社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

マルチコアおよびGPUを用いたグラフ型データベースの性能評価

森島　　信† 　松谷　宏紀†,††,†††

† 慶應義塾大学 理工学部 〒 223-8522 神奈川県横浜市港北区日吉 3-14-1
†† 科学技術新興機構さきがけ
††† 国立情報学研究所
E-mail: †{morisima,matutani}@arc.ics.keio.ac.jp

あらまし　グラフ型データベースは、データをグラフ形式で蓄積し処理するデータベースである。グラフ型データベースはノード間の関係性を表現するのに向いているため、SNS(Social Networking Service) やソーシャルグラフを基にしたリコメンデーションエンジンへの応用が期待されている。グラフ型データベースで最も計算量が多い処理は、グラフの探索である。本論文では、グラフ探索処理の並列化と GPU による高速化を実現し、その性能を評価する。ここではグラフ型データベース Neo4j を対象に、Dijkstra 法や A*法を高速化する。評価では、Facebook の次数分布を基にした 100,000 ノードのグラフに対して探索を行ったときの計算時間を測定した。データ構造を変える際のオーバーヘッドを含めない場合は、オリジナルの Neo4j と比べて、8 ノードで並列化すると、Dijkstra 法において 16.2 倍、A*法において 13.8 倍高速化できた。一方、GPU では、Dijkstra 法において 26.2 倍、A*法において 32.8 倍高速化できた。オーバーヘッドを考慮しても、この性能向上は有意である。
キーワード　構造型ストレージ、グラフ型データベース、GPU、マルチコア

Performance Evaluation of Graph Database using Multicore and GPU

Shin MORISHIMA† and Hiroki MATSUTANI†,††,†††

† Faculty of Science and Technology, Keio University 3-14-1, Hiyoshi, Yokohama, JAPAN 223-8522
†† PRESTO, Japan Science and Technology Agency
††† National Institute of Informatics
E-mail: †{morisima,matutani}@arc.ics.keio.ac.jp

Abstract　Graph databases use graph structures to store data sets as nodes, edges, and properties. They are used to store and search the relationships between a large number of nodes, such as social networking services and recommendation engines that use customer social graphs. Since graph search queries typically require high computation power, in this paper, we accelerate the graph search functions (Dijkstra and A* algorithms) of a graph database Neo4j using multicore and graphics proceccing units (GPUs). We use 100,000-node graphs generated based on a degree distribution of Facebook social graph for evaluations. Although our initial evaluation results do not include overhead of building extra data structures, the results show that, compared to the original Neo4j, the 8-core parallelized version improves the Dijkstra and A* search performance by 16.2x and 13.8x, respectively. The GPU based implementation improves the Dijkstra and A* search performance by 26.2x and 32.8x, respectively.
Key words　Structured storage, graph database, GPUs, multicore

1. はじめに

今日、情報通信技術と各種センシング技術の発展にともない、膨大な情報 (ビッグデータ) が生成されている。これらの情報は主にデータベースとして蓄積されるが、現在最もよく使われているデータベースシステムである RDBMS(Relational DataBase Management System) は、汎用性が高く豊富な機能を持つものの、スケーラビリティはそれほど高くない。そのため、構造型ストレージと呼ばれる、一つの機能のみに特化しているが、スケーラビリティに優れるデータベースが近年注目されている。

本論文で扱うグラフ型データベースは構造型ストレージの一種であり、データをグラフ形式で蓄積し処理するデータベースである。このデータ構造は、ノード間の関係性を表現するのに

向いており、SNS(Social Networking Service) やソーシャルグラフを基にしたリコメンデーションエンジンへの応用が期待されている。グラフ型データベースで最も計算量が多く、ボトルネックとなる処理はグラフ探索処理である。そこで、グラフ型データベースのスケーラビリティを向上させるために、マルチコアと GPU でのグラフ型データベースの高速化を実現し、その評価を行う。本論文では、グラフ型データベースの一つである Neo4j を用いて、Neo4j に実装されているグラフ探索処理のアルゴリズムのうち、実用性の高い Dijkstra 法と A*法の二つのアルゴリズムを高速化する。具体的には、OpenMP 互換ライブラリによる並列化と CUDA による高速化を行い、性能を評価する。

