メニーコアマシン環境における並列 B 木マージを利用したビッグデータ処理方式の検討

天沼 駿亮　鷹野 俊介　加藤 和彦　阿部 洋丈

CPU 単体のクロック周波数の増加が限界になりつつある現状から、一筐体で数十個の CPU コアと数 TB 規模の主記憶を搭載するメニーコアマシンがこれから普及していくことが予想される。メニーコアマシンは 1 台のマシンで多数の CPU を持つため、処理をする上でクラスタ環境が必要だった処理をメニーコアマシンで行うことが出来る。そのため、ビッグデータの並列分散処理に代表されるような、現在はクラスタ環境を利用して実現されている処理をメニーコアマシンで実現することができれば、計算機の運用コストが下がると期待できる。しかし、メニーコアマシンの持つ性能を活かすためには従来のロック手法では並列度の低下を招いてしまう。そこで、本研究では複数の B+-tree を用いてデータにロックを掛けずに並列分散処理を行う処理手法を提案する。この手法では各処理ノードに B+-tree を割当て、各々のノードで処理を行いその結果を B+-tree に格納する。その後、全ての B+-tree を一つの B+-tree にマージすることによって結果を得る。その際、各ノードが限られた範囲にのみ操作を行うことでロックフリーな並列分散処理を実現している。この手法を用いたワードカウントのアプリケーションを作成し、単語データを用いて検証を行った

We forecast that much attention will be paid on many-core machine because manufacturing technology is approaching to technical limit of performance. Many-core machine processes in parallel without connecting a number of cluster computers because the machine equipped with many CPUs and tremendous main memory. We consider that managing cost of machines could be cut by constructing parallel distributed processing on a single many-core machine. In general, a process is independent from other process in parallel processing because confliction of threads tend to cause performance degradation. We propose a highly parallel processing scheme with merging multiple B+-trees on many-core machine without locking data. We prepare large amount data divided into the number of threads and assign data to each threads and processing each data. Each thread makes a B+-tree based on each data after processing, and they merge each B+-tree into one. We had conducted an experiment using the program of word count with this method to verify the efficiency of the proposed scheme.

1 はじめに

私たちを取り巻くデータ量は爆発的な増加を見せており [1]、これらのデータを可能な限り分析、活用して価値を見出す試みに注目が集まっている [2]。CPU のクロック周波数の向上が頭打ちとなっている現状から、プロセッサ数を増やして性能向上を図る傾向にある。そのため、プロセッサの数を活かした並列分散処理手法が関心を集めている [3]。現在、大規模なデータ処理を行うシステムとして、Google 社が開発した MapReduce [4] や、そのオープンソース実装である Hadoop[5] をはじめとするクラスタ計算機を用いた分散処理形態が広く用いられている。また、Hadoop 上で動作し、中間データの保存用に活用することでより高速な処理を可能にした Spark[6] も登場している。クラスタ計算機を用いた分散処理基盤はスケーラブルな処理基盤を比較的容易に構築することが出来るという利点を持つ。しかし、その反面、高速な処理を期待する上で複数台のクラスタマシンを用いることを前提としているため、システム管理に多大なコス

BigData Processing on Many-core Machine with B+-tree.

Amanuma Shunsuke, Takano Shunsuke, Kato Kazuhiko, Abe Hirotake, 筑波大学大学院　システム情報工学研究科　コンピューターサイエンス専攻, Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba.

トを要するという問題点がある。

一方で、一筐体で数十規模の CPU コアと 1 から数 TB 以上の大容量主記憶を持つメニーコアマシンが入手可能となってきている。クラスタマシンは台数を増やしていくことで数千から数万ものコアを用いて並列処理を行う基盤を作成することが出来る。その反面、クラスタマシンほどのスケーラビリティを期待するのは難しい。しかし、メニーコアマシンは 単一のマシン内で動作することから CPU 同士の通信や CPU とメモリとの通信をクラスタ環境よりも高速に行うことが出来る。よって、メニーコアマシンで処理が可能な規模であれば、クラスタ環境よりも効率的に処理を行うことができる可能性がある。さらに、メニーコアマシンはクラスタ環境に比べて、マシン台数を減らすことが出来るため、管理コストの大幅な削減が期待出来るという利点がある。また、サーバ運用の視点から見たとき、クラスタ環境で運用する場合に比べてメニーコア環境で運用する方が、消費電力やサーバー管理コスト、パフォーマンスの点で優れていると考えられる [9]。また, 処理を行った中間データを保持する上で十分な大きさの主記憶を保持していれば、HDD 等の記憶媒体からデータを読み取る場合に比べてアクセスコストを削減できる。よって、機械学習のような何回も同じ実行を繰り返すアルゴリズムはクラスタ環境に比べて、メニーコアマシン環境でより効率的なデータ処理を実現することが出来る。しかし、クラスタ環境と同じ方法では、メニーコアマシンの性能を活かしきることは困難であると思われる。そのため、メニーコアマシンの性能を引き出すためのソフトウェア技術を新たに開発する必要がある。

