルゴリズムを利用する．また，ファイルの読込みには断片の読込みと allgather による全体の再構成を用いる．

Implementing MapReduce on K-Computer

Motohiko Matsuda[1]　Naoya Maruyama[1]

Abstract: KMR is an implementation of a popular MapReduce framework on K-computer, which adapts to a highly parallel environment. There, low overhead communications and efficient file-I/O are naturally required. It is because there are many compute-nodes, where communications are repeated to each of them many times, and also they share a single file system and simply reading files by each node reveals a bottleneck. Thus, a new MapReduce implementation is needed particularly designed for a highly parallel environment. For the basic information, we report file-I/O performance and overheads of RDMA (remote direct memory access) operations, which motivate our design decisions. Following that, some of the KMR implementation is shown. It uses a $\log(N)$-step algorithm for all-to-all communication needed in the shuffle-stage of MapReduce. It is based on the combination of scatter-gather, where the $\log(N)$-step algorithm was not ever necessary but is now necessary for a class of the scale of K-computer. Its file operations are based on reading in chucks and aggregating them using allgather communication.

1. はじめに

単純サーチのような Embarassingly Parallel (EP) なアプリケーションであればスケーラビリティを得るのは難しくない．しかし高並列環境では，EP なアプリケーションを動かすことも自明でなく，ファイル I/O 等は面倒になりつつある．例えば，高並列環境ではノードにローカル・ストレージを持たない．そのため，一斉にデータベース・ファイルを読込む場合などは共有ファイルシステムにアクセスが集中するため性能が低下する．そこで，EP なアプリケーションに対しても，高並列環境で動かすための枠組みを考えておく必要がある．

データインテンシブ・アプリケーションに対しては MapReduce [1], [2] が広く認知されている．MapReduce はモデルと API が単純なので，EP なアプリケーションを効果的に記述したり，科学技術計算にまつわる前／後データ処理などを簡便に記述できると期待される．しかし，「Google's MapReduce」の流れを汲む MapReduce ではノードにローカル・ストレージを仮定しているので，高並列環境では違ったアプローチが必要になる．

ターゲット計算機である K [3], [4], [5] での実行環境は基本的に C，C++，Fortran 言語と MPI しかない．そこでファイル I/O を記述する自然な方法は MPI-IO であるが，そのためには MPI-IO がプラットフォームに最適化されて

[1] 理研 計算科学研究機構
AICS, Riken

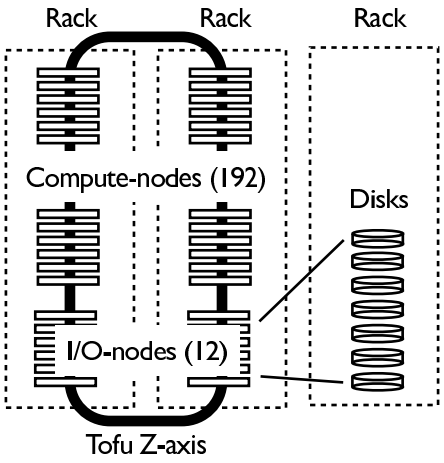

図 1 K のファイル I/O 構成

いる必要がある．しかし，K で使っている MPI-IO 実装である ROMIO は一般にプラットフォームに最適化されているとは言えない．また，MPI-IO をプラットフォームへ最適化することは簡単でもない上，ベンダとの協力が必須であり手軽には行えない．

ファイル I/O やそれにともなう集団通信を使ったデータ集約を行う場合，非同期的な処理が必要になる．一般に MPI は，メッセージの不要なコピーを行わない保証や非同期的な処理の記述が不得意である．一方，K では MPI 拡張としてリモートメモリアクセス（RDMA）が提供されている．そこで，オンコア（メモリ上）で動作する MapReduce を RDMA を使って提供し，その上でストリーム処理を行う MapReduce を改めて提供するようにする．K では，通常のストレージに相当する量のメモリも利用できるので，オンコアの処理も十分役立つはずである．オンコア処理とストリーム処理の適した方を利用者が選べるようになる．

既に多くの MapReduce の実装があるが，オンコア処理を提供するというように他とは狙いが異なるので，新たな MapReduce 処理系 KMR の実装を行うことにした．以下で，その報告を行う．節 2 で K の構成とそこから導かれる設計方針を，節 3 で関連研究について述べる．節 4 で実装の重要点について，節 5 で予定しているアプリケーションについて述べる．節 6 では今後の予定を述べる．