本論文の構成は以下の通りである。2 章は本論文の関連研究を述べる。3 章は対象とするグラフ型データベースの詳細を述べる。4 章はマルチコアおよび GPU を用いたグラフ探索処理の実装を述べる。5 章は計算時間とオーバーヘッドの評価を述べる。6 章で本論文をまとめる。

2. 関連研究

構造型ストレージの一種であるキーバリューストア型データベースの GPU での実装および評価を Hertherington らが行っている [1]。CPU-GPU 間のデータ転送を除いた場合、33 倍の性能向上を得られているが、データ転送のオーバーヘッドを考慮すると、性能向上は大きくない。対して、グラフ型データベースの場合は、キーバリューストア型と比べてデータ量に対する計算量の割合が非常に大きいため、データ転送のオーバーヘッドは小さくなる。

各種のグラフ探索アルゴリズムが GPU を用いて高速化できることは、数々の研究によって示されている。Ortega-Arranz らは Dijkstra 法の高速化を GPU で行い、CPU に対して、最大で 220 倍の性能向上に成功している [2]。Merrill らはグラフ探索アルゴリズムの一つである幅優先探索の高速化を GPU で行い、CPU に対して最大 29 倍の性能向上に成功している [3]。また、Nobart らはグラフの最小スパニングツリーを求めるアルゴリズムの高速化を GPU で行い、CPU に対して最大 14 倍の性能向上に成功している [4]。本論文では、これらの利得がグラフ型データベースに応用しても得られることを示す。

3. グラフ型データベース

本論文で、高速化の対象とするグラフ型データベースは Neo4j である。Neo4j は Java 言語で実装されたオープンソースかつ代表的なグラフ型データベースの 1 つである。3.1 節から 3.5 節では Neo4j に実装されているグラフ探索アルゴリズムについて説明する。

3.1 Dijkstra 法

Dijkstra 法は、重み付きグラフにおいてある始点からの最短経路を求めるアルゴリズムである。始点から重みの小さい順に探索を行い、最短経路が判明しているノードを増やしていくことで全てのノードへの最短経路を求める。一般的な Dijkstra 法は始点から全てのノードへの最短経路を求めるが、Neo4j では、始点と終点を与え、その 2 点間の最短経路を求める仕様となっている。これは、Dijkstra 法を最後まで実行せず、終点までの距離が求まった時点で探索を終了することで実現する。

3.2 A*法

A*法は、Dijkstra 法を改良したアルゴリズムである。それぞれのノードに目的ノードまでの重みの推定値を与え、その推定値が小さい順に探索を行う。

3.3 Shortest Path

Shortest Path はある 2 ノード間のホップ数が最小となる経路を全て求めるアルゴリズムである。始点からホップ数が少ない順に全ての経路を探索し、目的ノードに到達した時点で終了する。

3.4 All Path

All Path はある 2 ノード間で、設定したホップ数以下の経路を全て求めるアルゴリズムである。Shortest Path の終了条件をホップ数に変更したものと同じである。

3.5 All Simple Path

All Simple Path は All Path で求まる経路のうち、同じノードが二回現れる経路を除外したものである。

Shortest Path、 All Path、 All Simple Path は条件が異なるだけで、探索処理自体は同じである。よって、Neo4j で実装されているグラフ探索処理は実質 3 種類といえる。本論文では Dijkstra 法と A*法を扱う。

4. グラフ探索の高速化

4.1 隣接行列の作成

Neo4j は Java 言語で書かれており、ノードや辺に関する情報をまとめ、それらを様々な用途に使用できるよう、ノードや辺はクラスとして定義されている。Neo4j でのグラフ探索アルゴリズムはこれらのクラスのまま実装されているが、このようなデータ構造をそのまま並列ライブラリを用いて並列化したり、GPU 上に実装することは困難である。そのため、ノードのノード ID と辺の両端のノードの ID と辺の重みの情報をそれぞれのクラスから抽出し、その ID から隣接行列を生成し、隣接行列に対して探索を行う。この場合、通常の隣接行列に対する探索をマルチスレッドと GPU で実装すればよく、直接実装する場合よりも単純化できる。

