Vol.2010-HPC-126 No.5
2010/8/3

GPUを考慮したMapReduceのタスクスケジューリング

白幡晃一[†1] 佐藤仁[†1] 松岡聡[†1,†2,†3]

大規模データ処理のためのプログラミングモデルとしてMapReduceモデルがスケーラブルな並列処理が可能となるため注目されている。一方、GPGPUと呼ばれる、GPUを汎用計算に応用する技術の研究・開発が進んでおり、GPUのスーパーコンピュータやクラウドへの導入が進みつつある。しかし、MapReduceのタスクをCPU・GPUに資源を割り振る方法は、GPU特性の影響からアプリケーションごとに性能は異なるため自明ではない。CPUとGPUが混在する不均質な大規模環境を想定し、CPU上とGPU上で実行されているMapタスクの動的なプロファイルを利用してジョブ実行時間を最小化するハイブリッドオンラインスケジューリングを提案する。K-Meansアプリケーションで実験を行った結果、CPUのみの使用に対し、2GPUの使用とスケジューリングアルゴリズムの適用をした場合、CPUのみにスケジュールした場合に比べ、ジョブ実行時間において1.02-1.93倍の高速化を達成した。

Improving MapReduce Task Scheduling for CPU-GPU Heterogeneous Environments

Koichi Shirahata,[†1] Hitoshi Sato[†1]
and Satoshi Matsuoka[†1,†2,†3]

MapReduce is a programming model that enables efficient massive data processing in a large-scale computing environment such as supercomputers and clouds. On the other hand, recent such large-scale computers tend to employ GPUs to enjoy its good peak performance and high memory bandwidth. However, scheduling MapReduce tasks onto CPUs and GPUs for efficient execution is difficult, since it depends on running application characteristics and underlying computing environments. To address this problem, we propose a hybrid online scheduling technique for GPU-based computing clusters, which minimizes the execution time of a submitted job using dynamic profiles of map tasks running on CPUs or GPUs. Our experimental results using a K-Means application show that the proposed technique achieves 1.02-1.93 times faster than simple techniques, such as ones that CPU only or GPU only schedulings.

1. はじめに

近年は情報爆発時代と言われており、気象、生物学、天文学、物理学など様々な科学技術計算において大規模データ処理の重要性が高まっている。大規模データ処理のためのプログラミングモデルの研究・開発も進められており、その中でもGoogleが開発したMapReduce[1]プログラミングモデルはスケーラブルな並列処理が可能となるため注目を集めている。

一方、GPGPU[2]と呼ばれる、GPUを汎用計算に応用する技術の研究・開発が進んでいる。GPUは高いピーク性能とメモリバンド幅を持つことに加え、CUDA[3]等の開発環境の整備も進んできたため、スーパーコンピュータやクラウドへの導入が進みつつある。東京工業大学のスーパーコンピュータであるTSUBAME2.0では計算ノードに3台のGPUが搭載され、本格的なCPUとGPUのハイブリッドスパコンとして構築されることになっている。

しかし、このような不均質な大規模計算環境上でMapReduceなどの大規模データ処理を効率的に行うためのCPU・GPUに資源を割り振る方法は自明ではない。これは、計算ノード内のCPU・GPUの数、メモリ、ストレージへのI/Oバンド幅などの計算環境や、GPUに向き・不向きなどのアプリケーション特性に依存するためである。しかし、CPU・GPUいずれか一方しか利用しない場合や、単純にアイドルなCPU・GPUに資源を割り振ってしまうだけでは計算速度や消費電力などの効率が最適でない場合がある。そのため、CPU・GPUそれぞれの長所を活用するためのハイブリッド実行スケジューリングが必要となる。

そこで、我々は、CPUとGPUが混在する不均質な環境を想定したMapフェーズのハイブリッド実行とタスクスケジューリング手法を提案する。クライアントがCPU用とGPU用のアプリケーションバイナリを投入すると、マスターノードはMapタスクのCPU・GPUへの割り振りを、スレーブノードはCPU・GPU上でのMapタスクの実行を行う。また、このとき、Mapタスク実行時間のプロファイリングを動的に行い、ハイブリッド実行のタスクスケジューリングを行う(**図1**)。

K-Means[4),5)]アプリケーションで実験を行った結果、ハイブリッド実行とタスクスケジューリングを改良することにより、高速化を実現した。Mapタスク実行時間においてCPUに

