Vol.2014-HPC-146 No.16
2014/10/3

並列ファイルシステムへのアクセス局所性を考慮したMapReduce負荷分散実現に向けて

滝澤 真一朗[1,a)]　松田 元彦[1,b)]　丸山 直也[1,c)]

概要：計算科学アプリケーションには，そのワークフローを MapReduce モデルで容易に記述できるものも多く，MapReduce を採用することにより，実装の容易化，並列実行の自動化等の恩恵を受けられる．一方，計算科学アプリケーションは大規模並列システムで実行されるため，そのワークフローを実行する MapReduce 処理系にも高いスケーラビリティや，並列ファイルシステムに対応した高速 IO の実現が求められる．本研究では MapReduce 実行中の並列ファイルシステムへのアクセスの局所性を高めつつ，スケーラブルに動的負荷分散を行う処理系の実現を目指す．本稿では，並列ファイルシステム上のファイルの位置に基づく，該当ファイルを入力とするタスクを静的に割り振る手法を提案し，大規模並列システム向け MapReduce 処理系 K MapReduce に，京コンピュータを対象システムとして実装した．ファイル読み込み性能の評価を行った結果，N ファイルを N ノードが読み込む評価において，ランダムにファイルをノードに対応させた場合に対して，本提案は平均して 9%の性能向上を達成した．また，1 ファイルを N ノードが読み込む評価においては，本提案は平均して 4.5 倍の性能向上となった．

1. はじめに

テキストマイニングやログ解析等の大規模データを並列に処理するプログラミングモデルとして MapReduce [1] が用いられている．MapReduce による並列プログラムは，計算対象を Map タスクと Reduce タスクの 2 つの逐次処理として分割実装するだけの容易なものであり，処理の並列化は MapReduce 処理系が自動的に行う．このプログラム記述の容易さを理由にデータ処理を中心に MapReduce は採用されているが，近年ではゲノム解析や分子動力学の計算科学アプリケーションにおいても，MapReduce の粗粒度なタスク実行形態がワークフロー実行要求に適しているということで，使われ始めている [2], [3], [4].

粗粒度タスクの連携実行を行う計算科学アプリケーションについて，将来のエクサスケールシステムにて重要となるアプリケーション分野についてまとめられている「計算科学ロードマップ」 [5] を参照し，調査を行った．その結果，防災連携シミュレーションやマルチスケール・マルチフィジックスシミュレーションの連成計算のように複雑なワークフロー実行を要求するものもあるが，多くはパラメータスタディや，アンサンブル計算であり，これらのワークフローは MapReduce モデルで容易に表現可能である．実際，パラメータスタディは Reduce なし（あったとしても結果集計等の軽量タスク）の Map のみの Embarrasingly Parallel な MapReduce として，アンサンブル計算は繰り返し処理を行う Iterative MapReduce として表現できる．

しかしながら，京コンピュータのような大規模並列システムで計算科学アプリケーションのワークフローを MapReduce プログラミングモデルにて効率よく実行するには，IO の効率化と負荷分散を同時に実現する必要がある．MapReduce 処理において，タスクの実行時間のばらつきの原因となる箇所は (1)Map タスク入力データの局所性，(2)Map/Reduce タスク実行時間のばらつき，(3)Reduce 入力データ量のばらつき，と 3 カ所ある．本研究では (3) については，計算科学アプリケーションでは計算パターンより適用効果が少ないと考え，(1)，(2) のみについて課題としてとらえる．(1)，(2) については，Hadoop [6] の様な既存システム・研究において，データの局所性を考慮したタスク配置や投機的実行により部分的に解決されている．ただ，データを専用ファイルシステムに保存したり，中央集中型のスケジューラを採用している問題がある．大規模並列システムでは並列ファイルシステムが用いられているため，MapReduce 計算に専用ファイルシステムを用いる場合には無駄なコピーが発生する．ゲノムデータ等，計算科学アプリケーションの入力データにはペタバイト級の大

[1] 理化学研究所 計算科学研究機構, RIKEN AICS
[a)] shinichiro.takizawa@riken.jp
[b)] m-matsuda@riken.jp
[c)] nmaruyama@riken.jp

容量のものもあり，専用ファイルシステムへのコピーはストレージ容量とコピー時間が無駄になるため，並列ファイルシステムへの直接的なアクセスが求められる．またスケジューラが中央集中型の場合，大規模並列にはスケールしない．

本研究では，MapReduce を大規模並列システム上で効率よく実行するために，並列ファイルシステム上のデータの局所性を考慮した負荷分散の実現を目指す．本稿では，その実現を目的に並列ファイルシステム上の入力ファイルのストライプ情報を取得し，入力ファイルに近い計算ノードに該当ファイルを処理するタスクの割り当てを行う．本提案を我々が開発している大規模並列システム用 MapReduce 処理系である K MapReduce に実装し，ファイル読み込み性能の評価を行った．その結果，N ファイルを N ノードが読み込む評価において，ランダムにファイルをノードに対応させた場合に対して，本提案は平均して 9%の性能向上を達成した．また，1 ファイルを N ノードが読み込む評価においては，本提案は平均して 4.5 倍の性能向上となった．さらに，データの局所性を考慮した動的負荷分散実現に向けてコストモデルを構築した．