隣接行列はメモリ使用量を抑え、かつ隣接行列へのアクセス数を減らして高速化するために、通常の隣接行列ではなく、配列として実装する。図 1 はグラフで表現されたデータから配列としてデータを抽出した様子を示している。左側のグラフのノードの中の数字がノード ID、辺の横の数字が辺の重みを表している。配列は、図 1 に示すように、辺の端点の数と同じ要素数の配列 2 つとノード数と同じ要素数の配列 1 つを用意する。配列 1 には、それぞれのノードから出る辺の目的ノードを入れ、配列 2 には配列 1 に対応する辺の重みを入れる。配列 3 はその要素に対応するノード ID のノードから出る辺が配列 1、配列 2 の何番目からに相当するかを入れ、ポインタとして用いる。例えば、図 1 においてノード 3 の 2 番目のノードの重みを知りたい場合は、配列 3 の 3 番目の要素が指している場所から 2 番目の配列 2 を参照すればよく、結果は 4 となる。

この配列に対して探索アルゴリズムを実装し、その結果を Neo4j に返し、Neo4j で処理することで、Neo4j で直接実行した場合と同じ出力形式で結果が得られる。

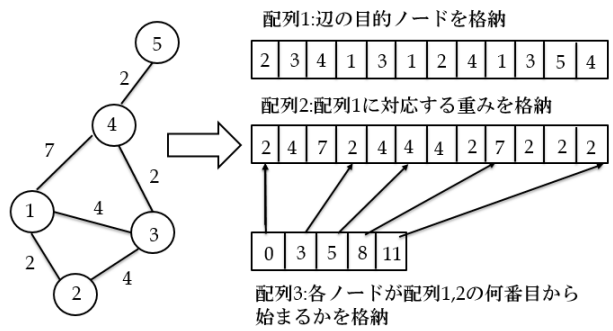


図 1　隣接行列の配列への実装

4.2　JOMP を用いた並列化

マルチスレッドでの並列化は、JOMP と呼ばれる Java を OpenMP の構文で並列化することができる API を使用する [5]。これを用いることで、マルチスレッドによるデータの競合が発生しなければ、容易にマルチコアでの実装が可能である。Dijkstra 法では、データの競合が発生する箇所はないため、そのままマルチスレッド化ができる。

しかし、A*法では、計算対象のノードと計算が終了したノードを保存するリストにおいて競合が発生する。通常は、この二つのリストは可変長配列やリスト構造で生成するが、複数のスレッドでリストに追加や削除を行う時に競合が発生する。競合を回避するため、ノード数と同じ要素数の配列を用いる。あるノードがリストに入っている場合そのノード ID の要素を 1 に、入っていない場合 0 とすることで、この二つのリストを実装する。A*法では、1 つのノードを複数のスレッドで同時にリストに追加、削除することはないので、この方法で競合は回避できる。また、リストのアクセスを高速化できる。メモリ使用量は増加するが、隣接行列の要素数の方が多く、このリストのメモリ使用量の全体に占める割合は少ない。

4.3　CUDA を用いた並列化

GPU での実装は、jcuda という CUDA を Java から呼び出す API を使用する [6]。

GPU における実装において、重視すべきことは、GPU が持つ多数のコアを全て活用するように設計することである。そのためにスレッド数をコア数に比べて多くするのが一般的である。しかし、この実装ではスレッド数を CUDA の 1 ブロックで扱える最大のスレッド数 1,024 とする。これは、コア数と同程度の数である。このように少ないスレッド数で実装する理由は、ブロック間で同期をとる場合一度カーネルを終了し、もう一度カーネルを起動させるグローバル同期が必要だからである。

今回扱うアルゴリズムの Dijkstra 法と A*法はいずれも、次に探索するノードを求めるために最小値を求める演算、アルゴリズムの終了判定、グラフの探索によるコストの更新の 3 つのステップを終了までループさせることで行う。このうち、ボトルネックとなる部分は 1 つ目の最小値を求める演算である。

最小値を求める演算はリダクション演算の一つで、CUDA では、要素数と同数のスレッド数として、リダクション演算を行うカーネルを複数回再帰的に呼び出すことで実装するのが一般的であり、この方法のほうが 1,024 スレッドで最小値を求めるよりも高速である。図 2 は一般的な方法で最小値を求めるために要素数をスレッド数として実装した場合とスレッド数 1,024 として実装した場合のそれぞれのフローチャートである。