本研究ではメニーコアマシン上でほぼロックフリーに並列分散処理を行う処理手法の提案を行う。

ビッグデータ処理の多くが並列性を引き出しやすい構造とみなせるということを利用し、複数に分割されたビッグデータそれぞれに対して対応するノードで個別に処理を行い、最後にロックフリーにノードの結果の集計を行う。

本論文は以下の章によって構成される。第 2 章では、本研究に関連する技術と研究における基本的な概念について説明する。第 3 章では本研究で用いるプログラムの処理手法について説明する。第 4 章では、比較実験について説明する。第 5 章では本論文の結論と今後の課題について述べる。

2 基本概念

2.1 B+-tree

B+-tree は木構造の一種である B-tree から派生した構造である。B-tree は木の全てのノードに key-value ペアの並びを所持し、その中でも最下層のノードを葉ノードと呼ぶ。葉ノード以外のノードを内部ノードと呼び、そこには key-value ペアだけでなく他の内部ノード、もしくは葉ノードへのポインタを格納している。このデータ構造はオーダーが最悪でも O(logn) であると言う特徴を持つため、オーダーが O(n) であるブロック構造に比べて少ない計算回数で処理を行うことが出来るという利点を持つ。B+-tree は B-tree の派生系であり、葉ノードには key-value ペアの並びを持ち、内部ノードにはキーと内部ノード、もしくは葉ノードへのポインタが格納されている構造となっている。

2.2 一貫制御

本研究では B+-tree を用いて並列処理を行う。並列処理を行う上でスレッド間のデータ競合を防ぎ、データ更新競合を制御する必要がある。破損からデータを守る手段としてはロックをかけるのが一般的であり、データをロックするための手法として楽観的ロックと悲観的ロックが存在する。

楽観的ロックは処理を行うレコードに対して更新タイムスタンプを保持し、レコードの更新の直前に更新タイムスタンプが最初の状態と変わっていないかどうかで排他処理を行うというロック方法である。この方法はスレッドの競合がないため、デッドロックを気にしないでいい反面、一つのデータに対して更新処理が複数ノードから行われる並列処理には向いていない。

悲観的ロックは処理を行うレコードに対し、他のスレッドからの更新を受けていないかどうかを最初に確認し、抑止されている場合は待機し、そうでない場合は他からの更新を抑止したうえで処理の終了時に抑止を解除する。この方法は複数のスレッドから更新を受

けてもデータの破棄が行われないため並列処理向きではあるが、複数スレッドから同時にアクセスがあった場合にデッドロックが生じる可能性が存在する。

この悲観的ロックを用いた古典的な B-tree の並列化方法 [7] が存在する。B-tree の持つノードが保持できるデータ量の最大数を N とし、データの挿入の際、データ数が N 以下であるならば safe であると判別し、データを削除する際、データ数が N/2 以上であれば safe であると判別する。ノードの探索を行う時、対象が safe であるノードであれば葉に処理を行っても再帰的な変更を必要としないため、ロックをかける必要がないことから並列度を高めることが出来るという考えである。メニーコアマシンに対してこの手法を適用する場合、スレッドの数がとても多くなるため、更新待ちのスレッドが多くなることが予想され、処理速度の低下につながると考えられる。そのため、本研究で用いるプログラムは B+-tree の特性を利用したロック不要な並列処理を行うことでこの問題を解決する。