2. 関連研究

MapReduce 処理の負荷分散，特にデータアクセス性能の改善についての関連研究を述べる．

MapReduce の処理系として広く利用されている Hadoop [6] では，分散ファイルシステム HDFS に格納されたデータに対して，データの局所性を考慮したタスク割当を行う．MapReduce 実行のための専用のファイルシステムを用いている点が本研究と異なる．また Hadoop では比較的サイズの大きい（デフォルトでは 64MB）チャンク単位をタスクの入力とするのに対して，本研究で対象とする並列ファイルシステムでは比較的小さいサイズ（京では 1MB）でファイルを分割し分散格納しており，タスクの入力とする際には複数のファイルサーバからの読み込みが必要となる．また，Hadoop では処理時間に遅延の生じているタスクは他のノードでも平行して実行する投機的実行による負荷分散を行うが，中央集中型のスケジューラを用いているため，大規模並列システムではスケールしない．同様な手法は共有メモリシステムを対象とした MapReduce 処理系である Phoenix [7] でも採用されているが，同様な問題を抱えている．

MapReduce 処理の IO 最適化についての研究が行われている．Twister [8], [9] では PageRank 等の MapReduce 処理を繰り返し行う必要のある計算において，静的な入力データの読み込みを減らし，IO 量，読み込みオーバーヘッドの削減を行っている．一方で，タスクは静的に計算ノードに割り振られるだけであり，タスク毎に計算量が異なる場合には負荷分散は行われない．CooMR [10] では Hadoop の MapReduce 処理の内，Map 出力から Shuffle 完了までの IO 最適化を行っている．具体的には Map 出力の中間 Key-Value Pair をバッファリングした後に Log-Structured ファイルシステムに書き込むことで，IO 要求の削減，IO 競合の削減を行っている．また，Shuffle 時のソートでは Key のみをディスクから読み込むことで，IO 量を削減している．一方でタスクのスケジューリングは Hadoop のものそのままであり，スケーラビリティに課題がある．Hadoop の Delay Scheduling [11] では複数利用者複数ジョブの環境化において，データアクセスの局所性を高めるために，タスク割当を一定時間遅延させる方法を採用している．大規模並列システムでは別途ジョブスケジューラにて全体のジョブ実行管理がされているため，このような手法は不要であるが，本手法同様に，タスク実行を遅延させることによって並列ファイルシステムへのアクセス競合を軽減できる可能性はある．

負荷分散のため，ノード間でタスクの Work Stealing を行う MapReduce 処理系がある [12], [13], [14], [15]．これらは Work Steal を行うノード (Thief) は他の全ノードに Steal を試みたり，残りタスクの多いノードから Steal を試みるものである．数千から数万ノード規模の大規模並列システムでは，全ノードにタスクを問い合わせるのはスケーラビリティに乏しく現実的ではなく，Thief ノード近傍のノード等，数に制限を設ける必要がある．また，データの局所性を高め IO 性能を向上するために，Work Steal する際には，Theif と Steal 対象タスクの入力データの位置関係を考慮する必要がある．

MapReduce において，Map フェーズでは個々のタスクの入力データ量はおよそ等しくなるが，Reduce フェーズでは Key に対する Value の量にばらつきが生じ，入力データ量がタスク毎に異なりうる．これは Reduce タスクの負荷・実行時間のばらつきとなり，ジョブ全体の実行時間の増加へとつながりうる．この問題の解決を目標とする研究として，入力を細かく分割して処理する SkewTune [16] や，Gufler らによる Key 毎の Value サイズのばらつきを見積り，各ノードが処理する Value サイズが均等になるように Shuffle を行う手法 [17], [18] が提案されている．有効な手法ではあるが，本研究で対象とする計算科学アプリケーションでは，Key 数が少ないものや，Reduce の計算量が小さいものが中心であるので，効果は少ないと考えられる．

3. MapReduce での負荷不均衡発生場所

MapReduce は対象とする問題を Map と Reduce の 2 種類の計算に分け，Map タスクの出力である Key-Value Pair（以降 KV）を同じ Key でまとめた結果を Reduce タスクの入力として処理する計算モデルである．プログラム開発者は Map と Reduce の計算内容をそれぞれ逐次処理で実装すればよく，処理系にて入力を分割し，自動的に各

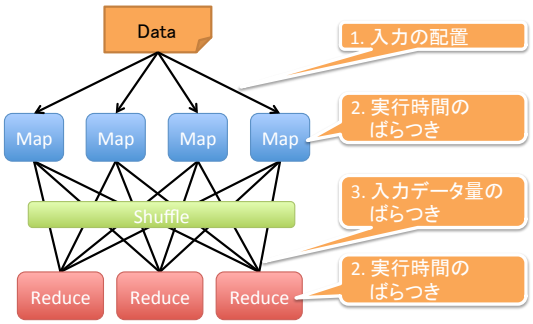

図 1 MapReduce における負荷不均衡発生場所

Map/Reduce タスクを並列に実行する．

MapReduce プログラミングモデルでは，この通り並列処理を透過的に実現しているため，実行性能は処理系の実装依存となる．MapReduce 処理はデータドリブンであるため，タスクの入力データへのアクセス性能が全体の実行性能に大きく影響を与える．また，全ての Map タスクの実行完了後に Reduce タスクは開始するので，タスクの実行時間のばらつきを最小とするため，負荷分散を行う必要がある．図 1 に MapReduce 処理における，タスク実行時の負荷不均衡が発生しうる場所とその原因を示し，以下にその説明をする．