2. K の構成と設計方針

2.1 K の構成

計算ノード数が多くなると一つのファイルシステムを全ノードで共有するのは現実的でない．K では Lustre を拡張した FEFS [3] を使っているが，全ノードからアクセスできるグローバルファイルシステムと，部分ノードからアクセスできるローカルファイルシステムがある．K の各ノードはディスクを持っておらず，グローバルファイルシステムかローカルファイルシステムにアクセスする．

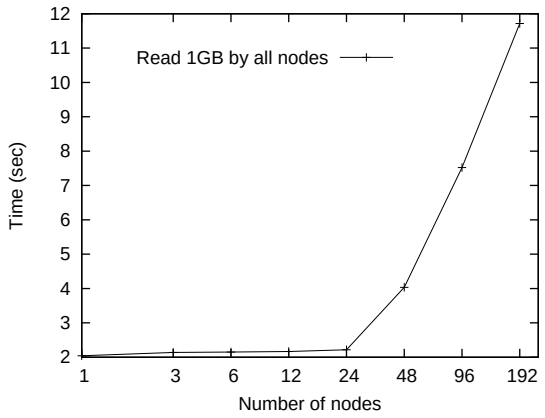

図 2 ファイルの一斉読込み時間

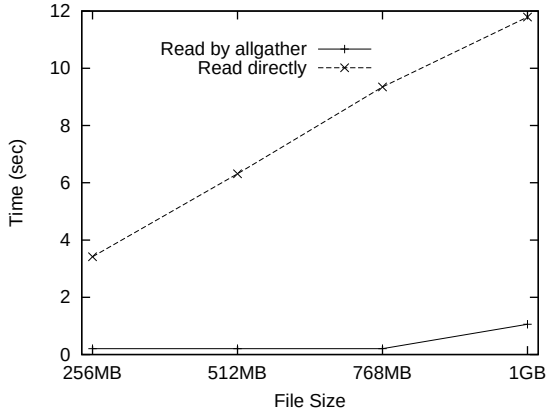

図 3 allgather を使うファイル読込み時間

図 1 に K のファイル I/O 構成を示す．192 ノードが 2 ラックに収まり Tofu ネットワークの Z 軸に載っている．計算ノード 2 ラック毎に 0.5 ラックに，ローカルファイルシステムのディスクが収納される．この 2.5 ラックを単位として，単位内のノードはディスクと Tofu の Z 軸を共有する．また，192 の計算ノードに対して 12 の I/O ノードがある．I/O ノード 12 ノードのうち 3+3 ノード（二重冗長構成）がファイバチャネルでディスクに接続されている．二重の冗長構成は対称で，ディスクを共有する．I/O ノードも Tofu ネットワークに接続されているので，ローカルファイルシステムには高速でアクセスできる．

2.2 K のファイル読込み特性

図 2 にファイル読込み時間を計測したものを示す．サーチ問題のデータベース・ファイルの読込みを想定して，1GB のファイルを読む時間を計測している．各ノードが一斉に 1GB のファイル・リードを行っている．ファイル I/O 構成に示したように，K では 2 ラック 192 ノードを単位として I/O が構成されている．そこで 2 ラック分について，ノード数を変化させた時の読込み時間を計測した．

2 ラックで I/O ノードと Tofu の Z 軸を共有しているため，ノード数が増えるに従い時間がかかるようになるのが観測できる．トータル 192GB の読込みに約 12 秒かかって

いるので，バンド幅は約 16GB/s ぐらい出ている．

このディスク I/O 構成で単純に各ノードが一斉に読込むのは，当然効率が悪い．そこで，各ノードが一部分を読込んだ後 allgather を使って全体を再構成する方法が考えられる．

図 3 に allgather を使ったファイル読込み時間を計測したものを示す．この計測では，読込むデータサイズを 256MB から 1GB まで変化させている．allgather を使うと圧倒的に性能が良いのが分かる．また，ほとんどの時間がオーバーヘッドによるものか，allgather を使う場合は 1GB 以外ではほぼ一定の時間になっている．