†1 東京工業大学
†2 科学技術振興機構
†3 国立情報学研究所

© 2010 Information Processing Society of Japan

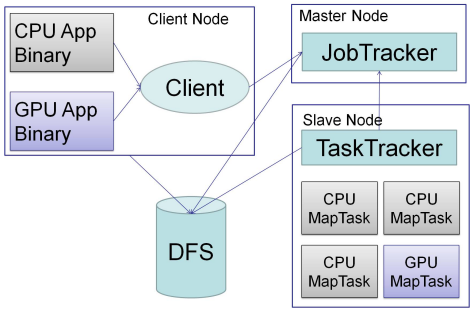

図 1 Hadoop 上での CPU・GPU ハイブリッド実行の概要
Fig. 1 The overview of CPU-GPU hybrid processing on Hadoop

対する GPU の加速倍率が 1.0-1.25 倍となる高速化を達成した。またジョブ実行時間において、CPU のみの使用に対し、2GPU の使用とスケジューリングアルゴリズムの適用を行った場合、1.02-1.93 倍の高速化を達成した。

2. MapReduce と GPGPU

この章では MapReduce の概要と主な実装について述べた後、GPGPU の概要を CPU との比較を中心に述べる。

2.1 MapReduce

MapReduce は大規模データセンターでウェブのデータ解析を効率よく処理するために Google によって開発されたモデルである。データを複数のノードに分散させるため、耐故障性や局所性に優れている。データ処理のプロセスには Map、Shuffle、Reduce の 3 つのフェーズがあり、Map フェーズで中間データとなる key-value ペアを生成し、Shuffle フェーズで同じ key に対して value のリストを生成し、Reduce フェーズで key-value をまとめあげ、最終出力となる key-value のペアを生成する。プログラマは Map 関数と Reduce 関数のみを書けばよく、並列化は自動的に行われ、データは分散システムに格納される。ウェブのデータ解析の他にも様々な機械学習アルゴリズムへ MapReduce を適用し、有効であるという事例も報告されている。

MapReduce の主な実装として Hadoop[6]、Phoenix[7]、Mars[8] などが挙げられる。Hadoop は GFS(Google File System) や MapReduce などのオープンース実装を行っている Java ソフトウェアフレームワークである。MapReduce の他に分散ファイルシステムに HDFS、データベースシステムに HBase などのシステムが搭載されている。Phoenix はプログラミング API とランタイムシステムを含む、共有メモリシステムのための実装である。Mars は GPU のための MapReduce フレームワークであり、データインテンシブなタスクやコンピューティングインテンシブなタスクを GPU で効率的に実装するためのフレームワークを提供する。Hadoop は企業や研究機関での様々な導入事例があり、広く普及しているため、今回は Hadoop を対象にした。

Hadoop の MapReduce アプリケーションを実行する際の実行過程は次のようになる: (1) クライアントが MapReduce ジョブを投入する。(2)JobTracker がジョブ全体の管理を行う。(3)TaskTracker がジョブを分割してできたタスクの実行を管理する。(4) 分散ファイルシステムが上記 3 つのエンティティ間のジョブファイルの共有に用いられる。マスターである JobTracker はスレーブ上のジョブの構成要素のスケジューリングや進行状況の監視、失敗したタスクの再実行の管理などを行う。スレーブである TaskTracker は JobTracker によって指示された Map タスクと Reduce タスクを実行する。各 Map タスクは入力データをチャンクサイズ (通常は 64MB) の大きさに分割した入力ファイルを持つ。

2.2 GPGPU

近年 GPGPU (General-purpose computing on GPU)[2] と呼ばれる、GPU を汎用的な計算に応用する技術が進歩している。GPU はもともと画像処理などの演算をパイプラインで実行していたが、GPU アーキテクチャの進化により、最近はパイプラインを制御するためにプログラマブルシェーダを使用することにより、柔軟性がもたらされている。これにより、グラフィックにかかわらず一般のアプリケーションにも GPU を自然に利用することができるようになった。

GPU は SIMD 型データ処理を行っているため単純な並列計算に向いている。また、スレッド数が非常に多いため、CPU に比べかなり高いピーク性能を発揮する。しかし、GPU は計算を行う前に CPU からデータ転送を行われる必要があり、その際のオーバーヘッドは無視できない時間となることが多い。また、条件分岐が入ってしまうとオーバーヘッドがかさみ、極端に効率が悪くなってしまう。一方、CPU は多岐にわたる用途に使用できるように進歩したため、大量のデータを複雑なロジックで処理することを得意としている。そのため、逐次計算や分岐の多い計算は GPU よりも向いている。また、GPU と協調して動作する場合、GPU は単純な計算しかできないのに対し、CPU は GPU に計算用のデータの送受信や、前処理・後処理などの役割を担う。そのため、GPU は単独では動作できないが、CPU は単独で動作可能である。このように、GPU と CPU にはそれぞれ特徴があるため、