1. **入力の配置** MapReduce において，Map タスクは分割された入力データを入力として，複数並列に実行される．そのため，各 Map タスクは入力データから近いノード，またはプロセッサにて実行できれば，データアクセスの競合が減り実行効率の向上が期待できる．このようにデータの局所性を向上するために，Hadoop では対象データを保存しているノードに該当データを処理するタスクを割り振り，Richard らの研究 [12] では NUMA アーキテクチャを考慮し，対象データを格納しているメモリから近いプロセッサにタスクを割り振る．しかしながら，データの配置によっては常にデータの近傍でタスクを実行することは困難であり，リモートアクセスが発生し，タスク実行時間の遅延へとつながる．
2. **実行時間のばらつき** 上記より，常に高い局所性を維持することは困難であり，同じ入力データ量，かつ入力に対して計算量が比例するタスクであったとしても，データへのアクセス距離による遅延の影響により，実行時間にばらつきが生じる．また，入力データ量と計算量が比例しない計算の場合では，実行時間の推定が難しく，ばらつき発生の予測が困難であり，負荷分散のためには Work Stealing [12], [13], [14], [15], [19] のような動的な手法が必要である．
3. **入力データ量のばらつき** Map タスクの入力は，利用者または処理系により自動的に，ほぼ均等に分割することが可能であるが，Reduce タスクの入力は計算ロジックにより決まるため，利用者によるプログラムの修正がない限り，タスク間での入力サイズの均等化は困難である．MapReduce 処理系にて行える負荷分散のアプローチとして，Reduce タスクの入力サイズを見積り，複数の Reduce タスクをグループ間で入力サイズが均等になるようにグルーピングし，グループ単位でノードに割り当てる方法が提案されている [16], [17], [18]．しかしながら，この手法が有効に働くには Map 出力にて多種類の Key を生成する必要があるため，アプリケーションロジックと実行規模に大きく依存するので，適用範囲が限られる．

4. 大規模並列システムでの計算科学アプリケーションのワークフロー実行

計算科学アプリケーションには様々な分野，様々な実行形態のものがあるが，将来のエクサスケールシステムにて重要となるアプリケーション分野についてまとめられている「計算科学ロードマップ」 [5] を参照したところ，パラメータスタディや，アンサンブル計算，モンテカルロシミュレーション等，粗粒度タスクの並列実行を要するものが多くあることがわかった．これら計算のワークフローは MapReduce モデルで容易に表現できるので，MapReduce を採用することにより，研究者はアプリケーションロジックの実装のみに集中でき，生産性の向上が期待できる．しかしながら，計算科学アプリケーションワークフローの特性や，アプリケーションを実行する並列システムの特徴のため，解決しなければならない課題がある．

4.1 ワークフローの特徴

アンサンブル計算やモンテカルロシミュレーションは条件の異なる複数のシミュレーションを実行し，その結果を集計して，次イテレーションの条件の計算や結果の集計を行う．MapReduce モデルで表現する場合，前段のシミュレーションを Map タスクとして，後段の結果集計を Reduce タスクとして表現できる．しかしながら，結果集計では，全 Map タスクの計算結果を 1 またはごく少数のノードに集める必要がある．つまり，Reduce タスクの入力 Key 数は少なく，Reduce タスク実行時の実行時間のばらつきは防ぐことができない．

また，社会シミュレーション等で採用されているマルチエージェントシミュレーションでは，数十のパラメータを変化させて実行する必要があり，実行条件に応じて実行時間が大きく変化する．

以上より，図 1 の「3. 入力データ量のばらつき」の課題解決を図っても効果は少ないと考え，本研究では「1. 入力の配置」と「2. 実行時間のばらつき」の課題解決を目的とする．

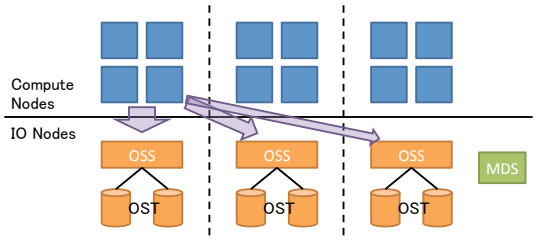

図 2 並列ファイルシステムの構成

4.2 データアクセスの特徴

計算科学アプリケーションを並列システム上で実行する場合，処理対象となるデータは全計算ノードからアクセスできる，並列ファイルシステムに置かれていている．実行時に計算ノードローカルなストレージにステージインして計算することで，局所性を高め，高速なデータアクセスが可能であるが，次に示す問題がある．

- ストレージ容量の制限．特に観測データの全対全の相関計算を行う場合には，複数のノードに重複してデータを格納することになり，無駄が生じる．
- 中間出力ファイルの処理．シミュレーションを数珠つなぎで連続して行う場合や，アンサンブル計算では，Map タスクの連続実行や MapReduce 処理全体を繰り返し実行する必要がある．このとき，タスクの出力ファイルをローカルストレージに書き出していては，次のタスクの実行時にネットワーク転送や共有ファイルシステムを介したコピーを必要とするため，オーバーヘッドとなる．