この計測では，実際のディスクに対する I/O の時間を除外するようにして計測した．実際のディスク I/O はパラメータやトラフィックの影響を受けるので，現時点での計測にあまり意味がない．また，平常時もファイル・ステージングがバックグラウンドで動作している．ファイル・ステージングは，ジョブ起動の前段としてグローバルファイルシステムからローカルファイルシステムにファイルをコピーするものである．これにより K のファイル I/O 構成に合ったローカルファイルシステムからファイル参照が行えるようになる．しかしこのため，制御できないバックグラウンドのトラフィックが発生する．また，現在 K は試験利用期間中で調整中であり，ディスク I/O の性能等は大きく変わる可能性が高い．そこで，性能測定は I/O ノードにバッファリングされた状態（ホット状態）を仮定することにした．性能測定に先立って，計算ノードの 1 つからファイルを読込んでバッファをホットにしている．

ファイルは分割して読込んでいるが，そのチャンクサイズは Lustre ファイルシステムのストライプ・サイズに合せて 1MB にしている．計測は 5 回行いその最小値でグラフをプロットしている（変動は小さい）．

この実験での実装は，ファイルを読込んだ後，MPI の allgather を呼んでいる．これは同期動作になっている．非同期処理にすることで，まださらに性能が改善する余地が十分残っている．

2.3 K の通信オーバーヘッド特性

MapReduce ではファイル I/O を行ったりと非同期的な処理が主であり，非同期通信を多用したいと考えている．そのため，性能の観点からの興味は通信オーバーヘッドにある．しかし，通信オーバーヘッドに関しては公表された計測データが見つからないので計測を行った．特に K では，MPI 拡張としてリモートメモリアクセス（RDMA）が提供されている．RDMA と iSend の通信オーバーヘッドの比較を行った．ここで通信オーバーヘッドは，「logP モデル」でいうところのオーバーヘッドの意味で使っている．つまり，通信に関わる CPU の消費時間である．

図 4 と**図 5** に小さいサイズのメッセージ通信時の通信オーバーヘッドを示す．オーバーヘッドの計測なので，データ転送に時間のかかる大きいサイズのデータでは計測していない．RDMA も iSend も通信発行後にリクエストに対して完了操作を行う必要がある．（RDMA の場合はリクエストではなく完了キューと呼ばれる）．ここではオーバーヘッドに完了操作の時間も含めて計測している．

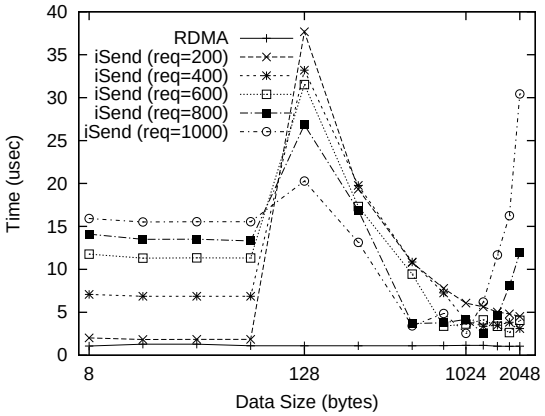

図 4 通信オーバーヘッド比較（2KB まで）

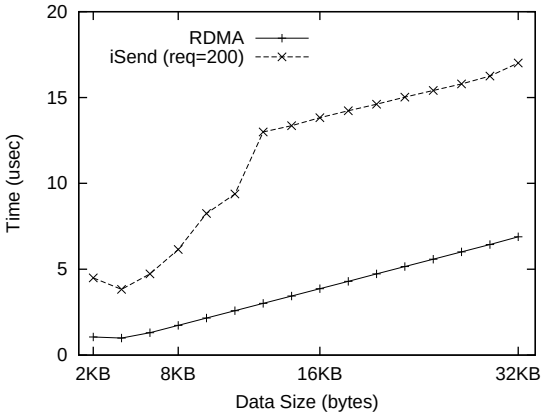

図 5 通信オーバーヘッド比較（2KB から 32KB）

オーバーヘッドの計測では，リクエストを一定回数発行してからその半数のリクエストの完了操作を行うことを繰り返している．グラフで「req=200」とある場合，まずリクエストを 400 回発行し，続いて 200 回完了操作を行う．その後は，200 回発行操作，200 回完了操作を繰り返す．つまり，「req=200」の場合，リクエストがペンディングしている個数は 200 から 400 の間である．RDMA の場合，リクエストがペンディングしている個数はほとんどオーバーヘッドに影響しなかった．よってグラフには，RDMA については「req=200」の場合のみを示している．