© 2010 Information Processing Society of Japan

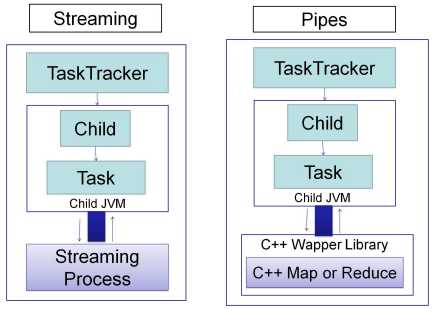

図 2 Hadoop Streaming と Hadoop Pipes
Fig. 2 Hadoop Straming and Hadoop Pipes

両者を使い分け、活用することにより、効率よく計算を行うことが可能となる。

GPGPU 向けのプログラミング環境として、NVIDIA が提供する C 言語の統合開発環境である CUDA が挙げられる。CUDA は C 言語の拡張であり、抽象度が高いため、容易にプログラミングが可能である。化学計算や疎行列計算、ソート、検索、物理モデルなど、多岐に渡る分野におけるアプリケーションに対してスケーラブルな並列計算を可能にした。これらのアプリケーションでは数百個のコアと数千個のスレッドが並列にスケールする。

3. ハイブリッドタスクスケジューリング手法

CPU と GPU が混在する環境で、Map タスクを CPU と GPU に割り振る際のタスクスケジューリングを行い、高速化する手法を提案する。この章では、まず Hadoop から GPU を呼び出す方法について述べ、続いて CPU と GPU のハイブリッド実行の方法、ハイブリッド実行におけるタスクスケジューリング手法の順に述べる。

3.1 Hadoop から CUDA を呼び出す手法の比較

Hadoop 上で CPU と GPU のハイブリッド実行を行うためには、まず GPU アプリケーションを Hadoop から呼び出せるようにする必要がある。Hadoop は Java で実装されており、アプリケーションも通常 Java で実装されるため、GPU を呼び出す手法が必要となる。Hadoop から GPU を呼び出す手法には、Hadoop Streaming、Hadoop Pipes、JNI、jCUDA などが存在する。以下ではそれぞれの特徴について述べる。

- **Hadoop Streaming**
  Hadoop Streaming は Hadoop とユーザプログラムの間のインターフェイスに Unix 標準ストリームを用いている (**図 2**)。そのため、MapReduce プログラムから標準入出力に読み書きできる言語であれば任意のものを使用することができる。ユーザは任意の実行可能バイナリやスクリプトを Mapper や Reducer として利用することができる。
- **Hadoop Pipes**
  Hadoop Pipes は Hadoop MapReduce の C++インターフェイスである。標準入出力を用いて Map と Reduce のコードと通信する Streaming とは異なり、Pipes はソケット接続を行い、TaskTracker が C++の Map 関数あるいは Reduce 関数で実行されている処理と通信を行うチャネルを作る。JNI（後述）は使用しない。
- **JNI**
  JNI(Java Native Interface) はネイティブプログラミングインタフェースであり、これを使用することにより、JVM で実行される Java コードが C、C++、アセンブリ言語など他のプログラミング言語で書かれたアプリケーションやライブラリと相互運用できるようになる。JVM で動作させるには処理速度の面で不利とされる計算量の多いプログラムを部分的にネイティブコードに置き換えて高速化したり、標準クラスライブラリからはアクセスできないオペレーティングシステムの機能を利用するプログラムを通常の Java クラスのように呼び出すことが可能となる。
- **jCUDA**
  jCUDA(Java for CUDA)[9] は CUDA のホスト API の Java バインディングを行うことにより、Java ベースのアプリケーションから CUDA を呼び出し、GPU を利用することができる。jCUDA は CUDA プログラミングのためのオブジェクトモデルである。倍精度の計算が可能であり、CUDA2.1 ドライバ API と CUDA2.1 ランタイム API に対応している。また、CUFFT ルーチンをサポートや OpenGL との相互運用性があり、CUBLAS ルーチンのサポートも行われる予定である。

以上のように、Hadoop から CUDA を呼び出すにはいくつかの方法がある。Hadoop Streaming では任意の言語で書かれたバイナリを利用できるが、標準入出力を解析してデータをやり取りを行う必要がある。一方、Hadoop Pipes では Hadoop ランタイムはオブジェクトや key などの抽象と通信することができるため、標準入出力を解析する必要がない。JNI では、メソッドやフィールドのような Java データ構造は JNI インターフェースを通じた間接的な関数呼び出しを必要とするため、メソッドが Java オブジェクトフィールドを扱う必要がある場合、複数の関数呼び出しのためにかなりのオーバーヘッドを被ってしまう。また、プログラムを動かすためには実行環境に依存したライブラリを用意しなければならず、