図 2 に想定する並列ファイルシステムの構成を示す．並列ファイルシステム上のファイルは一定サイズに分割され，複数のファイルサーバ（OSS）に分散して格納される．ファイルサーバの配下にはファイルの実体を格納するストレージデバイス（OST）がある．計算ノードが並列ファイルシステム上のファイルにアクセスする際には，まずファイルの位置情報をメタデータサーバ (MDS) に問い合わせ，取得した情報をもとに OSS に並列アクセスを行う．本研究では，計算ノードとファイルサーバ間に距離の遠近があり，距離に応じてファイルアクセス性能が異なるものと仮定する．詳細は後述するが，京コンピュータはこのような構成となっている．また，Fat Tree のようなフルバイセクションネットワークにおいても，ネットワーク構成やファイルサーバの配置により，遅延の大小が発生し，距離が定義できる．

Hadoop 等の MapReduce では，データローカリティを満たすようにタスク配置が行われる．具体的にはファイルを保持するノードにタスクを割り当てる，それが無理な場合はネットワーク的に近い場所にあるノードにタスクを割り当て，データアクセスコストの削減を行う．一方で，並列ファイルシステムの場合は問題が困難になる．ファイルは複数の OST/OSS に分割して保存されているため，これらに対して適した位置にある計算ノードに判定する必要がある．特に観測データの相関解析等，複数の入力ファイルを必要とする計算では，多数の OSS 群へのアクセスが求められるが，それらに対する最適なアクセス性能を達成する計算ノードにタスクを割り振る仕組みが求められる．

5. 並列ファイルシステムの局所性を考慮した MapReduce 実行

並列ファイルシステムへのアクセス局所性を考慮した MapReduce 処理の負荷分散実現に向けて，我々は
(1) 処理対象データの局所性を考慮したタスク割り当て
(2) 局所性を考慮した動的負荷分散
を行うことを考えている．並列ファイルシステム上の入力ファイル群の位置情報をもとに，それらから近い計算ノードを判定し，それらを入力とするタスクを割り振る．その結果，入力ファイルの配置パターンによってはタスク量にばらつきが発生するため，また，入力ファイルのサイズやアクセス性能に寄らずタスクの実行時間にばらつきが生じる場合を考慮して，データアクセス性能を考慮した動的負荷分散を行う．

本稿では上記 (1) についての結果を報告する．京コンピュータを対象とした実装の詳細は以降で述べるが，基本はタスクの入力ファイルのストライプ情報を取得し，ファイル断片が格納されている OST 群から近い計算ノードに，該当ファイルを処理するタスクを割り当てる．これにより，ファイルアクセス時の通信量・ネットワーク競合の削減が期待できる．

5.1 京のネットワーク，ファイルシステム

京コンピュータは 6 次元トーラスネットワークの Tofu にて計算ノード間通信，IO 通信が行われる．Tofu の物理座標は (X, Y, Z, A, B, C) で表され，サイズは 24×18×17×2×3×2 である．ジョブ実行時には隣接するノード集合が切り出されてアプリケーションに割り振られ，アプリケーションからは 3 次元トーラスとして見える．

京コンピュータのファイルシステムには Lustre ベースの FEFS が用いられている．Tofu の 6 次元座標 Z=0, C=1 は FEFS の OSS として動作する，IO 専用ノード (IO ノード) として働く．IO ノード配下には 2 つの OST がある．そのため，システム全体では IO ノードは 2,592 台，OST は 5,184 台存在する．しかしながら，アプリケーションからは常にこれら全てを利用できるわけではない．京コンピュータでは 192 ノード (正確には X×Y×Z×A×B×C = 1×1×16×2×3×2) 単位に 6 つの IO ノード（12 の OST）が存在し，アプリケーションが使用するノード数に応じて IO ノード数が変化し，IO 性能が変化する．例えば，3,072 ノードを 4×4×16×2×3×2 の形状で利用する場合，アプリ

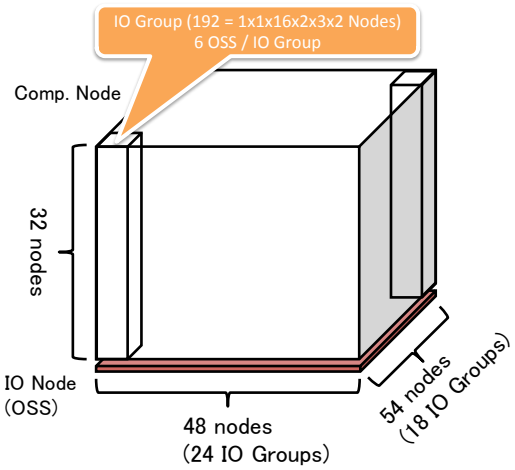

図 3 京ファイルシステム概要図

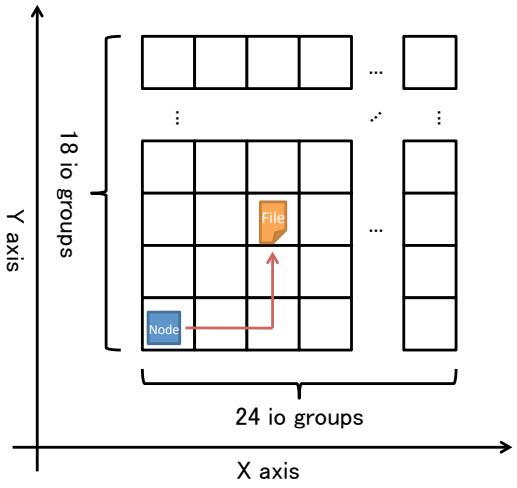

図 4 IO Group をまたぐファイルアクセス

ケーションが享受する IO 性能は 192 ノード利用時の 16 倍となる．ただ，標準ではストライプサイズ 1MB，ストライプカウント 12 に設定されており，1 ファイルは 12 を超えた OST に保存されることは無い．