グラフから，RDMA は iSend に比べてオーバーヘッドが少ないことが分かる．また，オーバーヘッドの時間も一貫している．メッセージサイズ 2KB までの場合で 1.0–1.3usec の範囲にある．一方，iSend ではペンディングにするリクエスト個数によってオーバーヘッドが大きく変わっていることが観測される．

Tofu ネットワークのペイロードサイズは 2KB 弱である．

そこで，図 4 にはメッセージサイズ 2KB まで，図 5 にはメッセージサイズ 2KB 以上の計測を行った．計測はメッセージサイズ 2KB までは 1000,000 回行った平均値，2KB 以上では 100,000 回行った平均値である．計測は 5 回行いその最小値でグラフをプロットしている（変動は小さい）．

現在 K は試験利用期間中で調整中であり，オーバーヘッド計測についても MPI の性能等が今後大きく変わる可能性があることを注記する．

2.4 MapReduce の設計方針

K では今のところ MPI アプリケーションのみが許されており，Java 言語処理系も計算環境としては提供されていない．対象も HPC アプリケーションであるので，実装は MPI で行うことになる．

方針の一つは，汎用の MapReduce を提供しそれを使ってファイル I/O 処理を記述することである．元論文の Google's MapReduce をはじめ多くの MapReduce 実装ではファイルから読込みストリーム処理する部分がプリミティブとなっている．しかし，K ではクラスタ環境と異なりノード毎のローカルディスクを仮定できない．ローカルディスクも共有されており，その効率的な利用が性能の要となる．K の構成と特性のところで示したように，K ではファイル I/O 自体を並列に処理する必要がある．

どういうプログラム環境でファイル I/O を記述するかが問題であるが，MapReduce がその記述に適していると考えている．ファイル I/O の処理は，個々のノードでファイルを読込むことと allgather 等の集団通信を使って集約することである．これらは，個別処理，データ交換，集約処理をバルク同期的に続けて行うので，MapReduce そのものである．ファイル I/O に最適化した MapReduce は，map-reduce の組合わせで提供する．つまり，MapReduce の実装自体も MapReduce のアプリケーションである．逆に言うと，MPI のプログラムは基本的に send-receive で記述されるので，非同期的な動作の記述にあまり適していると言えない．

方針のもう一つは，MPI による実装と同時に RDMA による実装も提供することである．K の構成と特性のところで示したように，RDMA は一貫して低オーバーヘッドである．高並列環境では非同期動作と低オーバーヘッドな実装が必要なので，RDMA の利用が自然である．共有メモリと同様，RDMA 書込みはレースコンディションがあるためデバッグ等が難しくなることも多い．しかし，MapReduce は意味的にバルク同期なので処理系実装のバグかどうかの判断が付け易い．RDMA を利用し易いプログラム環境といえる．

データセンターとは環境が異なるため，フォールトトレランスの考え方が異なる．K のアプリケーションは MPI で記述されたバッチ計算環境で動作する科学技術計算が基本であるので，Google's MapReduce や Hadoop のようなフォールトトレランスの実現は難しい．現状，基本的にはフォールトトレランスについては何も考慮していない．K には粒度の荒い耐故障を提供するのが向いているので，チェックポイントの提供を考えている．

3. 関連研究

MR-MPI

開発中の KMR は MR-MPI [6] を大いに参考にしている．MR-MPI はマスタ／スレーブ・モードというものを持っており，map 時にマスタ・プロセスからキーを取得する動作が可能である．これは多くのアプリケーションで有用だと考えられるので，KMR にも取り入れた．

MR-MPI も Google's MapReduce や Hadoop と同様に入力と結果をファイルへ入出力することを前提として作られている．これらは，MapReduce をプリミティブとして使う場合に不便なことも多い．利用方法が異なるため，KMR ではメモリ上にキーバリューを生成する MapReduce を重視して実装している．

その他，MR-MPI の API は map 操作がキーバリューのキー総数をリターンするようになってるが，その部分は考え方が異なる．map 操作がキー総数を返すためには同期する必要があるが，map 操作毎に同期する必然性はない．そのため，KMR では将来の最適化を考えて異なる API を採用することにした．