© 2010 Information Processing Society of Japan

Java の利点である可搬性が失われてしまう。jCUDA では CUDA2.1 までしかサポートされていないため、CUDA2.2 以上で実行できるかは明らかではない。さらに、jCUDA はホスト関数しかラップしていないため、コンパイルされた CUDA バイナリファイルをロードするためのモジュール関数を使用する必要がある。以上のことから、今回は Hadoop Pipes を対象にした。

3.2 CPU と GPU による MapRecuce のハイブリッド実行

MapReduce に GPU を適用するため、各 Map タスクを CPU と GPU に割り振り、ハイブリッド実行を行う手法を提案する。アプリケーションバイナリは CPU 用のものと GPU 用のものをそれぞれ用意し、双方をジョブに投入する。各 Map タスクは CPU・GPU どちらで実行されたか、また GPU で実行された場合のデバイス番号を識別することができ、スケジューラは各スレーブノードから CPU・GPU 上での Map タスクの使用状況を確認することによって管理する。タスクスケジューラが適宜 CPU バイナリと GPU バイナリを各 Map タスクに振り分ける。この際、CPU バイナリを渡された Map タスクは CPU 上で実行し、GPU バイナリの場合は GPU 上で実行する。その後、Reduce フェーズでは通常通りに各 Map タスクの出力結果をもとに Reduce を行う。双方での Map タスクの出力の形式は同じであるため、GPU 上で Map タスクを実行しても Reduce タスクには影響を与えない。

3.3 スケジューリングの方法

CPU と GPU の混在する不均質環境におけるハイブリッドタスクスケジューリング手法を提案する。基本的な考え方は、CPU と GPU の性能比に比例させて Map タスクを割り当てることにより、ジョブ実行時間の短縮を図るというものである。

スケジューリング手法には大きく分けて、静的なスケジューリングと動的なスケジューリングがある。静的なスケジューリングは CPU と GPU の性能比があらかじめ把握できていれば可能であり、実行時にプロファイリングをする必要がない。しかし、アプリケーションによって CPU と GPU の相対性能は大きく異なることがわかっている[10] ため、静的な方法は典型的にはこのような不均質な環境には適していない。一方、動的なスケジューリングでは、各 Map タスクが終了する度に CPU と GPU それぞれの、実行が終わった Map タスク全体の平均実行時間を繰り返し計算してプロファイルを取得することにより、不均質な環境に適応することが可能となる。CPU と GPU の性能は計算ノード内の CPU・GPU の種類や数などの計算環境や、アプリケーションの性質に依存するため、CPU と GPU を同時実行し、実行時のプロファイルを動的に取得することにより、より速いほうにより多くの資源を割り当てるようにする。

3.4 スケジューリングのアルゴリズム

上記の方法で得たプロファイルから Map タスクを CPU と GPU に割り振るスケジューリングについて述べる。CPU と GPU の混在環境でのジョブ実行時間の最小化のためのモデルを導入する。パラメータは Map タスク数を N、CPU コア数を n、GPU 台数を m、CPU と GPU の性能比を a、1 つの GPU 上で実行された Map タスクの実行時間を t とする。以上のパラメータに基づき、ジョブの実行時間の最小化を行うための定式化を行う。なお、各 Map タスクは 1 つの CPU コアまたは 1 つの GPU 上で実行される。CPU と GPU の性能比 a(以下加速倍率と呼ぶ) は以下の式で算出する。

$$a = \frac{mean\ map\ task\ time\ run\ on\ CPU}{mean\ map\ task\ time\ run\ on\ GPU}$$

1 つの GPU 上での Map タスクの実行時間を t とする。CPU の実行時間は加速倍率により at となる。x を CPU 上で実行する Map タスク数、y を CPU 上で実行する Map タスク数とする。以上のパラメータに基づき、ジョブ全体、すなわち全 Map タスクの実行にかかる時間を目的関数として定式化すると、以下のようになる。

$$minimize\ f(x,y)$$

$$subject\ to\ f(x,y) = max\{\frac{x}{n}at, \frac{y}{m}t\}$$

$$x + y = N$$

$$x, y \geq 0$$

- 目的関数: CPU 上で x 個、GPU 上で y 個の Map タスク実行が共に終了するまでにかかる時間
- 制約式: N 個の Map タスクを CPU と GPU に割り振るための条件式

