Vol.2011-HPC-130 No.14
2011/7/27

GPGPUを用いた高速大規模グラフ処理に向けて

白幡晃一[†1] 佐藤仁[†1]
鈴村豊太郎[†1,†2] 松岡聡[†1,†3,†4]

データ量の肥大化，ストレージの省コスト化，オンラインソーシャルネットワークの成功等に伴い大規模グラフ処理の重要性が高まっている．また，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．大規模グラフ処理ライブラリの一つに PEGASUS があり，MapReduce の反復処理によって計算することができる．GPU を利用した MapReduce 処理ライブラリの一つに Mars があるが，大規模グラフ処理に対して GPU を使用してどの程度高速化できるのか，またメモリあふれへの対処やマルチ GPU 化した場合のデータの割り振り方法は明らかではない．Mars 上にグラフ処理アプリケーション（PageRank，Random Walk with Restart，Connected Components）を実装し，PEGASUS との比較実験を行った結果，反復 1 回あたり PageRank で 2.17〜9.53 倍，RWR で 2.18〜5.47 倍，Connected Components で 2.41〜8.46 倍の高速化がされることを確認した．

Towards GPGPU-Based Large-Scale Fast Graph Processing

Koichi Shirahata,[†1] Hitoshi Sato,[†1]
Toyotaro Suzumura[†1,†2] and Satoshi Matsuoka[†1,†3,†4]

Large-scale graph processing is becoming more important due to the large volume of available data, the low cost of storage and the stunning success of online social networks. Besides, recent large-scale computing systems tend to employ GPUs to gain good peak performance and high memory bandwidth. Mars is one of the MapReduce library accelerated with GPUs;however, the problems on GPGPU-based large-scale graph processing, such as the performance improvement by using GPUs, the memory management in GPUs, and the data distribution between multiple GPUs, are not investigated. In order to clarify the problems, we implemented several graph processing applications such as PageRank, Random Walk with Restart(RWR) and Connected Components on top of Mars and compared the results with PEGASUS, one of the large-scale graph processing library. Our experimental results show that the mean time of one iteration in Mars is 2.17-9.53 times faster on PageRank, 2.18-5.47 times on RWR, and 2.51-8.46 times on Connected Components respectively compared with PEGASUS.

1. はじめに

近年はデータ量の肥大化，ストレージの省コスト化，オンラインソーシャルネットワークの成功等に伴い大規模グラフ処理の重要性が高まっている．グラフはコンピュータネットワーク，ソーシャルネットワーク，モバイルネットワーク，World Wide Web 等，至るところに存在する．大規模グラフ処理に関する研究も進められており，一例として 2010 年 6 月にはスーパーコンピュータでの大規模グラフ処理の性能を競う Graph500 ランキングが開始されている[1)]．

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 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[2)]．アルゴリズムの研究も進められており，GIM-V(Generalized Iterative Matrix-Vector multiplication) と呼ばれる行列ベクトル積の反復計算によってグラフ処理を MapReduce で計算することが可能となっている[3)]． PEGASUS は Hadoop ベースのグラフ処理ライブラリであり，MapReduce の反復計算によって大規模グラフ処理を実行できる．

一方，GPGPU[4)] 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[5)] 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んでいる．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ている．GPU を使用した MapReduce ライブラリも存在しており，GPU の高い並列度を活かして処理を行うことができる．

しかし，大規模グラフ処理に対して GPU を有効利用する方法は明らかではない．GPU

†1 東京工業大学
†2 IBM 東京基礎研究所
†3 科学技術振興機構
†4 国立情報学研究所

を使用してどの程度高速化できるかは明らかではなく，また GPU のメモリ量は限られているため，メモリあふれへの対処やマルチ GPU 化した場合のデータの割り振り方法を考慮する必要がある．