図 3 に京のファイルシステムの概要図を示す．図中では 192 ノードの集合を IO Group と定義しており，IO Group は X 軸方向に 24，Y 軸方向に 18 ある．図 4 は IO Group をまたいだファイルアクセスを表しており，Z 軸を上から見た図である．同じ IO Group 内の OST にのみファイルが格納されている場合には，Z 軸直下の IO ノードにアクセスだけだが，そうでない場合には図に示す通り，IO ノード間でルーティングが行われる．この通り，IO Group をまたいだ IO を行う時には計算ノードと IO ノード間に距離があり，性能に影響する．

5.2 設計と実装

本研究では IO Group 内で閉じた IO を行う際には最も高い局所性を達成し，IO Group をまたぎ，距離が遠くなる IO ほど局所性が低くなると定義する．そのため，目的はファイルが属する IO Group にいる計算ノードに，該当ファイルを処理するタスクを割り振ることとする．ファイルが複数の IO Group にまたがる場合や，複数のファイルを入力とするタスクの場合は，ファイルが置かれている IO Group 群から平均的に近い IO Group 内計算ノードにタスクを割り振る．

上記を満たすタスク割当を実現するため，我々が開発している大規模並列システム向け MapReduce 処理系である K MapReduce (KMR) [20] に対して，`kmr_assign_file()` という関数を実装した．この関数は Shuffle 同様の集団通信を行うが，入力 KV の Value としてファイルパス（ヌル文字で区切ることにより複数のパスを与えることが可能）を受け取り，該当ファイルから近い計算ノードにそのファイルを割り振る処理を行う．京コンピュータでは計算ノードとファイルが格納されている IO ノードの 6 次元 Tofu 座標が取得できるので，これより IO Group の座標 (X,Y) を抽出し，計算ノードと IO ノードの IO Group のマッピングを行う．

具体的には以下の流れの処理を行う．

(1) 計算ノードの IO Group を取得し，KV<(X,Y),Node_ID> を生成後，Shuffle する．

(2) FEFS に問い合わせ，ファイルのストライプ情報を取得し，断片が保存されている IO ノード群を判定し，IO ノード群の IO Group を取得する．次の式に従い，ファイルの IO Group を求める．

$$(X,Y) = \sum_{i=1}^{num\_iog} \left(\frac{stripes\_X_i}{stripe\_cnt} X_i, \frac{stripes\_Y_i}{stripe\_cnt} Y_i\right)$$

num_iog はファイル断片が格納されている IO Group の数，$stripe\_cnt$ はファイルのストライプ数，$stripes\_X_i$，$stripes\_Y_i$ は IO Group (X_i,Y_i) 内のファイル断片が格納されている IO ノード数を意味する．つまり，IO ノードが多く使われている IO Group に近い IO Group をファイルの IO Group としている．ただこの方法では多数の IO Group にまたがる OST にファイル群が配置されている場合，また離れた IO Group 下の OST に配置されている場合には中間に偏ってしまう．そのため，IO Group の数に閾値を設け，選択アルゴリズムを切り替えることも検討している．

IO Group 計算後，KV<(X,Y), ファイルパス > を生成し，Shuffle する．

(3) (1)，(2) により，同じ IO Group に属す計算ノードとファイルが集まる．これらをランダムに対応させ，KV<Node_ID, ファイルパス > を生成し Shuffle する．これにより，ノードにファイルが割り当てられる．

6. 評価

提案手法を用いた場合のファイル読み込み性能の評価を

京コンピュータ上で行った．評価は，kmr_assign_file()等でノードにファイルを割り振った後に，そのファイルを読み込む処理を Map タスクとして，全ノード並列実行して時間を計測した．2 つの設定にて評価を行ったが，いずれも 3,072 ノード（4×4×16×2×3×2，16 IO Group）を用いている．読み込むファイルのサイズは 1MB, 2MB, 4MB と 2 倍刻みで 1,024MB までとした．

6.1 評価 1: 各ノードが異なるファイルを読むケース

各ノードが異なるファイルを処理する，一般的な Map/Reduce タスクの IO パターンにおける読み込み性能を評価した．ファイル数はノード数とし，ストライピング設定は京コンピュータ標準のストライプサイズ 1MB，ストライプ数 12 とした．結果，確保した 192（16 IO Group x 12 OST）の OST 全てに全ファイル分散して格納されるケースである．