上式の解に対応する x, y が CPU と GPU にそれぞれ割り振る Map タスク数となる。この計算をスケジューラがスレーブからの一定時間おきに送られるハートビートを受ける度に繰り返すことにより、常に最新の計算結果を保持できるようにする。計算結果により x が 0 となった場合には残りの Map タスクはすべて GPU に割り振り、y が 0 となった場合にはすべて CPU に割り振るようにする。このアルゴリズムを適用することにより、性能の高いプロセッサが性能の低いプロセッサの実行が終わるのを待つアイドルな時間を最小化する。

© 2010 Information Processing Society of Japan

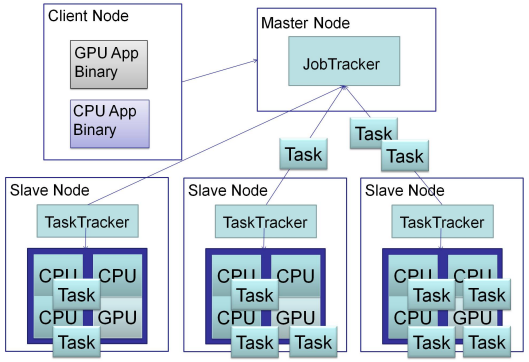

図 3 Hadoop のタスクスケジューリングの構造
Fig. 3 The structure of task scheduling on Hadoop

4. 設計と実装

提案手法を実現するため、Hadoop から CUDA を呼び出し、CPU と GPU に Map タスクを割り振る実装を行う。この章では、Hadoop から CUDA バイナリを呼び出す方法、JobTracker と TaskTracker が協調して複数 GPU に Map タスクを割り当てる方法、Map タスクのスケジューリング手法について述べる (**図 3**)。

4.1 Hadoop からの GPU の呼び出し

前章で述べたように、Hadoop から他の言語を呼び出すにはいくつかの方法があり、それぞれに異なる特徴を持つ。今回は CUDA の呼び出しを対象としている。CUDA は C または C++の拡張であるため、C++との親和性を考慮して Hadoop Pipes を使用する。Hadoop Pipes はソケット上で C++プロセスに対しその環境のポート番号を渡すことにより、C++プロセスが親の Java Pipes タスクに対して永続的なソケット接続を確立する。タスクの実行中、Java プロセスは入力の key-value ペアを外部のプロセスに渡し、外部のプロセスはユーザが定義した Map 関数または Reduce 関数を起動し、出力の key-value ペアを Java プロセスに渡す。TaskTracker 側からみると、TaskTracker の子プロセス自身が Map または Reduce コードを走らせているように見える。Hadoop におけるタスクの呼び出しは次のように行われる: (1)MapReduce プログラムがジョブを立ち上げ、JobClient を起動する。(2)JobClient は JobTracker に対してジョブを投入する。(3)JobTracker が TaskTracker に対して Map タスクないし Reduce タスクを割り当てる。(4)TaskTracker は Child JVM を立ち上げ、その中で Map タスクないし Reduce タスクを実行する。Pipes を使用する場合は、Child JVM が実行する Map タスクや Reduce タスクが、C++で書かれた Map クラスや Reduce クラスと互いに key-value の入出力を行うことによって通信を行う（図 2)。CPU と GPU を同時に呼び出す必要があるため、C++の Map タスクに CPU 用と GPU 用のものをそれぞれ用意し、Child JVM が双方とも通信できるようにした。ジョブの実行時に GPU プログラムとプログラムをそれぞれ引数として指定し、Pipes のマスターが双方を読み込むことによって実現した。タスクスケジューラは、CPU で Map タスクを実行したい場合は CPU バイナリを Map タスクに割り当て、GPU で実行したい場合は GPU バイナリを割り当てるようにコントロールする。

ジョブの実行時にユーザが CPU・GPU のバイナリを指定すると CPU と GPU に Map タスクを割り振る機能を追加した。これには CPU・GPU のバイナリの指定、CPU・GPU への Map タスクの割り振りの二つのステップがある。CPU・GPU のバイナリの指定について述べる。まず、ユーザは CPU・GPU のアプリケーションバイナリをそれぞれ指定してジョブを投入する。続いてクライアントは JobTracker にジョブを渡す。TaskTracker は JobTracker に対して一定時間ごとにハートビートを送ることにより、行うべき Map タスクがないか尋ねる。この際、TaskTracker は CPU・GPU が空いているかを知らせる変数を持ち、これも合わせて JobTracker に伝える。またスケジューリングを行うためのパラメータとして、終了した Map タスクの実行時間や TaskTracker が管理する DataNode の CPU と GPU の数などの情報を合わせて伝える。続いて、JobTracker は TaskTracker によって渡された Map タスクの情報を受け取る。CPU・GPU のどちらで実行したか、既に実行を終えた Map タスクの実行時間などの情報などから次の Map タスクを CPU・GPU のどちらで実行するかを前章で扱ったスケジューリングアルゴリズムによって計算する。計算結果に基づき、CPU・GPU のどちらで実行させるかを判断し、その情報を TaskTracker に伝える。TaskTracker は JobTracker から CPU 上で Map タスクを実行するよう指示された場合は CPU 上で、GPU 上で実行するよう指示された場合は GPU 上でそれぞれ実行する。