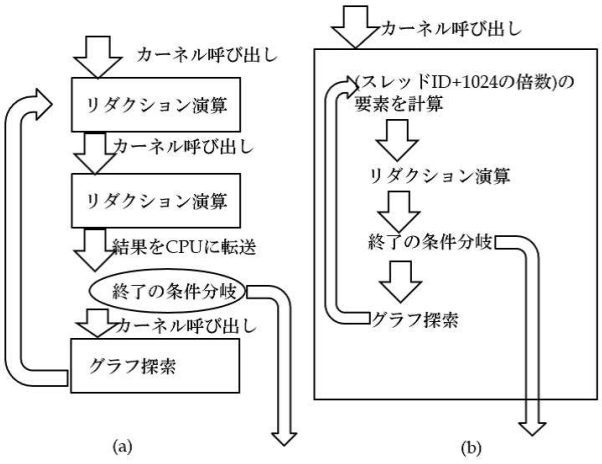


図 2　隣接行列の配列への実装 (a) スレッド数=要素数の場合 (b) 要素数=1 ブロックで扱える最大数の場合

一般的なリダクション演算は、結果が求まるまで再帰的にリダクション演算を行うが、図 2 は 2 回で結果が求まる場合である。図の四角の中がカーネルで処理する演算、楕円の中が CPU で行う演算である。図 2(a) では、まず、最小値が求まるまで 2 回カーネルを呼び出してリダクション演算を行う。次に、ループが CPU で実行されており、終了の条件分岐も CPU で行わなければならないため、求めた最小値を CPU に転送し、終了の条件分岐を行う。終了でなければ、グラフの探索をしてコストを更新する、という流れをループで繰り返す。

対して、図 2(b) では、1,024 以上の要素数のリダクション演算は行えないため、始めにスレッド ID+1,024 の倍数の要素について最小値を求め、要素数を 1,024 にしてからリダクション演算を行う。グローバル同期が必要ないため、ループもカーネル内で実行でき、終了の条件分岐、グラフ探索も同じカーネル内で行う。

図 2(a) では、ループ 1 回につき、3 回のカーネル呼び出しと 1 回の CPU への転送が必要となる。対して、図 2(b) では、アルゴリズムの始めにカーネルを呼び出すだけである。カーネル呼び出しと CPU への転送のオーバーヘッドが、リダクション演算の高速化による利得よりも大きいため、スレッド数は 1,024 とする方がアルゴリズム全体では高速になる。

スレッド数を 1,024 とした場合、GPU の多数のコア全てを利用することはできない。しかし、Kepler アーキテクチャの GPU では、使っていないコアがある場合、異なるストリームとして複数のカーネル実行することで、カーネル並列に実行することができる。グラフ型データベースでは、1 つの対象グラフに複数回の探索をすることが想定されるため、複数のカーネルを並列に実行することで、レイテンシを変えずに、スループットを向上させることができる。1 つのカーネルでは、全てのコアを利用できないが、複数のカーネルを実行することで、全てのコアを活用することができる。

5. 評　　価

5.1 対象グラフ

対象のグラフは、平均次数を決めてランダムに生成したランダムグラフと実在するデータに近いグラフとして、Facebookの次数分布に基づくグラフの二種類とした。

5.1.1 ランダムグラフ

ランダムグラフは、平均次数を決め、その次数に達するまでランダムに2つのノードを選び、重み付き双方向リンクで結ぶことで生成した。次数、ノード数と計算時間の関係を評価するために、平均次数10と100の場合でノード数を変化させ、それぞれ評価に用いた。

5.1.2 Facebookの次数分布に基づくグラフ

文献[7]を基に、Facebookの次数分布に近似した次数分布を持つグラフを作成し、ノード数を変化させて評価に用いた。次数の中央値は99、平均値は197である

5.2 評価環境

Neo4j、マルチコアでの評価に用いたCPUは、AMD Opteron 4238で動作周波数は3.3GHzである。マルチコアでのコア数は全ての評価で8とした。GPUの評価では、ローエンドのGPUとして、NVIDIA Quadro K600を用い、ハイエンドのGPUとして、NVIDIA GeForce GTX 780 Tiを用いた。それぞれのGPUの主な性能は表1に示す。

表1　GPUの主な性能

性能項目	Quadro K600	GeForce GTX 780 Ti
コア数	192	2,880
コアクロック	875MHz	875MHz
メモリクロック	900MHz	1,750MHz
メモリバス幅	128bit	384bit
メモリバンド幅	29GB/s	336GB/s