実行オプション等の設定について MR-MPI では個別の API を用意しているが，この部分は文字列をキーとしてオプションを設定するような API に変更した．オプションの拡張が容易であるためである．オプションの指定には MPI の MPI_Info を使っている．

MPI 上の MapReduce 実装には他にも [7] などがある．

SSS

Google's MapReduce やその流れを汲む Hadoop では，map とそれに続く reduce という強く関連付けられた一連の操作で完結する処理を考えている．それに対して，map と reduce を自由に繰返し組合わせる哲学に従う MapReduce も多い [8], [9], [10], [11], [12], [13]．その典型として，SSS [8] がある．これは，MapReduce をプリミティブとして扱う実装方針には自然であるので，SSS と同じ考えに基づくことにした．

SSS の map 処理は局所性がないように作ってある．しかし HPC 計算では局所性が重要であるので，その点では考え方が異なる．Ogawa らの指摘通り map と reduce を本質的に同じものと認識した上で，KMR では map は局所性のある操作だと仮定する点が異なる．例えば map-map と続く場合に，map 操作のカスケード／チェイニングというような局所性を利用する最適化ができるように設計を進めている．

Spark [9]，Twister [10]，MRAP [11] は繰返し処理のため，オンコア（メモリ中）に途中結果を保持するようになっている．Spark ではキャッシュとして保存すると同時に，フォールトトレランスのため，キャッシュはデータの再構成ができる情報も保持している．Phoenix [14] は，マルチプロセッサ実装に特化した MapReduce を提供している．

その他の言語

高レベルの記述を用いそれによって MapReduce 操作を最適化する研究もあり，その代表に DryadLINQ などがある [15].

MapReduce と並列 DBMS の比較について述べている論文もある [16], [17]．そこでは MapReduce は並列 DBMS に比べて劣っているとし，性能上の違いを生じる理由として以下の点を挙げている [17]．• レコードのパースの繰返し，• データ圧縮をしていない，• パイプライン処理をしていない，• ストレージ・ブロックサイズに合わせたスケジューリングをしていない，• 列方向のレコード・ストレージを選べない．これらは，並列 DBMS との比較として提示されているが一般的な事項である．性能を出すためにはこれらを考慮して実装を行う必要がある．MapReduce で記述できるような問題に関しては，キーによる局所性を PGAS 言語のアドレス空間のように見なす点も指摘されている．

また，元論文の [1] の当初から，(parallel-prefix) scan 操作との関連も指摘されている．MapReduce と DBMS 操作や scan 操作との関連では，CM-Lisp [18] がもっと直接的である．並列プリミティブは map と（shuffle を含んだ）reduce しかなく MapReduce のようであるが，CM-Lisp は汎用の高並列プログラミング言語になっている．

4. 実装

4.1 全対全通信

KMR の実装は基本的には他の MapReduce と違いはない．ただし，MapReduce の実装では shuffle 時の全対全通信（all-to-all）の実装が重要である．特に，ノード数が多い高並列環境までサポートしなくてはならない点, 低オーバーヘッドな実行が要求される点，から全対全通信の実装が重要である．低オーバーヘッドな実行は，ファイル I/O 遅延のないオンコア動作をする上で重要である．K の RDMA のオーバーヘッドは小さいが，それでも高並列環境では，単純にノード数に比例するようなアルゴリズムはショートメッセージに対して使えなくなる．Tofu NIC（ICC）やトポロジに最適化した集団通信を行う必要がある．

データ転送が主な集団通信で効率的なアルゴリズムは，scatter-gather の組合わせで表現されることが多い [19]．また, 通信の並列度が落ちないようにするため, scatter-gather が複数 root を持つように構成する．これまで，全対全通信では scatter-gather の組合わせは必要にならなかった．しかし，現実に高並列環境が実現されノード数が多くなったことで，$\log(N)$ ステップのアルゴリズムが必要になってきた．

$\log(N)$ ステップの全対全通信は gather の後 scatter を行うことで実現できる．gather で各ノードのデータを集約した後，逆に scatter で各ノードへ分散する．gather のステップをいつの時点で終了するかで root の数を変えることができるので，それによって並列度を調整する．