そこで，GPU を搭載する環境で Mars に GIM-V アルゴリズムを適用し，MapReduce の反復計算によってグラフ処理を行う．PageRank，Random Walk with Restart，Connected Components の各グラフ処理アプリケーションを複数の MapReduce を組み合わせることによって実装した．PEGASUS との比較実験を行った．反復処理 1 回分の平均時間を測定した結果，PageRank で 2.17〜9.53 倍，RWR で 2.18〜5.47 倍，Connected Components で 2.41〜8.46 倍の性能向上を確認することができた．

2. 背　　景

この章では大規模グラフ処理と MapReduce[2] モデル，GPGPU について述べる．

2.1 大規模グラフ処理

大規模グラフ処理の一実装として，PEGASUS[3] が挙げられる．PEGASUS は Hadoop ベースのグラフ処理ライブラリであり，PageRank，Random Walk with Restart などの典型的な計算を実行することができる．PEGASUS のためのプリミティブとして，GIM-V(Generalized Iterated Matrix-Vector multiplication) がある．GIM-V とは，行列ベクトル積を一般化したものであり，多くのグラフ処理は GIM-V により記述できる．また GIM-V は MapReduce により記述できるため，大規模並列処理が可能となっている．

2.2 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．

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．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．Mars は C++で実装されており，Hadoop に比べファイル入出力が少ない Mars を対象とする．

2.3 GPGPU

近年 GPGPU (General-purpose computing on GPU)[4] 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，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 言語と C++の拡張として提供されている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. Mars への GIM-V の適用

この章では Mars への GIM-V の適用方法，および Mars の構造について述べる．

3.1 GIM-V

GIM-V(Generalized Iterative Matrix-Vector multiplication) とは，行列ベクトル積の一般化である．行列ベクトル積は combine2，combineAll，assign の 3 つのオペレーションによって表現できる．$n \times n$ の行列 M と，大きさ n のベクトル v があるとする．通常の行列ベクトル積は次のように表現されるとする．

$$M \times v = v' \text{ where } v'_i = \sum_{j=1}^{n} m_{i,j} v_j$$

combine2 で $m_{i,j}$ と v_j の掛け算を行い，combineAll でノード i についての combine2 の結果 n 個分の和を取り，assign で v_i を新しい結果 v'_i に更新する．GIM-V では上記の計算の一般化を行う．オペレータ $\times_G$ を導入し，3 つのオペレーションを任意に定義する．

$$v' = M \times_G v$$
$$\text{where } v'_i = \text{assign}(v_i, \text{combineAll}_i(\{x_j \mid j = 1..n, \text{ and } x_j = \text{combine2}(m_{i,j}, v_j)\}))$$

combine2 で $m_{i,j}$ と v_j を連結し，combineAll でノード i についての combine2 の全結果を連結し，assign で v_i をどのように v'_i に更新するかを決定する，というオペレーションを行う．また，GIM-V は反復解法であり，上記のオペレーションをアルゴリズムによって定められた収束条件を満たすまで繰り返し適用する．3 つのオペレーションを適当に定義することにより，PageRank や Random Walk with Restart，Connected Components などの有用なアルゴリズムを得ることができる．

PageRank は Google によって使用されている，ウェブページの相対的な重要性を計算するための有名なアルゴリズムである[9]．ウェブページの数 n を PageRank ベクトル p とするとき，次の固有方程式を満たす．

$$p = (cE^T + (1-c)U)p$$

c は減衰率であり，通常 0.85 に設定される．E は行について正規化された隣接行列であり，U は全要素が $1/n$ の行列である．固有ベクトル p^{cur} を全要素 $1/n$ で初期化し，次の PageRank を $p^{next} = (cE^T + (1-c)U)p^{cur}$ によって計算する．GIM-V では，列について正規化された行列 E^T によって M を構成し，$p^{next} = M \times p^{cur}$ によって計算される．3 つのオペレーションは次のように定義される．

$$\text{combine2}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$$
$$\text{combineAll}_i(x_1, \ldots, x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$$
$$\text{assign}(v_i, v_{new}) = v_{new}$$