本評価では，(1) ファイルアクセスの局所性を考慮したタスク配置を行う提案手法と，(2) 考慮せずランダムに配置する方法，および (3) ノードローカルなランクディレクトリを用いた場合の性能比較を行う．ランクディレクトリは MPI プロセスローカルのストレージであるが，実体は MPI プロセスを実行しているノードと同じ IO Group 内の IO サーバ上に，ストライプ数 1 で格納されているファイルである．各 MPI プロセスはランク値に該当するファイルをループバックマウントしている．このため，MDS へのアクセス負荷は削減されるが，IO スループットは並列ファイルシステムに直にアクセスする場合よりも劣る．(1) と (2) では図 4 における X-Y 軸方向の IO アクセス競合の頻度が異なるため，競合を軽減する提案手法の効果を確認する．また，(3) と比較することにより，X-Y 軸方向の IO アクセス競合と MDS アクセス負荷の無い理想的な場合に対する効率を確認する．

図 5 にファイル読み込みの性能を示す．凡例「Affinity」が提案手法であり，ファイル位置を考慮してタスクが配置される．凡例「Random」はタスクで処理するファイル名順に全ノードに 1 タスクずつ割り振った場合であり，ファイル位置は考慮されていない．凡例「Rank Directory」は京のランクディレクトリを用い，各 MPI プロセスローカルのファイルシステムにファイルを配置し，それぞれが読み込む場合である．横軸が読み込みファイルサイズ，縦軸が「Rank Directory」に対する相対性能である．それぞれ 5 回実行し，その最小値を示している．最小値とした理由は 2 つある．1 つは京コンピュータでは全ノードからアクセスされる MDS は 1 つしか無く，実行中の他のジョブの影響を受け，MDS アクセス性能が変化すること．もう 1 つは IO Group を占有利用した場合にも，バックグラウンドで他ジョブのステージングが行われているため，IO スループットに影響がでるためである．このため，外乱の影響を最も受けていないと思われる最小値を採用した．

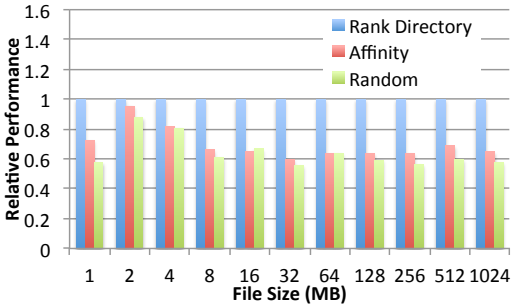

図 5 評価 1: 異なるファイルを読み込むときのリード性能

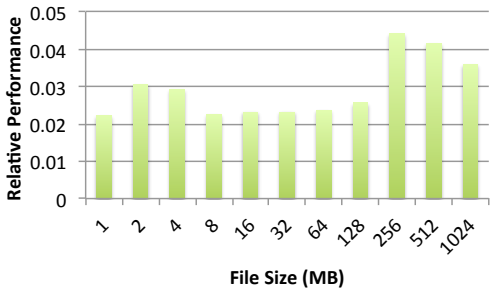

図 6 評価 1: ファイル割り振り処理の相対性能

全体的に Rank Directory の性能が高い．Rank Directory と Affinity の違いとして MDS へのアクセスの有無があるが，性能差の原因の調査は本稿には間に合わなかった．Affinity は Random に対して，1GB ファイルの時点で 12%の性能向上となっており，平均して 9%の性能向上となった．Affinity では計算ノード直下，または隣接する IO Group 下の IO ノードに対してファイルアクセスが行われるが，Random では計算ノードとは異なる IO Group 下の IO ノードへのアクセスが頻発し，図 4 に示す IO ノード間での X-Y 通信にて競合が発生するためである．

図 6 に Random のファイル割り振り処理性能に対する，Affinity の相対性能を示す．平均して 3%の性能しか出ていないが，Affinity の場合はサイズによらず 0.35 秒の実行時間であったため，タスクの処理時間によっては問題とはならない．

6.2 評価 2: 全タスク同一ファイルを読むケース

Map/Reduce 処理において，共通データベースにアクセスする等，全タスクが同一ファイルを読み込む IO パターンでの読み込み性能を評価した．タスク数はノード数と同じ 3,072 とし，各タスクが読み込むファイル数は 1 とし，京コンピュータ標準のストライピング設定とした．結果，確保した 192 の OST に対して，ファイルは 12 OST にのみ格納され，ファイル配置に偏りのあるケースである．

本評価では，(1) ファイルアクセスの局所性を考慮したタスク配置を行う提案手法と，(2) 考慮せず全ノードに均

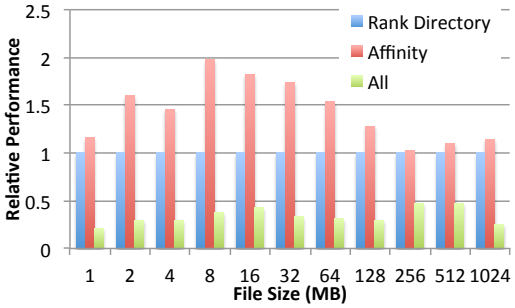

図 7 評価 2: 異なるファイルを読み込むときのリード性能

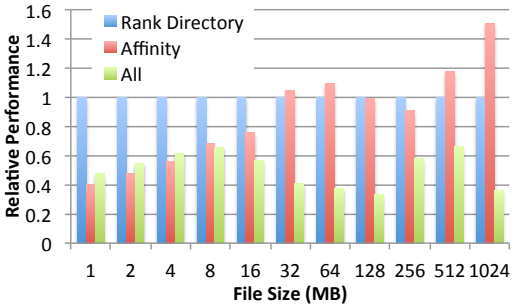

図 8 評価 2: 異なるファイルを読み込むときのタスク実行性能

等にタスク配置する方法，および (3) ランクディレクトリを用いた場合の性能比較を行う．(1) と (2) では図 4 における X-Y 軸方向の IO アクセス競合の頻度が異なるため，競合を軽減する提案手法の効果を確認する．(3) と比較することにより，ファイル共有する場合 (1,2) と局所性を高めるために共有しない場合 (3) の効率の違いを確認する．