複数 GPU を使用する場合には、アプリケーションに GPU デバイスの識別番号を伝える必要がある。各 Map タスクは GPU デバイスの識別番号の配列を保持し、デバイスの使用状況の情報を格納する。TaskTracker はハートビートの際に GPU デバイスの情報を伝える。JobTracker は空いている番号から順に Map タスクを割り当て、Map タスクに割り当てた番号のデバイスが使用中である旨の情報を保持させる。

© 2010 Information Processing Society of Japan

4.2 Hadoop 上での GPU との Map タスクの割り振り

JobTracker は CPU スロットまたは GPU スロットに空きがないかを調べ、空きがある場合には TaskTracker に Map タスクを割り振る。TaskTracker はノード内のタスクの管理をしており、タスクの状態が実行中であるか、実行が完了しているか、実行に失敗しているかなどの情報を持っている。JobTracker と TaskTracker は一定時間ごとに互いにハートビートによる通信を行い、ジョブとタスクの情報を交換する。

これら JobTracker、TaskTracker に対し、提案スケジューリング手法を導入する。TaskTracker はノード内の Map タスクの実行が終了するごとに新しい Map タスクの要求、各 Map タスクの実行時間のデータの管理を行う。JobTracker は TaskTracker の要求に対し、スケジューリングの計算を行い、計算結果に従って各 TaskTracker に Map タスクを割り当てる。スケジューリングの計算を行うため、Map タスクから渡された情報をもとに、CPU・GPU それぞれの平均実行時間、CPU に対する GPU の加速倍率を算出する。実行の始めは性能比が未知であるため、空いているすべての CPU コアと GPU に Map タスクを割り振る。各 Map タスクの実行が終了するごとに、CPU・GPU の平均実行時間、加速倍率を再計算し、CPU・GPU のスロット数と残りの Map タスクから先述のアルゴリズムにより割り振りを計算し、計算結果に従ってスケジューリングを行う。この際、JobTracker のスケジューラが TaskTracker の Map タスク実行状況を確認し、CPU・GPU に空きがあるかどうか確認する。空きがない場合は空きができるまで待ってから割り振るようにする。これにより、性能の高いプロセッサが性能の低いプロセッサの実行を待つアイドルな時間を最小化する。

5. 実　　験

CPU と GPU のハイブリッド実装、およびタスクスケジューリングアルゴリズムの改良による性能の変化を調べるため、実際のアプリケーションをハイブリッド実行することにより、ジョブ実行時間、および Map タスク実行時間の比較を行った。

5.1 概　　要

CPU と GPU のハイブリッドスケジューリングの必要性を示すため、CPU と GPU のハイブリッド実行およびタスクスケジューリングアルゴリズムの適用による効果を測定する。実アプリケーションを用いてハイブリッド実行時のジョブ実行時間および Map タスク実行時間の測定を行う。

アプリケーションには K-Means を用いる。Map フェーズの GPU の活用による効果を見るため、各アプリケーションの計算は Map フェーズでは各データセットに対して K-Means の計算を行い、Reduce フェーズで各 Map タスクの計算結果を集約する。K-Means はクラスタリングの計算手法として最もよく利用されるアプローチの一つである。K-Means では以下の計算を行う：(1)k 個のランダムなクラスタを選ぶ。(2) 各データ（点）に対し、最も近いクラスタを見つけ、この点を対応するクラスタに割り当てる。(3) 各クラスタに割り当てられたデータの平均を取ることによって新しい k 個のクラスタを再計算する。(4) クラスタの位置が固定されるまで以上の操作を繰り返す。1 セットのクラスタの数 k は 128、データの数は 2 次元の点を 262144 個とし、これを 4000 セット集めた 20GB のファイルを使用した。

表 1　実験環境

	CPU	GPU
デバイス	AMD Opteron(Dual Core)	Tesla S1070
周波数	2.4GHz	1.296-1.44GHzGHz
メモリ	1.0GB	16GB