Random Walk with Restart(RWR) はグラフのノード間の近さを測るアルゴリズムである[10]．RWR では，ノード k からの近接ベクトル r_k は次の等式を満たす．

$$r_k = cMr_k + (1-c)e_k$$

e_k は k 番目の要素が 1 でその他の要素が全て 0 のベクトルであり，c はリスタート確率パラメータであり，典型的には 0.85 に設定される[10]．M は PageRank と同様に列で正規化され転置された隣接行列である．GIM-V では，RWR は $r_k^{next} = M \times_G r_k^{cur}$ によって計算され，3 つのオペレーションは次のように定義される ($I(x)$ は x が真のとき 1 で，偽のとき 0 をとる)．

$$\text{combine2}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$$
$$\text{combineAll}_i(x_1, \ldots, x_n) = (1-c)I(i \neq k) + \sum_{j=1}^{n} x_j$$
$$\text{assign}(v_i, v_{new}) = v_{new}$$

Connected Components は各ノードから連結しているノードを調べるアルゴリズムである．今回 HCC(Hadoop based Connected Components) と呼ばれる，各ノードから連結しているノードの最小の番号を調べるための GIM-V アプリケーションを利用する．HCC では各ノード v_i について反復 h 周目の連結番号 c_i^h を保持する． 初めは自分のノード番号 $c_i^0 = i$ で初期化し，各反復で c_i^h を隣接ノードに送信する．c_i^{h+1} は現在の番号と新たに受信した番号の小さい方の値で更新する．以上の計算は GIM-V で次のように記述される．

$$c^{h+1} = M \times_G c^h$$

3 つのオペレーションは次のように定義される．

$$\text{combine2}(m_{i,j}, v_j) = m_{i,j} \times v_j$$
$$\text{combineAll}_i(x_1, \ldots, x_n) = \text{MIN}\{x_j \mid j = 1..n\}$$
$$\text{assign}(v_i, v_{new}) = \text{MIN}(v_i, v_{new})$$

GIM-V の MapReduce 処理は GIM-V BASE と呼ばれる 2 段階アルゴリズムによって行われる．ステージ 1 では combine2 オペレーションを行い，出力は key が始点ノード番号で value が部分的に連結された結果である．ステージ 1 の出力がステージ 2 の入力となる．ステージ 2 では combineAll によってステージ 1 の全結果を連結し，assign によって新しいベクトルで元のベクトルを置き換える．この 2 段階アルゴリズムはアプリケーションによって定められた収束基準が満たされるまで反復的に繰り返される．

GIM-V BASE は元は Hadoop のために考えられたものであるが，実装には依存しない

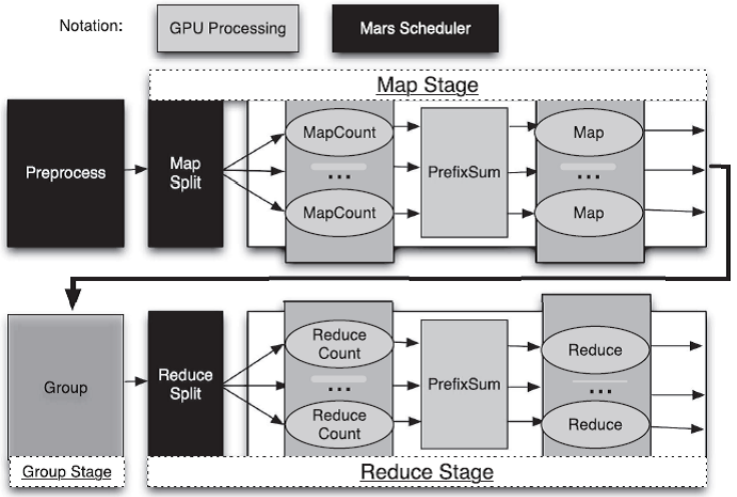

図 1 Mars のワークフロー
Fig. 1 The Mars workflow

ため他の MapReduce 処理系にも取り入れることができる．我々は GPU 向け MapReduce フレームワークである Mars に GIM-V BASE を適用し，グラフ処理計算を行う．