図 7 にファイル読み込み性能を示す．凡例「Affinity」と「Rank Directory」は 6.1 節の評価と同じであり，「All」は全ノードに 1 タスク割り振り，各ノード 1 ファイルを読み込む場合の性能である．横軸が読み込みファイルサイズ，縦軸が「Rank Directory」に対する相対性能であり，それぞれ 5 回実行し，最小値を示している．

6.1 節の結果と異なり，Rank Directory よりも Affinity の方が優れた性能を示している．これは IO ノードにおけるキャッシュの影響である．Rank Directory の場合には，ファイル内容は同じであっても全ノード異なるファイルを読み込んでいるため，キャッシュが有効に働かない．一方で Affinity や All の場合には，全ノードが並列ファイルシステム上の同一ファイルを読み込むので，ファイルの実体を保存する IO ノードにてキャッシュが有効に働く．All に対する Affinity の相対性能は 8MB のときが最大で 5.2 倍，平均して 4.5 倍の性能であった．一方で，我々の論文 [20] にて提案してた Affinity-Aware File Access を用いると，All に対する相対性能は 1,024MB のときが最大で 75.5 倍，平均して 15.4 倍の性能であった．Affinity-Aware File Access は単一ファイルの同時読み込みしかサポートしていないが，アクセスパターンに応じて読み込み手段を変える必要があると言える．

図 8 にジョブ実行性能を示す．サイズが 16MB 以下において，読み込み性能は Affinity は Rank Directory を上回っていたが，ジョブ実行性能では逆になっている．これは Affinity ではファイルアクセスの局所性を高めるため，特定ノードにタスクが集中しているためである．実際，今回は 1 ノードあたり最大 16 タスク割り振られている．ただ，KMR では OpenMP によりタスクをノード内並列して実行するため，京の計算ノード内コア数 8 に合わせて，同時に 8 タスク実行されること，また同一ファイルを読み込んでいるため，計算ノード側でのバッファキャッシュが利用されるため，性能比も 2.5 倍程度となっている．

7. 考察

7.1 他システムへの適用

本研究では対象システムには京コンピュータを選択したが，並列ファイルシステムを使用しており，ファイルサーバと計算ノード間の距離を定義できるシステムであれば，他のシステムにも適用可能である．実際，Blue Gene ではファイルシステムに GPFS を採用しており，計算ノードからは IO Forwarding を行う IO ノードを介してデータアクセスが実現される [21]．並列ファイルシステムへのアクセスは IO ノード単位となるため，京コンピュータよりもアクセス競合は減ると考えられるが，IO ノードとファイルサーバの局所性を考慮したタスク割当を行うことで，本研究の成果と同等な結果が期待できる．

7.2 負荷分散の実現に向けて

提案手法を適用すると，局所性を高めるために，少数のノードにのみ多数のファイルが割り振られ，そのファイルを入力とするタスク実行時に負荷バランスが崩れて，タスクの実行時間にばらつきが生じうる．これを防ぐには，静的割り振り時に均等ファイルを割り振ることも可能ではあるが，タスク実行時間が入力サイズに依存しない場合には対応できない．そのため，動的負荷分散手法が求められる．

動的負荷分散を行うためのコストモデルについて考察する．ノード A に割り振られているタスク T の一部である t をノード B が代わりに実行する場合を考える．このときタスクは局所性を満たすように配置されているとする．簡単のため，各タスクは計算部分と IO 部分に分離でき，計算量と IO サイズは等しいとし，それぞれ $Comp$ と $Size$ で表す．ノードの計算性能を M_p，IO 性能を IO_p とすると，ノード A における実行時間は次の通りに表される．

$$Time_A = (T - t)(\frac{Comp}{M_p} + \frac{Size}{IO_p}) \quad (1)$$

t タスクをノード B で実行する場合，該当ファイルにアクセスするときの帯域には変化は無いと考えられるが，距離が離れる分，アクセス遅延が生じる．これを L と表す

と，ノード B における実行時間は以下になる．

$$Time_B = t(\frac{Comp}{M_p} + \frac{Size}{IO_p} + L) \quad (2)$$

京の場合，トーラスの 1 ホップ遅延を l，ノード B からファイルへのアクセス時に横断する IO Group の数を hop，隣接する IO Group 間の IO 通信数を c とすると，L は $l \times hop \times c$ と表せる．ただ，hop はノード B の位置に，c は他ノードの IO に依存するパラメータである．

以上より，2 つのノードの実行時間が均等になるときに負荷分散が実現されるので，その指標は以下の式のとおりである．

$$Min(Time_A - Time_B)^2 = \quad (3)$$
$$Min\big[(T-2t)\{\frac{Comp}{M_p} + \frac{Size}{IO_p}\} - t \times l \times hop \times c\big]^2$$