3 B+-tree を用いた並列分散処理手法

3.1 基本方式

あらかじめ対象データを各スレッドのメモリに乗る大きさになるよう分割して、メモリに配置する。必要であるならば HashMap などのデータ構造に一旦データを格納するなどの処理を行う。次に、データに対して各ノード毎に計算処理を行い、ノードの数だけ B+-tree を作成する。その後、各 B+-tree の要素を一つの B+-tree にマージし、結果を集計する。この手法の利点は中間データをメモリに保持することを前提としているため、MapReduce 等のクラスタ処理における shuffle フェーズの部分で中間ファイルを HDD に書き込む処理を行わずに集計処理を行うことが出来る点と、ロックフリーにマージを行うことが出来るため、デッドロックの心配をせずに並列処理を行うことが出来る点にある。

3.2 実装

以下にワードカウントプログラムの実装を例として説明を行う。プログラムは図 2 のように、おおまかに Insert, Merge, Reorganize の 3 つのフェーズから成り立っている。まず Insert で各々のスレッドでファイルを読み込み、B+-tree を作成する。次に各スレッドで作成された B+-tree を一つの B+-tree に Merge する。最後に Merge された B+-tree を Reorganize して、バランスすることで、全ての単語を内包した B+-tree が完成し、ワードカウントが終了する。この全てのフェーズにおいて出来るだけロックを行わずに処理を行う。

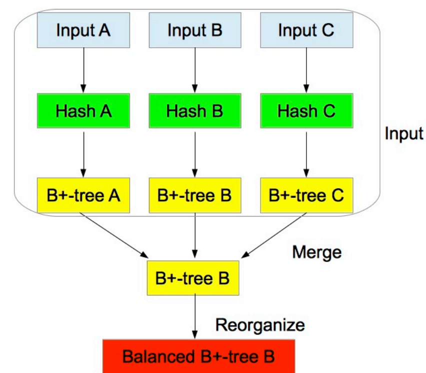

図 1 各ノードの処理の流れ

3.2.1 Insert

まず、ファイルから単語を切り出し、B+-tree に登録していく作業を行う。しかし、いきなり単語一つ一つに対して B+-tree insert を行っていくとすると、単語を B+-tree に挿入するオペレーションの数が単語の個数と同数となってし まうため、データの規模が大きくなればなるほど処理速度に遅延が生じると考えられる。

そこで、最初にハッシュテーブルに単語を追加していくことで、B+-tree への挿入回数をへらす。ハッシュテーブルはデータの入力、取得をO (1) で行うことが出来る。一方木構造ではO (log(n)) でデータの取得を行うため、データ数が 100 倍になれば約 2 倍、データ数が 1000 倍になれば約 3 倍程度の時間がハッシュテーブルに比べて必要となってくる。そのため、単語をあらかじめハッシュテーブルに挿入し、単語の重複をなくした状態で B+-tree に挿入していくことで処理速度の向上が期待できると考えられる。

ハッシュテーブルに代入することによってキーと値のペアを作成する。ワードカウントでは単語をキー，単語の数を値として格納する。この処理を分割されたビッグデータに対して並列に行うことによって、各スレッドがスレッド内では重複を持たない単語のキーと値という中間結果を保持している状態になる。この状態から ハッシュを B+-tree に変換する作業を行い、次のフェーズに移行する。

ワードカウントのプログラムでは単語を抜出、格納するという簡潔な処理のみを行うことや、単語が重複することを考慮した結果から ハッシュテーブルを用いてキーと値の格納を行っているが、それ以外の複雑な処理を要するプログラムでは赤黒木や B+tree などの他のデータ構造に直接中間結果の格納を行うことも考えられる。また、この手法は主記憶に中間結果を保持できることを仮定として考えている。これは、ビッグデータ本体が主記憶に収まりきらないデータ量であっても、演算や関数の計算によってカウントに必要なデータのみを抜き出すことによって主記憶に格納することが可能であるという仮定に基づくものであり、序論で述べた Spark システムにもこの仮定が使われていることからも合理的な考えであると思われる。