3.2 Mars

Mars のデータ構造は他の MapReduce 実装で用いられているキューや連結リストのような動的なデータ構造ではなく配列を使用する．入力レコード，中間レコード，出力レコードは，key 配列，value 配列，directory index 配列の 3 つの配列に格納される．directory index 配列は < key offset, key size, value offset, value size > のエントリから構成される．directory index 配列を調べることで，key または value を対応する key 配列または value 配列の offset の場所から取得する．

Mars のワークフローを**図 1** に示す． Mars スケジューラは CPU 上で実行され，タスクを GPU に割り振る．Map ステージの前に，Mars は CPU 上でディスクから入力データの前処理としてメモリ上で入力データを key/value の組に変換する．完了後，メモリ上の入力レコードを GPU デバイスメモリに転送する．

Map ステージでは，Map Split が入力レコードを GPU スレッドに均等に割り振る．各スレッドはユーザに定義される MapCount 関数を実行し，Map が出力する中間レコードの数と合計サイズを計算する．続いてランタイムが GPU ベースの Prefix Sum を実行して出力サイズと書き込む位置を得る．最後に，CPU が出力バッファをデバイスメモリ内に割り当ててから，各 GPU スレッドはユーザに定義される Map 関数を実行して出力結果を得る．各スレッドの書き込みの位置が事前に計算されており他のスレッドと干渉することがないため，並行するスレッド間での書き込み競合は発生しない．Shuffle ステージでは，ソートベースのアプローチによってレコードを集約する．ハッシュベースのアプローチでは各ハッシュ値のレコード毎にソートを行う必要があるためである．ソートは GPU ベースの Bitonic Sort を実行する[11][12]．Reduce ステージでは，Reduce Split が同じ key のレコードのグループを GPU スレッドに割り振る．Map と同様に結果のサイズと書き込みの位置を各スレッドが把握するため，Reduce は lock-free に実行される．

4. 実　　装

Mars 上にグラフ処理汎用モデル GIM-V の実装を行った．PEGASUS の実装を C++に移植することによって Mars 上に実装した．PEGASUS との比較実験を行う際に公平な比較をできるようにするため，アルゴリズムの変更は行っていない．

各アプリケーションについての処理は次のようになる．PageRank は前述の方法である GIM-V BASE の通り，2 つの MapReduce で処理を行う．収束判定については，2 つ目の Reduce で各スレッドが前回の結果との差を計算することにより行う．RWR は前述の方法とは異なり，実際には 5 つの MapReduce で処理が行われる．1 つ目の MapReduce で combine2 を行い，2 つ目と 3 つ目で combineAll と assign を行い，4 つ目で新しいベクトルと元のベクトルの差を計算し，5 つ目で収束判定を行う．Connected Components は前述の方法とほぼ同じであるが，3 つの MapReduce で処理が行われ，3 つ目で新しいベクトルと元のベクトルとの差の計算を行う．

反復処理は各 MapReduce 毎に結果をファイルへ読み書きすることによってデータの受け渡しを行う．本来 Mars では MapReduce の反復処理に対応しておりファイルへの読み書きは必要ないが，複数の MapReduce を組み合わせる場合にはライブラリに修正を加える必要がある．ファイルへの読み書きを減らすことは今後の課題の一つである．

5. 実　　験

Mars に GIM-V を適用した場合の性能を評価するため，Mars と PEGASUS の比較実験を行った．

表 1　実験環境

	CPU	GPU
デバイス	Intel(R)Core(TM)i7	Tesla C2050
周波数	2.67GHz	1.15GHz
物理コア数	4 コア	448 コア
メモリ	12.3GB	2.8GB

5.1 概　　要

各グラフ処理アプリケーションの反復処理 1 回分の平均時間を測定した．ベンチマークアプリケーションには，PageRank，Random Walk with Restart(RWR)，Connected Components の 3 つを用いた．アプリケーションの実装については前述した通りである．