一般に，いろいろな並列言語の処理系についても，高並列になるとポイント・ツー・ポイント通信を生成しているだけでは性能がスケールしなくなると考えられる．MapReduce を汎用計算の記述に用いる際に shuffle 処理は不利に見えるが，全対全集団通信を生成できる点では逆に有利かも知れない．

4.2 ファイル読込み

全ノードから一斉にファイルアクセスが集中するといった極端な状況はファイル・ステージングの運用によって避けられている．しかし，ローカルファイルシステムを共有するノード（192 ノード）からのアクセスであっても，一斉読込みでは性能が低下する．そのため，KMR では K の特性に合わせたファイル読込みを実装する．ファイル断片の読込みと allgather によるファイル再構成が基本的な考え方である．節 2 の K の特性を報告したところで既に示したように，allgather の利用によりファイルの一斉読込み性能は大きく改善する．

ただし，ファイル I/O についてはデザイン空間が広い．allgather による性能改善はファイル全体を一気に読込む場合である．ストリーム的な処理をする場合は，これとは違った読込み方法が必要になる．また，K ではファイル・ステージングがありノードとファイルの関係を把握する必要がある．さらに，Tofu ネットワークの Z-軸に載ったノード群がローカルファイルシステムを共有するため，トポロジ情報も考慮する必要がある．ファイル・ステージングの方法はユーザーが指定でき，ジョブの物理ノードへの割付けはジョブスケジューラの役割であるので，様々な組合わせが発生する．さらに実装面からは，単純なファイルの読込みに read を使うか mmap を使うかなどの選択肢がある．結局，アプリケーションの必要に応じて，順にバリエーションを実装して行くしかない．

また，ファイル・ステージングの使用によって，ファイルを分断して作成する必要にせまられる場合がある．ファイル分断の対策として「File Composition Library」[20] などが提案されている．そういった I/O 関連のライブラリとの連携もできるようにしていく予定である．

5. 予定するアプリケーション

ゲノム配列解析ソフト GHOST [21] は K 上の MapRe-

duce 開発の動機付けとなったアプリケーションの一つである．GHOST の MPI 分散並列版では，データベース・スキャンをサーチ対象のゲノムシーケンスを変えることで並列に行う．GHOST の開発者らは，このデータベース・スキャンを簡便に行うためにマスタ／スレーブ動作を行うライブラリを MPI で作成している．我々は，このライブラリを記述できるようにマスタ／スレーブ・モードを持つ MapReduce を開発している．さらに，GHOST では各ノードで大きなデータベース・ファイルを読込むので，ファイル I/O を最適化してやる必要がある．その部分を MapReduce に取り込むように開発を行っている．

また別のアプリケーション候補として，Parallel netCDF [22] ライブラリの実装を考えている．元々の Parallel netCDF は MPI-IO を下層に用いることで，プラットフォームへの最適化を図っている．しかし，MPI-IO 実装である ROMIO は K のファイル I/O 環境に最適化されていない．そこで，KMR のアプリケーションの一つとして Parallel netCDF を実装することを考えている．

6. 今後の予定

K では I/O ノードも計算ノードと同じ CPU と Tofu NIC（ICC）を備えている．現在は，I/O ノードでのユーザー・プログラムの動作は許されていない．もし，I/O ノードでのユーザー・プログラムの動作が許されれば，RDMA を使って計算ノードへのファイル読込みを非同期的に処理できる．RDMA を使うと，OS で提供される非同期リード（aio_read）とは格段に性能差がある読込みができるはずである．

K の Tofu NIC はハードウェアの仕様が非公開であり，アトミックな RDMA 操作が実装されているか分からない．もし，アトミックな RDMA 操作が実装されているならば非同期処理にはうってつけなので，KMR 処理系でも使用する予定である．

C++11 言語の関数閉包（lambda expression）は MapReduce を使った記述に相性が良さそうである．まだ各社コンパイラの実装が十分でないので現状は使えないが，将来使えるように KMR の実装を行っている．

謝辞

本論文の結果の一部は，理化学研究所が実施している京速コンピュータ「京」の試験利用によるものです．

ゲノム配列解析ソフト GHOST のソースコードを提供して下さるとともに研究の動機付けを与えて下さった，東京工業大学の秋山泰 教授と角田将典 博士に感謝します．

参考文献

[1] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, Symp. on Opearting Systems Design & Implementation (OSDI), pp.137–150 (2004)

[2] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, CACM Vol. 51 Issue 1, pp.107–113 (2008).