5.3 計算時間

対象グラフとして想定するSNS等のノード数である数百万から数億ノードの探索をグラフ全体に対して行うことは現実的ではないため、探索するノードを条件等を指定することで絞り込み、その対象グラフを探索することを想定して、10,000ノード、50,000ノード、100,000ノードの3種類のノード数のグラフを探索したときの計算時間を測定した。Neo4j以外での探索は、隣接行列を作成するオーバーヘッドが存在するが、この節ではオーバーヘッドは無視し、オーバヘッドに関しては5.4節で議論する。

5.3.1 Dijkstra法

Neo4jの仕様に合わせて、一般的なDijkstra法ではなく、始点と終点を与えて、2点間の最短経路を求める。この評価では、ランダムに始点と終点を選び、実行したときの実行時間を用いた。

図3にNeo4j、マルチコア、GeForce 780 Tiのそれぞれで Dijkstra法を1回実行したときの平均実行時間を示す。Neo4jの実行時間は平均次数が増えると、大幅に増加するのに対して、マルチコア、GPUの実行時間は平均次数が増加しても、実行時間はほぼ一定である。この違いは、それぞれの実装のデータ構造の違いによるものである。Neo4jは疎グラフに特化しているが、グラフ型データベースでは、様々な種類のグラフを扱うため、次数によって計算時間の変化しない、スケーラビリティの高い隣接行列による実装の方が適している。実際に、今回評価に用いたFacebookに基づくグラフでは、平均次数が高く、Neo4jの実行時間は平均次数が10の場合に比べて最大28.0倍に増加した。

マルチコアでは、Neo4jが特化している疎グラフの平均次数10の50,000ノードの場合はNeo4jより性能が僅かに悪化したが、その他の場合では性能向上に成功しており、Facebookに基づく100,000ノードのグラフでは、16.2倍に性能が向上した。GeForce 780 Tiでは、全ての場合で性能向上に成功しており、Facebookに基づく100,000ノードのグラフでは、26.2倍に性能が向上した。

図4は、GPUのカーネル並列実行の効果を評価するため、Dijkstra法を100回連続で実行したときの実行時間をローエンドGPU Quadro K600とハイエンドGPU GeForce 780 Tiのそれぞれでカーネルを逐次実行、並列実行した場合で比較したものである。

Keplerアーキテクチャでは、1つのブロックにつき192個のコアが使われる。この数はQuadro K600のコア数と同じである。しかし、この場合もカーネルを複数並列実行することで、リダクションの途中等で計算に使用中のコアの数が192を下回った場合やメモリアクセスの途中に他のカーネルを実行することで、スループットが向上する。実際に、Quadro K600のカーネル逐次実行と並列実行の場合を比較すると、並列実行によって、1.5から2.2倍の性能向上がみられた。

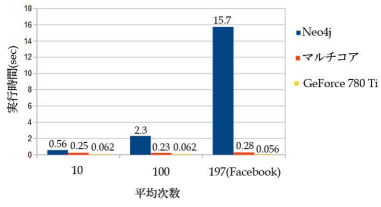
表1に示したとおり、Quadro K600とGeForce 780 Tiの性能の違いは、コア数と、メモリアクセスの速度であり、1コアの計算速度は同じである。1ブロックに使われるコア数が192であることから、Quadro K600とGeForce 780 Tiの実行時間の差はメモリアクセス速度の差によるものと考えられる。対象ノードが10,000の場合、メモリアクセス回数が少ないため、差は小さいが、50,000ノードと100,000ノードでは、GeForce 780 Tiの方が1.7から2.2倍高速となった。

GeForce 780 Tiのカーネル複数実行は、コア数が192を大きく上回っているため、大幅な性能向上がみられ、最大15.4倍の性能となった。

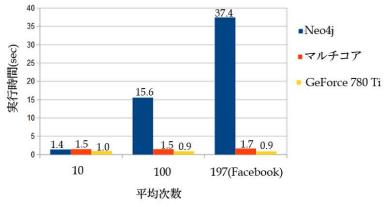
5.3.2 A*法

A*法では、目的ノードまでの重みの推定値を事前に与える必要がある。この評価では、A*法の推定値としてよく用いられる値として、それぞれのノードに座標を与え、あるノードから目的ノードへの距離を用いた。また、推定値の決め方は使用用途によって異なり、マルチコアやGPUでの実装に加えてしまうと用途によってその都度変更を加えなければならなくなるため、推定値の計算は、全てCPUのシングルコアの環境で行った。Dijkstra法と同様、始点と終点をランダムに決め、実行したときの実行時間を用いた。図5にNeo4j、マルチコア、GeForce 780 Tiのそれぞれで A*法を1回実行したときの平均実行時間を示す。