入力データは以下の値を持つ．SCALE は頂点数の 2 の対数であり，edge counter は頂点数に対する枝の数の比率である．合計の頂点数は 2^{SCALE} となり，枝の数は $edgefactor \times 2^{SCALE}$ となる．今回は SCALE を 14〜20 と変化させ，edge factor は 16 として実験を行った．グラフの生成は Graph500 で使用されている Kronecker generator を使用して行った[1)13)]．Kronecker generator は Recursive MATrix(R-MAT) グラフ生成アルゴリズム[14)]に類似しており，スケールフリー性がある（一部の頂点の次数は非常に多い）．Kronecker generator は R-MAT generator と同様に隣接行列のデータ構造を使用する．グラフの隣接行列を 4 つの均等に分割された領域に再帰的に分割し，枝を均等でない確率でこれらの領域に割り振る．初めは隣接行列は空であり，枝は一度に一つずつ追加される．各枝は確率 A，B，C，D で 4 つの領域のうち一つを再帰的に選択する．各確率は以下のように設定されている．

$A = 0.57$　$B = 0.19$

$C = 0.19$　$D = 1 - (A + B + C) = 0.05$

実験は GPU 搭載マシン 1 ノードで行った．1 ノードの CPU のコア数はハイパースレッディングを使用して 8 コア，GPU の台数は 1 台である．OS は Linux2.6.18 であり、コンパイラは gcc4.1.2，nvcc3.2 を使用した。各ノードで使用した CPU・GPU は**図 1** の通りである．CUDA ドライバは 4.0，CUDA ランタイムは 3.2，CUDA Capability は 2.0 である。GPU のメモリクロックは 1.5GHz，SM 数は 14，メモリバンド幅は 144GB/s，システムインターフェイスは PCIe x16 Gen2，共有/L1 キャッシュサイズは 64KB，L2 キャッシュサイズは 768KB である．PEGASUS に関する設定を以下に示す．Hadoop は 0.21.0，Java は 1.6.0(Oracle Java VM) を使用した．Mapper，Reducer の数を変化させて実験を行った．

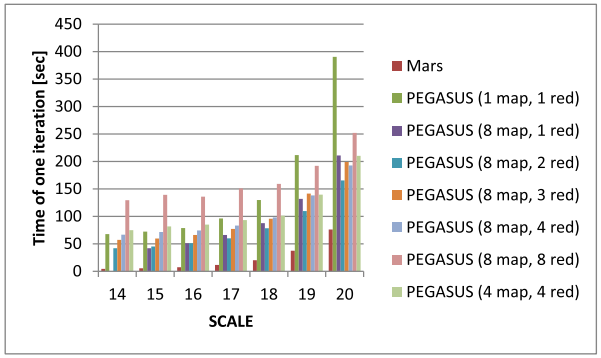

図 2　PageRank: 反復 1 回の平均時間
Fig. 2　PageRank: Time of one iteration

Mapper は最大 8 つ，Reducer は 4 つまで使用した．ファイルシステムには HDFS(Hadoop Distributed File System) を使用した．なお，PEGASUS では CPU のみを使用し，GPU は使用していない．