3.2.2 Merge

このフェーズでは前のフェーズでキーと値に分けられ、B+-tree に格納された中間データに対して B+-tree の特性を用いたマージ処理を行うことで最終結果を得る。B+-tree はその性質から全てのキーを葉に保有している。そして、その葉へのポインタを各内部ノードが所持しており、各葉ノードは内部ノードの持つポインタによって排他的に分割された構造を取る。つまり、キー空間 key0 から key10 を各葉ノードの key を境に分割しているということになる。その様子を図 3 に示す。 例えば葉の一段上のノードを参照するとそのノードにあるキー (key2,key4,key7,key9) はこの木に格納される可能性のあるキーを排他的に分割していることになる。つまり、この B+-tree はキー空間を 5 つの区間に分割していることになる。

このようにキー空間全体が内部ノードのキーによって分けられることを利用してマージ処理を行う。まず、マージ先となる B+-tree を全スレッドから一つ

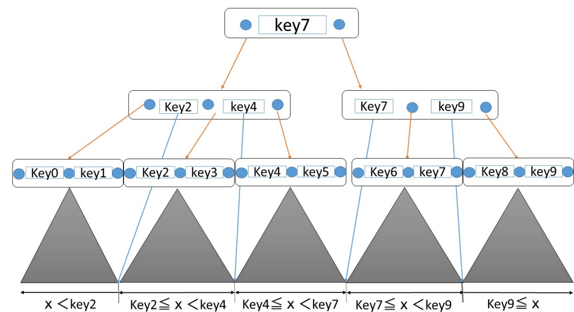

図 2　B+-tree によるキー空間の分割

選ぶ。次に分割されたキー空間にそれぞれ独立したスレッドを割当て、各スレッドがマージしたい全ての B+-tree から、割り当てられたキー空間の中にあるキーのみに対してマージ処理を行う。これによって、各々のスレッドがあらかじめ決められたキー範囲にのみしか処理を行わないため、全ての B+-tree にロックをかけずにマージ処理を行うことが出来る。

3.3 Reorganize

このフェーズではマージ処理を行った後の B+-tree をバランスする処理を行う。先ほどのマージ処理を終えた後、B+-tree には全ての単語が内包されている状態になる。しかし、キーの範囲によって単語の数はバラバラであり、マージを終えた段階ではあるキーの範囲のみとても単語数が多くなっていると言う状態も考えられる。その状態だと葉ノードがオーバーフローしている状態となり、B+-tree としての形ではなくなってしまう。そのため、オーバーフローしているノードに対してバランス処理を行い B+-tree の形にする必要があるのである。

このアンバランスな状態は B+-tree を下のレベルから順々にスプリット処理をしていくことで解決する。まず一番下の葉ノードに対してオーバーフローしているかどうかの判断を行う。その中でオーバーフローしているノードがあった場合、そのノードに対してスプリット処理を行う。葉ノードが全てオーバーフ

ローしていない状態になったことを確認したら次はその一つ上の中間ノードに対して再度オーバーフローしているかどうかの判断を行う。この操作を繰り返し行い、親ノードに達するまで処理を終えた時点でこのフェーズを終了するというのが一連の流れである。

このフェーズにおいて、各ノードに対して各スレッドを割り振り、オーバーフローしているかどうかの判断をさせ、レベルごとに処理ノード全てで同期を取ることによってノードのレベルの中で並列化することが出来る。

4 実験

実際にこの処理手法で実装したワードカウントプログラムを用いて実験を行い、マルチコアを有効活用できているかを明らかにする。この処理手法はメニーコア環境で動作することを目的に作成した手法であるが、NUMA アーキテクチャを考慮した方式の検討がまだ不十分であるため、今回は UMA アーキテクチャに絞ったマルチコア環境下で実験を行い、提案手法の可能性を検討する。

4.1 実験環境

実験環境として, CPU が Intel Xeon CPU E7-4870 2.40GHz 80cores (10cores × 8nodes), メモリが 1TB, OS が, Linux 3.2.0-4 の環境を用いた。 このマシンは NUMA アーキテクチャであり、メモリ 1 つに対して CPU コアが 10 個繋がった構造を一つのノードとし、それを 8 ノード分持つ。本実験ではこの内 1 ノードのみに限定して、10 コアの環境下で実験を行う。