今後，このモデルの改良と KMR への実装を進める．

8. まとめ

計算科学アプリケーションのワークフローを MapReduce モデルにて，大規模並列システムで高速に実行するために，我々は並列ファイルシステムへのアクセスの局所性を考慮した負荷分散の実現を目指している．本研究ではそれの実現のため，タスクの入力ファイルから近い計算ノードに該当タスクを割り振る手法を提案し，我々の開発している MapReduce 処理系 K MapReduce に実装し，ファイル読み込み性能の評価を行った．その結果，N ファイルを N ノードが読み込む評価において，ランダムにファイルをノードに対応させた場合に対して，本提案は平均して 9%の性能向上を達成した．また，1 ファイルを N ノードが読み込む評価においては，本提案は平均して 4.5 倍性能向上となった．さらに，データの局所性を考慮した動的負荷分散実現に向けてのコストモデルを構築し，今後の同機能の実現を目指す．

謝辞 本論文の結果（の一部）は，理化学研究所のスーパーコンピュータ「京」を利用して得られたものです．

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. 1–13 (2008).

[2] Langmead, B., Schatz, M. C., Lin, J., Pop, M. and Salzberg, S. L.: Searching for SNPs with cloud computing, *Genome Biology*, Vol. 10, No. 11 (2009).

[3] McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M. and DePristo, M. A.: The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data, *Genome Research*, Vol. 20, No. 9, pp. 1297–1303 (2010).

[4] Tu, T., Rendleman, C. A., Borhani, D. W., Dror, R. O., Gullingsrud, J., Jensen, M. O., Klepeis, J. L., Maragakis, P., Miller, P., Stafford, K. A. and Shaw, D. E.: A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories, *International Conference for High Performance Computing Networking Storage and Analysis (SC08)*, No. November, IEEE, Ieee, pp. 1–12 (2008).

[5] : 計算科学ロードマップ（2014 年 3 月版），`http://hpci-aplfs.aics.riken.jp/`.

[6] : Welcome to Apache Hadoop, `http://hadoop.apache.org/`.

[7] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems, *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 13–24 (2007).

[8] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J. and Fox, G.: Twister: a runtime for iterative MapReduce, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, New York, NY, USA, ACM, pp. 810–818 (2010).

[9] Zhang, B., Ruan, Y., Wu, T.-L., Qiu, J., Hughes, A. and Fox, G.: Applying Twister to Scientific Applications, *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 25–32 (2010).

[10] Li, X., Wang, Y., Jiao, Y., Xu, C. and Yu, W.: CooMR: Cross-task Coordination for Efficient Data Management in MapReduce Programs, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, SC '13, ACM (2013).

[11] Zaharia, M., Borthakur, D., Sarma, J. S., Elmeleegy, K., Shenker, S. and Stoica, I.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, *in Proceedings of the 5th European conference on Computer Systems*, pp. 265–278 (2010).

[12] Yoo, R. M., Romano, A. and Kozyrakis, C.: Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system, *2009 IEEE International Symposium on Workload Characterization IISWC*, IISWC '09, Ieee, pp. 198–207 (2009).

[13] Chen, R., Chen, H. and Zang, B.: Tiled-MapReduce : Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling, *The 19th international conference on Parallel architectures and compilation techniques*, PACT '10, ACM, pp. 523–534 (2010).

[14] Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U. A. and Pasquini, R.: Incoop : MapReduce for Incremental Computations, *Proceedings of the 2nd ACM Symposium on Cloud Computing*, Vol. 10, No. 6, p. 7 (2011).

[15] Stuart, J. A. and Owens, J. D.: Multi-GPU MapReduce on GPU Clusters, *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, Washington, DC, USA, IEEE Computer Society, pp. 1068–1079 (2011).

[16] Kwon, Y., Balazinska, M., Howe, B. and Rolia, J.: SkewTune: Mitigating Skew in Mapreduce Applications, *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, ACM, pp. 25–36 (2012).

[17] Gufler, B., Augsten, N., Reiser, A. and Kemper, A.: Handling Data Skew in MapReduce, *CLOSER '11: Proceedings of the 1st International Conference on Cloud Computing and Services* (Leymann, F., Ivanov, I., van Sinderen, M. and Shishkov, B., eds.), pp. 574–583 (2011).

[18] Gufler, B., Augsten, N., Reiser, A. and Kemper, A.: Load Balancing in MapReduce Based on Scalable Cardinality Estimates, *2012 IEEE 28th International Conference on Data Engineering*, IEEE, pp. 522–533 (2012).
[19] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y.: Cilk: an efficient multithreaded runtime system, *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, New York, NY, USA, ACM, pp. 207–216 (1995).
[20] Matsuda, M., Maruyama, N. and Takizawa, S.: K MapReduce: A Scalable Tool for Data-Processing and Search/Ensemble Applications on Large-Scale Supercomputers, *IEEE Cluster 2013 Conference* (2013).
[21] Yu, H., Sahoo, R., Howson, C., Almasi, G., Castanos, J., Gupta, M., Moreira, J., Parker, J., Engelsiepen, T., Ross, R., Thakur, R., Latham, R. and Gropp, W.: High performance file I/O for the Blue Gene/L supercomputer, *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* (2006).