5.2 実 験 結 果

Mars と PEGASUS の反復処理 1 回分の平均時間は，それぞれ**図 2**，**図 3**，**図 4** のようになった．PageRank の PEGASUS に対する Mars の加速倍率を**図 5** に示す．なお，反復回数はいずれも 10〜20 回程度であった．　PEGASUS の中で比較的性能の高い 8Mapper，2Reducer に対し，PageRank の場合 2.17〜9.53 倍，RWR の場合 2.18〜5.47 倍，Connected Components の場合 2.41〜8.46 倍の性能向上が見られた．原因として PEGASUS では Hadoop を用いているために Map，Reduce 毎のファイル入出力によるオーバーヘッドが大きいことが考えられる．Mars では Map，Shuffle，Reduce の間，データは GPU 上にあるため PEGASUS に比べファイル入出力が少ない．また，各アプリケーション毎に加速倍率が異なっており，高い順に PageRank，Connected Components，RWR となっている．これは，各反復での MapReduce の回数がそれぞれ 2 つ，3 つ，5 つと異なっていることに対応しており，MapReduce の回数が増えるほど Mars の場合のファイル入出力が多く発生することが原因の一つとして考えられる．また，加速倍率は入力データが大きくなるにつれて小さくなる傾向が見られる．これは Mars のファイル入出力の時間が相対的に増えているためであると考えられる．

PageRank の反復 1 回の平均時間をブレークダウンした結果をそれぞれ**図 6**，**図 7** に

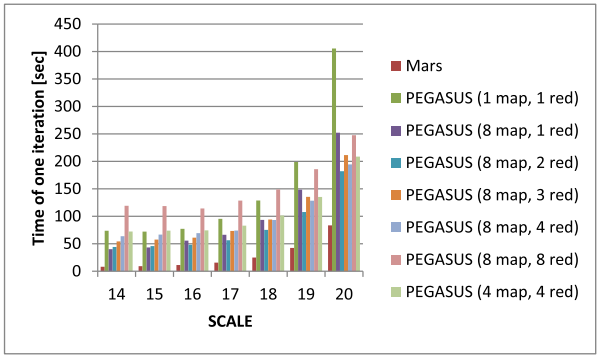

図 3 RWR: 反復 1 回の平均時間
Fig. 3 RWR: Time of one iteration

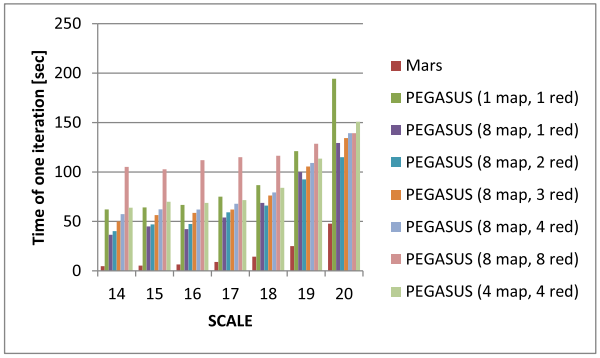

図 4 Connected Components: 反復 1 回の平均時間
Fig. 4 Connected Components: Time of one iteration