4.2 実験内容

入力データとして、Wikipedia の英語版のページから単語を抜出し、タブで区切りを行ったデータ 10GB 分と、Linux のソースコード [8] を全て txt ファイルに起こしたデータ 180MB 分用いる。Wikipedia のデータは 1GB を 10 ファイル、Linux のデータは小さいファイルが 28267 ファイル集まったデータセットである。そのデータに対して使用スレッド数を変化させ、ワードカウントを行い、処理速度の変化を観測する。また、各々のファイルの処理に対してフェーズごとに時間を計測し、各々のフェーズでどれだけの処理時間がかかっているかを確認する。

4.3 ワードカウントプログラムの処理の流れ

本実験で用いるプログラムの処理の流れの説明を行う。まず巨大なファイルを各スレッドの保持するメモリに乗るサイズまで分割する。それぞれのスレッドはそのファイルから単語をキー、個数をバリューとした組み合わせを抜き出し、ハッシュに保存していく。ファイル内全ての単語をハッシュに格納し終えた後、B+-tree へと格納する。入力の処理が全て終わった後全ての B+-tree に対して merge,reorganize の処理を行い、集計することで処理を終了する。

4.4 実験結果

図 3, 4 に Linux ソースコードに対しての実験結果、図 5, 6 に Wikipedia 単語データに対しての実験結果を示す。図 3, 5 からどちらのデータに対しても割り当てるスレッドの数が増加するに連れて、処理時間が減少していることが分かり、単一ノード内に於いてスケーラブルな処理が出来ていることが確認できた。また、図 4, 6 から、どちらの場合もほとんどの処理時間を insert と merge が占めていることが分かる。その中で、スレッド数によって処理時間が短くなっているのは insert の部分であり、merge と reorganize にかかる時間は 1 スレッド時の実験結果の約 7 分の 1 程度の処理時間から変化が無いことが分かる。また、reorganize に関しては図 6 の実験ではほとんど処理時間がかかっていないことから、ファイルが大きくなると、小さいファイルに比べて reorgznize にかかる時間はとても短くなっていくということが分かった。これらの結果から、まず merge 処理の時間が変わっていないことについて考察する。merge 処理を並列に行っているため、merge の部分がスレッド数の増加に応じて速度が上がっていなければならないのだが変化が無い。原因として考えられるのが、データに偏りがあるというケースである。マージ処理をするにあたって、一部のスレッドにのみデータが集中してしまい、並列な動作が実質的に行えていない状態になって

いる。もしくは、並列マージは正常に動いているが、同期のオーバーヘッドに時間がかかってしまっているケース等も考えられる。次に、図 4, 6 を比べて各々のデータの処理時間についてフェーズごとに比較し考察を行う。insert 部分にかかる時間は対象となるデータの大きさによって変化するため、180MB と 10GB のデータとではデータの大きさでは 50 倍程度の処理時間の差が生じると考えられる。実験結果からは約 40 倍の差が出ていると読み取れるが、ファイルの個数が Linux 側の方がとても多く、少ないとはいえ読み取りに時間がかかっていることを考慮に入れると、速度としては想定通りであると言える。次に merge 部分に関してだが、こちらはデータ全体の要素数に応じて処理時間が変わると考える。merge 処理では全ての B+-tree の該当する範囲に対して各々のスレッドがアクセスし、要素の集計を行う。つまり、アクセスする要素の数が多ければ多いほど処理の時間が長引くと考えられる。今回実験で用いたデータの要素数の差は約 30 倍程度の差があったため、これだけの時間の差が生まれたと考えられる。最後に reorganize 部分に関してだが、reorganize にかかる時間は作成される木の構造や葉の数、データ全体の要素数等、複数の要素によって変化すると考えられる。今回の実験ではどちらも 3 段の木が作成されていたものの、葉の数と内部ノード、データの要素数の全てが異なっていたため、処理時間に差が出ている。仮に葉の数以外の要素が全て同じで、葉の数が 2 倍異なる木に reorganize を行った場合、処理時間は 2 倍異なると考えられる。また、木の高さが増えればその段数分 reorganize の回数は増え、逆に内部ノードの数が増えればその分並列に処理できる葉の数が増えるため処理は高速化すると考えられる。