実験は、東京工業大学のスーパーコンピュータである TSUBAME 上で、GPU を搭載したノードを 1〜64 ノードと変化させて行った。分散ファイルシステムには Lustre を用いた。ストライプ数は 4 とした。I/O 性能は 32MB のサイズのファイルを対象にした場合で、write で 180MB/s、read で 610MB/s であった。各ノードの CPU・GPU は**図 1** のようになっている。1 ノードの CPU のコア数は 16 個、GPU の台数は 2 台である。GPU 上で実行する Map タスクは計算部分は GPU で実行するが、それ以外の部分は CPU 上で実行する必要があるため、各 GPU は 1CPU コアを占有する。そのため、1GPU を使用する場合は各ノードは 15 個の CPU コアと 1 台の GPU で同時に Map タスクを実行でき、2GPU を使用する場合は 14 個の CPU コアと 2 台の GPU で同時実行できる。各 Map タスクのサイズは 32MB とした。また Reduce の数は 1 ノードにつき 16 個とした。ただし、64 ノードの場合はヒープサイズの上限を考慮し、1 ノードにつき 15 個とした。

5.2 実 験 結 果

実験結果は**図 4** のようになった。Map タスク実行時間において CPU に対する GPU の加速倍率が 1.0-1.25 倍となる高速化を達成した。またジョブ実行時間において、CPU のみの使用に対し、2GPU の使用とスケジューリングアルゴリズムの適用をした場合に 1.02-1.93 倍の高速化を達成した。複数 GPU の利用による効果として、15CPU と 1GPU の場合の実行時間を 14CPU と 2GPU の場合の実行時間で割った値が、スケジューリングアルゴリズ

© 2010 Information Processing Society of Japan

ムを適用した場合に平均 1.29 倍、適用しない場合に平均 1.02 倍となった。

ノード数の増加によってスケールしていることを確認した。しかし、64 ノードのときは 32 ノードのときよりも実行時間が増加している。これは、Map タスク数が入力データサイズ 20GB をチャンクサイズの 32MB で割った数である 619 個であるのに対し、スロット数であるノード数 64 に 1 ノードのスロット数 16 をかけた値の方が大きくなるため、始めにすべての Map タスクを割り振ってもアイドルなスロットが存在するため、I/O や通信などによるオーバーヘッドの分だけ実行時間が増加したと考えられる。

上記の結果から、GPU の利用、およびタスクスケジューリングアルゴリズムの改良による MapReduce アプリケーションの性能向上を確認した。また、複数 GPU の利用により、1GPU の場合と比べ更なる性能向上を達成したことを確認した。タスクスケジューリングアルゴリズムの改良による性能向上の要因には以下が挙げられる。スケジューリングを改良しない場合は、空いている CPU・GPU スロットがあれば必ず Map タスクを割り振るため、性能が高いプロセッサが性能の低いプロセッサの実行の終了を待つアイドルな時間が生じる。一方、提案手法では CPU・GPU に割り振るタスク数を先述のアルゴリズムによってコントロールし、アイドルな時間を最小化している。性能向上はこのアイドルな時間を無くしたことにより達成されている。

6. 関連研究

CPU と GPU の混在する不均質環境でリダクション計算を同時実行するシステムがある[10]。各 CPU と GPU に割り振るチャンクサイズを変化させることによる実行時間の変化を調べているが、チャンクサイズは各ジョブごとに固定されており、動的に CPU と GPU に割り振るタスクの割合をスケジューリングすることは考慮していない。また複数ノードにおける実行についても考慮していない。

学習メカニズムによって CPU と GPU が混在する不均質環境でのデータの割り振りを最適化する研究がある[11]。これは CPU と GPU に対しチャンクをバランス良く割り当てる方法という点では類似しているが、事前情報なしに実行時にプロファイルを取りながらスケジューリングするわけではない。我々の研究では動的なスケジューリングを行うため学習を行う必要がない。

別のタスクスケジューリングの方法として、タスクを投機的に複数のノードで実行することによって不均質環境での計算効率を向上させる手法がある[12]。しかし、CPU と GPU が混在する不均質な環境を想定しており、このような環境では、データ転送時のオーバーヘッドや、ノード間の通信遅延などの問題が発生しうるため、一般にスケジューリング手法は異なる。また、この研究では異常タスクに対し投機実行を行うというものであるが、我々は異常タスクに対処するというよりはむしろ CPU と GPU のパフォーマンスを比較して高速化を図ろうとしている。