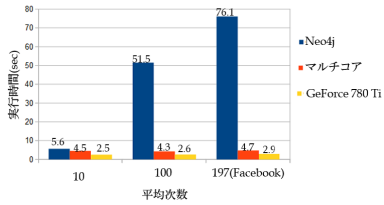
次数による実行時間の変化はDijkstra法と異なる傾向を示した。これはデータ構造によるものではなく、アルゴリズムの性質によるものである。A*法では、推定値に基づき、目的ノードまでの推定距離が最も短いノードを優先的に探索する。よって、次数が大きい方が、目的ノードに近づける確率が高くなり、探



(a)10,000 ノード

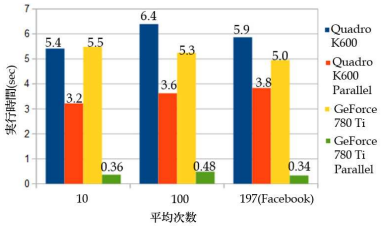


(b)50,000 ノード

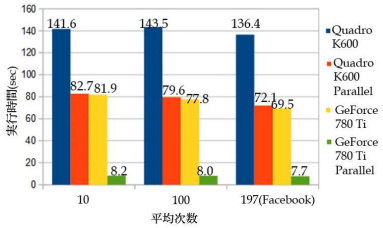


(c)100,000 ノード

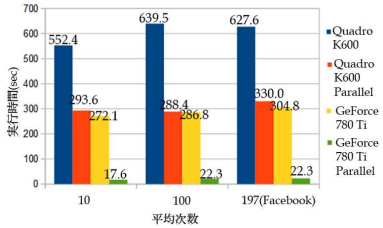
図 3　Dijkstra 法における Neo4j、マルチコア、GeForce 780 Ti の実行時間



(a)10,000 ノード

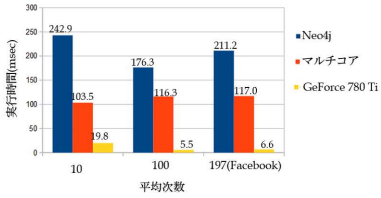


(b)50,000 ノード

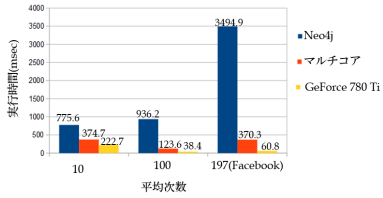


(c)100,000 ノード

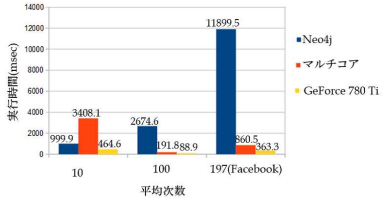
図 4　Dijkstra 法における Quadro K600、GeForce780 Ti の逐次、並列実行の実行時間



(a)10,000 ノード

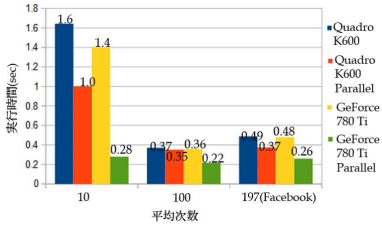


(b)50,000 ノード

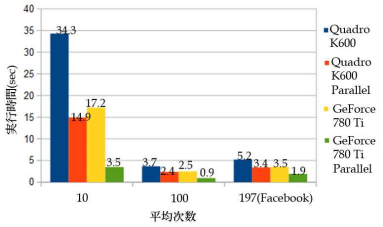


(c)100,000 ノード

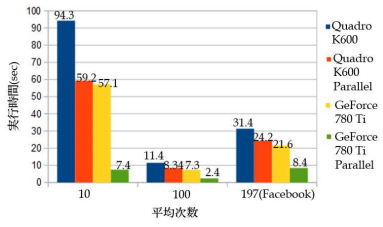
図 5　A*法における Neo4j、マルチコア、GeForce 780 Ti の実行の実行時間