[3] 富士通: 雑誌 FUJITSU, 特集：スーパーコンピュータ「京」, 2012-5 月号 VOL.63, NO.3 (2012).

[4] Maruyama, T., Yoshida, T., Kan, R., Yamazaki, I., Yamamura. S., Takahashi, N., Hondou, M., and Okano, H.: SPARC64 VIIIfx, IEEE Micro, Vol.30, Issue 2, pp.30–40 (2010).

[5] Ajima, Y., Inoue, T., Hiramoto, S., Shimizu, T., and Takagi, Y.: The Tofu Interconnect, IEEE Micro, Vol.32, Issue 1, pp.21–31 (2012).

[6] Plimpton, S. J. and Devine, K. D.: MapReduce in MPI for Large-Scale Graph Algorithms, Parallel Computing Vol.37, Issue 9, pp.610–632 (2011).

[7] Hoefler, T., Lumsdaine, A., and Dongarra, J.: Towards Efficient MapReduce Using MPI, European PVM/MPI Users' Group Meeting, pp.240–249 (2009).

[8] Ogawa, H., Nakada, H., Takano, R., and Kudoh, T.: SSS: An Implementation of Key-Value Store Based MapReduce Framework, Intl. Conf. Cloud Computing Technology and Science (CloudCom), pp.754–761 (2010).

[9] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I.: Spark: Cluster Computing with Working Sets, Conf. on Hot Topics in Cloud Computing (HotCloud) (2010).

[10] Ekanayake, J., Li. H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G.: Twister: a Runtime for Iterative MapReduce, Intl. Symp. on High Performance Distributed Computing (HPDC), pp.810–818 (2010).

[11] Sehrish, S., Mackey, G., Wang, J., and Bent, J.: MRAP: a Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns, Intl. Symp. on High Performance Distributed Computing (HPDC), pp.107–118 (2010).

[12] Elnikety, E., Elsayed, T., and Ramadan, H. E.: iHadoop: Asynchronous Iterations for MapReduce, Intl. Conf. on Cloud Computing Technology and Science (CloudCom), pp.81–90 (2011).

[13] Rehmann, K.-T. and Schoettner, M.: Applications and Evaluation of In-memory MapReduce, Intl. Conf. on Cloud Computing Technology and Science (CloudCom), pp.67–74 (2011).

[14] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C.: Evaluating MapReduce for Multicore and Multiprocessor Systems, Intl. Symp. on High Performance Computer Architecture (HPCA), pp.13–24 (2007).

[15] Isard, M. and Yu, Y.: Distributed Data-parallel Computing using a High-level Programming Language, Intl. Conf. on Management of Data (SIGMOD), pp.987–994 (2009).

[16] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S. R., and Stonebraker, M.: A Comparison of Approaches to Large-scale Data Analysis, Intl. Conf. on Management of Data, pp.165–178 (2009).

[17] Stonebraker, M., Abadi, D. J., DeWitt, D. J., Madden, S. R., Paulson, E., Pavlo, A., and Rasin, A.: MapReduce and Parallel DBMSs: Friends or Foes?, CACM, Vol.53, No.1, pp.64–71 (2010).

[18] Steele, G. L. Jr. and Hillis, W. D.: Connection Machine Lisp: Fine-grained Parallel Symbolic Processing, Conf. on LISP and Functional Programming (LFP), pp.279–

297 (1986).
[19] Matsuda, M., Kudoh, T., Kodama, Y., Takano, R., and Ishikawa Y.: Efficient MPI Collective Operations for Clusters in Long-and-Fast Networks, Conf. on Cluster Computing (Cluster), pp.1–9 (2006).
[20] 大野善之, 堀敦史, 石川裕: 並列ジョブのファイル I/O をひとつのファイルに集約する方式の提案と予備評価, 情報処理学会研究会報 HPC-2011-132 (2011).
[21] 東京工業大学 秋山研究室: ゲノム配列解析, (online) 〈http://www.bi.cs.titech.ac.jp/web〉.
[22] Li, J., Liao, W.-K., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M.: Parallel netCDF: A High-Performance Scientific I/O Interface, Conf. on Supercomputing (SC), pp.39–49 (2003).