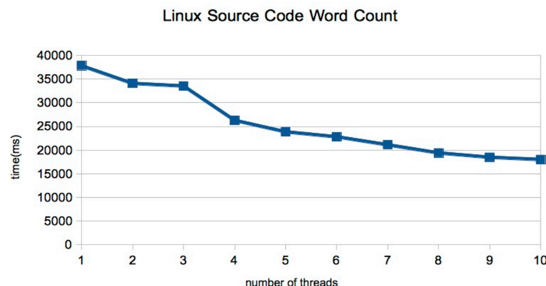

図 3　スレッド数の変化による処理速度の推移 (Linux)

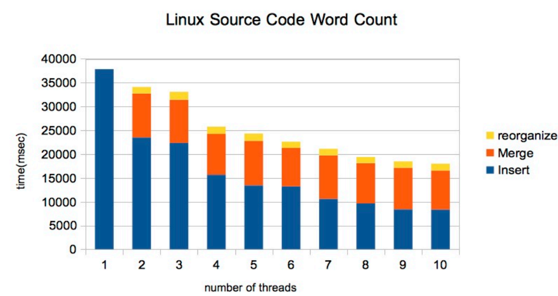

図 4　スレッド数の変化による各処理時間の内訳 (Linux)

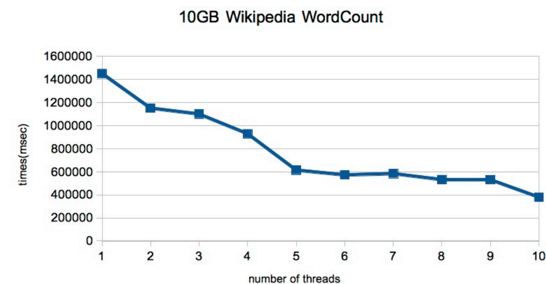

図 5　スレッド数の変化による処理速度の推移 (Wiki)

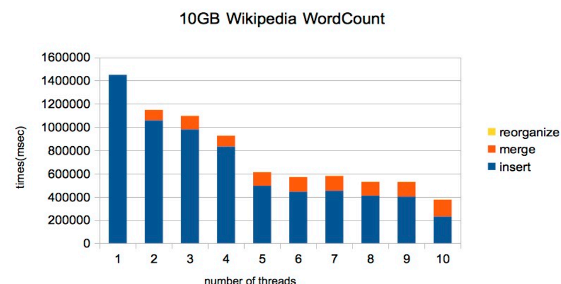

図 6　スレッド数の変化による各処理時間の内訳 (Wiki)

5 結論と今後の課題

本研究ではメニーコア環境に於けるロックフリーな並列分散処理方式の提案を行った。B+-tree を使った処理方式によってデータにほぼロックをかけずに並列分散処理を実現した。また、実験によってスレッド数の増加に応じてスケーラブルであることが確認できた。今後の課題としては、まず、メニーコア環境で実際に使用するための実装を行って行きたいと考えている。また、ワードカウント以外のアプリケーションの実装や、機械学習等の繰り返し処理に対する処理の最適化等を行っていきたいと考えている。

謝辞 本研究は JSPS 科研費 15K12006 の助成を受けたものです。

参 考 文 献

[1] John Gantz and David Reinsel: THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East - United States,2013

[2] Chen, Hsinchun, Roger HL Chiang, and Veda C. Storey: Business Intelligence and Analytics: From Big Data to Big Impact. MIS quarterly 36.5:1165-88 2012

[3] Akhter, Shameen, and Jason Roberts: Multi-core programming. Vol. 33. Hillsboro Intel press, 2006

[4] Jeffrey Dean and Sanjey Ghamawat: MapReduce:Simplified Data Processing on Large Clusters. Communication of ACM 51.1, 2008: 107-113

[5] Jeffrey Shafer: A storage Architecture for Data-Intensive Computing. Diss. Rice University, 2010

[6] Matei Zaharia, Mosharaf CHowdhury et al: Resilient Distributed Datasets: A Fault-Tolerant Abstruction for In-Memory Cluster Computing. USENIX Association, 2012

[7] R. Bayer and M Schkolnick: Concurrency of Operations on B-Trees. Acta Infomatica 9:1-21,1977

[8] http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.7.1.tar.bz2

[9] Raja Appuswamy,Christos Gkantsidis et al:Scale-up vs scale-out for Hadoop: time to rethink?, SOCC '13 Proceedings of the 4th annual Symposium on Cloud Computing Article No. 20, 2013