(a)10,000 ノード



(b)50,000 ノード



(c)100,000 ノード

図 6　A*法における Quadro K600、GeForce 780 Ti の逐次、並列実行の実行時間

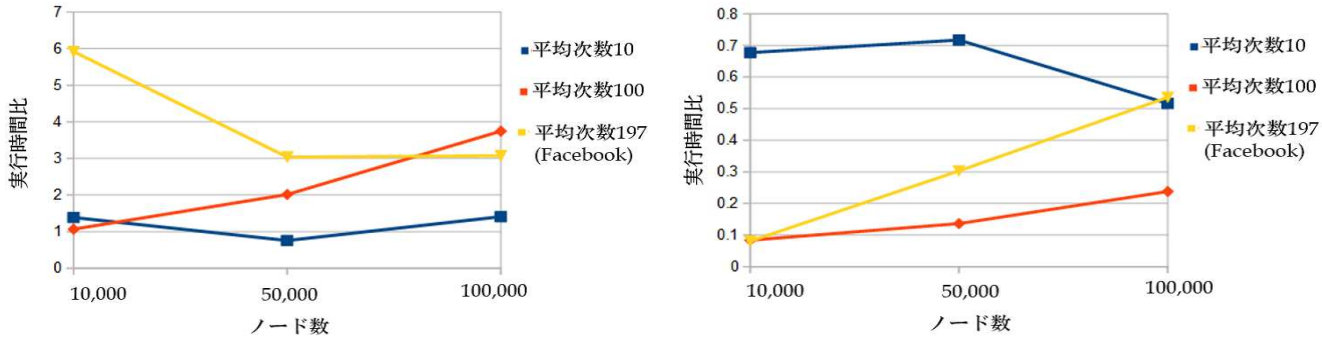
索回数が少なくなる。Facebook に基づくグラフの場合は、平均次数は多いが、次数の中央値が 99 で、10%のノードが次数 10 以下と、次数の少ないノードがランダムグラフに比べて多い。よって、平均次数 100 のランダムグラフが最も探索時間が短くなる。Neo4j では、10,000 ノードの時はこの傾向を示しているが、50,000 ノードと 100,000 ノードではデータ構造に基づく影響の方が大きくなり、次数が増加するにつれて実行時間が増加している。

マルチコアでは、平均次数 10 の 100,000 ノードの時は性能が悪化しているが、それ以外の場合は性能が向上した。Facebook に基づく 100,000 ノードのグラフでは、13.8 倍の性能向上がみられた。GPU では、全てのグラフにおいて性能が向上し、Facebook に基づく 100,000 ノードのグラフでは 32.8 倍に性能が向上した。図 6 は、A*法を 100 回連続で実行したときの実行時間をローエンド GPU Quadro K600 とハイエンド GPU GeForce 780 Ti のそれぞれでカーネルを逐次実行、並列実行した場合で比較したものである。

図 5 と同様に、アルゴリズムの性質により、どのノード数、実行環境でも、平均次数 100 の時の実行時間が短く、平均次数 10 の時の実行時間は長い。また、それぞれの環境での違いをみると、図 4 と同様の傾向を示した。Dijkstra 法との違いとして、GeForce 780 Ti のカーネル並列実行の時の性能向上の割合が Djikstra 法よりも小さくなった。これは、CPU において推定値を計算するのに生じるオーバーヘッドによるものだと考えられる。

5.4　オーバーヘッド

Neo4j で直接グラフ探索をせずに、隣接行列を作り、その隣接行列に対して探索を行う場合、隣接行列の生成にかかる時間はオーバーヘッドとなる。隣接行列は、どちらのアルゴリズムも同じものを用い、マルチコア、GPU も全て同じものを用いるため、ノード数と次数によってのみ変化する。このオーバーヘッドは、隣接行列の生成の際のものであるため、探索を実行



(a)Dijkstra 法　　(b)A*法

図 7　Neo4j の 1 回の実行時間対 GeForce 780 Ti の 1 回の実行時間とオーバーヘッドの和の比

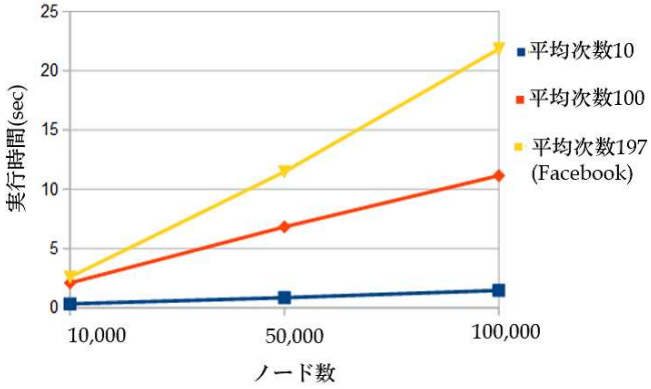


図 8　隣接行列生成のオーバーヘッド

する毎に発生するものではなく、ある対象グラフに対しての複数回の探索において、最初の一度のみ発生する。そのため、この実装方法は一つの対象グラフに対して探索の回数が多い用途に向いているといえる。逆に、一つのグラフに一度しか探索を行わないような用途では、高速化は期待できない。