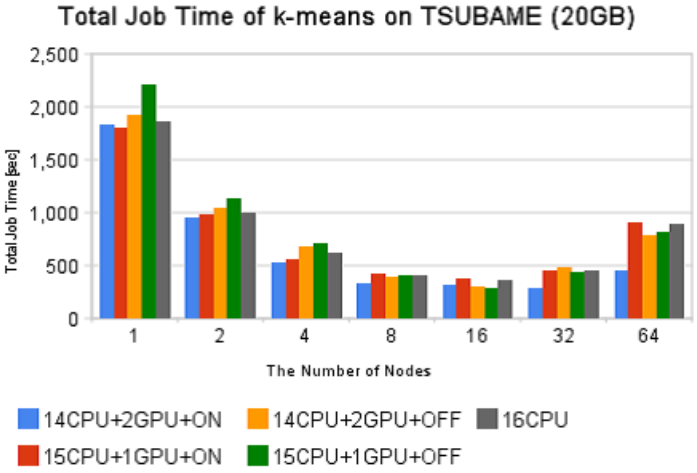

図 4 TSUBAME 上での K-Means アプリケーションのジョブ実行時間
Fig. 4 Total Job Time of K-Means on TSUBAME

不均質な計算環境のためのタスクスケジューリング手法に関する研究は古くから行われている[13]。現状では従来手法と類似しているが、よりタスクの詳細な挙動の分析を行うことにより、メモリバンド幅、ディスクアクセス時間を考慮したスケジューリングを行う予定である。

7. おわりに

CPU と GPU の混在環境を考慮し、MapReduce 実装の Hadoop 環境において、Map タスクを GPU から呼び出す機能、および CPU と GPU による Map フェーズのハイブリッド実行を実現した。また CPU と GPU の混在環境を考慮したタスクスケジューリングモデルを構築し、ジョブ実行時間の最小化のモデル化、およびその実装を行った。K-Means アプリケーションで実験を行った結果、Map タスク実行時間において CPU に対する GPU の加

© 2010 Information Processing Society of Japan

速倍率が 1.0-1.25 倍となる高速化を達成した。またジョブ実行時間において、CPU のみの使用に対し、2GPU の使用とスケジューリングアルゴリズムの適用をした場合、1.02-1.93 倍の高速化を達成した。

今後の課題としては、メモリ容量、バンド幅、ディスクアクセス時間、マスター・スレーブ間の通信時間などのモニタリングによる詳細なプロファイルや Map タスクの実行時間の内訳の取得を行い、これらを考慮してスケジューリングモデルの改良を行う予定である。

謝辞 本研究の一部は科学研究費補助金 特定領域研究 「情報爆発時代に対応する高度にスケーラブルな高性能自律構成実行基盤」（課題番号 18049028）および JST CREST「ULP-HPC:次世代テクノロジのモデル化・最適化による超低消費電力ハイパフォーマンスコンピューティング」の援助による。

参 考 文 献

1) Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI '04, Sixth Symposium on Operating System Design and Implementation*, pp.137–150 (2004).
2) D.Owens, J., Houston, M., Luebke, D., Green, S., E.Stone, J. and C.Phillips, J.: GPU Computing, *Proc IEEE*, Vol.96, No.5, pp.879–899 (2008).
3) John, N., Ian, B., Michael, G. and Kevin, S.: Scalable Parallel Programming with CUDA, *Queue*, Vol.6, No.2, pp.40–53 (2008).
4) K., J.A. and C., D.R.: *Algorithms for clustering data*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1988).
5) Hong-tao, B., Li-li, H., Dan-tong, O., Zhan-shan, L. and He, L.: K-Means on Commodity GPUs with CUDA, *Computer Science and Information Engineering, 2009 WRI World Congress*, pp.651–655 (2009).
6) Bialecki, A., Cordova, M., Cutting, D. and O'Malley, O.: Hadoop: a framework for running applications on large clusters built of commodity hardware (2005).
7) Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems, *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA)* (2007).
8) He, B., Fang, W., Luo, Q., K.Govindaraju, N. and Wang, T.: Mars: A MapReduce Framework on Graphics Processors, *Parallel Architectures and Compilation Techniques*, pp.260–269 (2008).
9) Company for Advanced Supercomputing Solutions Ltd.: jCUDA, http://hoopoe-cloud.com/Solutions/jCUDA/Default.aspx".
10) Vignesh, T.R., Wenjing, M., David, C. and Gagan, A.: Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations, *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, New York, NY, USA, ACM, pp.137–146 (2010).
11) Lu, C.-K., Hong, S. and Kim, H.: Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping, *MICRO '09*, pp.45–55 (2009).
12) Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R. and Stoica, I.: Improving MapReduce Performance in Heterogeneous Environments, Technical report, EECS Department, University of California, Berkeley (2008).
13) 須田礼仁：ヘテロ計算環境のためのタスクスケジューリング手法のサーベイ，情報処理学会論文誌コンピューティングシステム， Vol.47, No.SIG$_1$8($ACS_1$6), *pp*.92−−114(2006).

© 2010 Information Processing Society of Japan