図 8 はそれぞれのグラフの隣接行列を生成するオーバーヘッドを表している。図から、オーバーヘッドの大きさは、ノード数と次数の両方に比例して大きくなることがわかる。

図 7 は Neo4j の 1 回の実行時間対 GeForce 780 Ti の 1 回の実行時間とオーバーヘッドの和の比を表している。すなわち、1 を越えた場合、Neo4j の方が実行時間が長いことを表す。GeForce 780 Ti の 1 回の実行時間は図 3 と図 5 のものを用いた。

図 7(a) の Dijkstra 法では、ノード数 50,000 の平均次数 10 の場合を除いて、全ての場合でオーバーヘッドを含めても GeForce 780 Ti のほうが高速となった。これは、この実装方法が最も不利である条件下でも、Dijkstra 法は高速化できることを示す。すなわち、Dijkstra 法では、1 つの対象グラフに対して 1 度しか探索を行わないグラフデータベースとしては特殊な用途でもこの実装方法が利用できる。

図 7(b) の A*法では、A*法の場合、アルゴリズムの実行時間が Dijkstra 法よりも短いため、1 回の実行では、Neo4j よりも低速となり、複数回の実行でなければこの方法で高速化はできない。

オーバーヘッドは対象グラフの全体の隣接行列を作成したときにかかるものであり、グラフの更新の際のオーバーヘッドはこれに比べて極めて小さい。グラフ型データベースでは、一度作成したグラフを更新を繰り返して長期間に渡って運用するので、5.3 節で述べたグラフ探索の性能向上は有意である。

6. 結　　論

計算時間のみを考慮して 1 回の実行時間をみた場合、マルチコアでは Neo4j が特化している次数の少ないグラフを除いて高速化に成功し、Facebook に基づく 100,000 ノードのグラフでは最大 16.2 倍の性能となり、ハイエンドの GPU では、全てのグラフにおいて高速化に成功し、Facebook に基づく 100,000 ノードのグラフでは最大 32.8 倍の性能となった。

オーバーヘッドを考慮して、対象グラフを 1 回のみ探索した場合でも、Dijkstra 法の場合はほぼ全ての場合で高速化できる。しかし、A*法を高速化することはできない。しかし、グラフ型データベースでは、1 つのグラフに対して何度も探索を実行するので、計算性能の向上は有意である。

謝辞　本研究の一部は、JST 戦略的創造推進事業さきがけ「多様な構造型ストレージ技術を統合可能な再構成可能データベース技術」の補助による。

文　　献

[1] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor and Tor M. Aamodt "Characterizing and Evaluating a Key-value Store Application on Heterogenenous CPU-GPU Systems" Performance Analysis of Systems & Software, pp. 88-98, April 2012.

[2] Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos and Arturo Gonzalez-Escribano "A New GPU-based Approach to the Shortest Path Problem" High Performance Computing and Simulation, pp.505-511, July 2013.

[3] Duane Merill, Michael Garland and Andrew Grimshaw "Scalable GPU Graph Traversal" Principles and Practice of Parallel Programming, pp.117-128, August 2012.

[4] Sadegh Nobari Thanh-Tung Cao Stéphane Bressan Panagiotis Karras "Scalable Parallel Minimum Spanning Forest Computation" Principles and Practice of Parallel Programming, pp.205-214, August 2012.

[5] "JOMP" `http://www2.epcc.ed.ac.uk/computing/research_activities/jomp/index_1.html`

[6] "jcuda.org," `http://www.jcuda.org`

[7] Johan Ugander, Brian Karrer, Lars BackStrom and Cameron Marlow, "The Anatomy of the Facebook Social Graph" Arxiv prepring arXiv:1111.4503 November 2011.

[8] Ian Robinson, Jim Webbem and Emil Eifrem "Graph Databases", Oreilly, 2013.