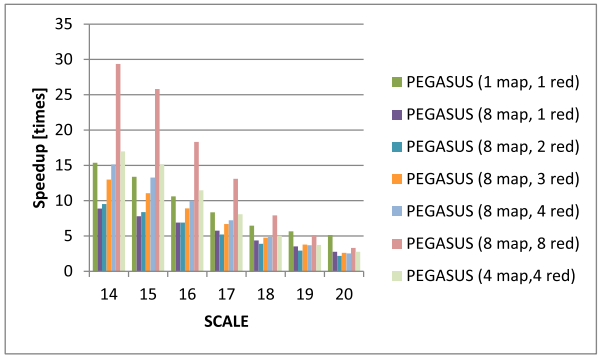

図 5 PageRank: PEGASUS に対する Mars の加速倍率
Fig. 5 PageRank: Mars speedup over PEGASUS

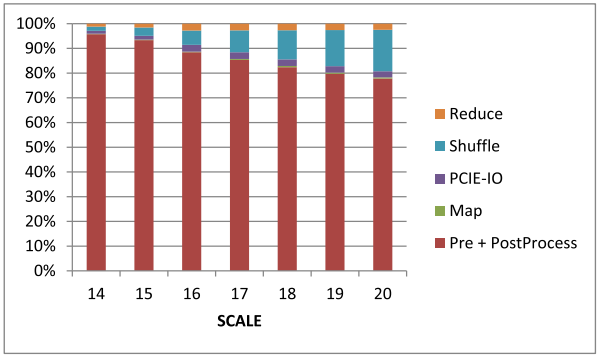

図 6 PageRank: Mars のブレークダウン
Fig. 6 RageRank: Time breakdown of Mars

示す． Mars では前処理と後処理の実行時間が 77.8%〜95.7%を占めている．これは，各 MapReduce の開始時と終了時にファイルへの読み書きが発生しており，ディスクへの入出力がボトルネックになっているためであると考えられる．PEGASUS では Map，Reduce ともに Mars に比べ時間がかかっているが，これは MapReduce の処理中に常にファイルへの読み書きが発生しているため，Mars に比べてディスクへの入出力が多いためであると考えられる．また Mapper，Reducer の数は 8 と 2 とのときに相対的に性能が高かった．これはジョブの Map タスク数は大きいため Mapper 数は多いほうが良く，一方 Reduce 数は多くすると逆に性能が劣化する場合があることを示している．

6. 議　　論

今回の結果から，大規模グラフを処理するために得られた知見を述べる。まず、全体の実行時間に占めるファイル入出力の割合が大きいという結果が得られた。PEGASUS では Mars に対して非常に実行時間が大きくなっているが、主な要因はファイル入出力が占めると考えられる。PEGASUS では Map，Reduce 毎にファイル入出力を行っている一方、Mars

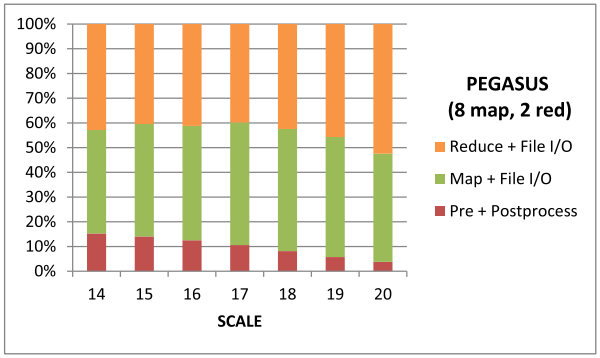

図 7 PageRank: PEGASUS のブレークダウン
Fig. 7 RageRank: Time breakdown of PEGASUS

では MapReduce 内でデータは GPU 内に留まっており、CPU へのデータ転送やファイル入出力は発生していない。しかし、現状の Mars の実装では各 MapReduce の開始時と終了時に CPU へのデータ転送とファイル入出力が発生している。そのため入力データ量が大きい場合に実行時間が PEGASUS と比べ相対的に大きくなってしまっている。

また、上記のファイル入出力は MapReduce 処理によるオーバーヘッドであるということができる。GIM-V により MapReduce によりグラフ処理を容易に記述できる一方、処理に無駄が多いという欠点がある。グラフ処理を GPU で実装する研究も行われており、一例として PageRank アルゴリズムを GPU で処理する実装がある[15]。今後の課題として、ナイーブな実装との比較実験を行う予定である。

7. 関 連 研 究

大規模グラフ処理システムの研究として Pregel が挙げられる[16]．Pregel はバルク同期並列を採用しており，各頂点が並列に関数を呼び出し，各反復ごとに頂点と枝が更新される．また耐故障性についてもチェックポイントリスタートを取り入れている．しかし，Linux 上で動作せず，また Erlang を使用してアプリケーションを書く必要がある．また，MPI をベースにした C++の分散メモリ用並列グラフ処理ライブラリに Parallel BGL がある[17]．グラフ処理ベンチマークに HPCS があり[18]，HPCS でグラフの中心性のアルゴリズムの実装などが行われている[19]．また Incremental GIM-V と呼ばれる、PageRank などのアルゴリズムでグラフの構造が変化する部分を特定して計算量を減らすことによる高速化の試みも行われている[20]．

GPU を用いたグラフ処理に関する研究として，GPU を用いた最短路問題を解く研究が挙げられる[21]．1000 万頂点から成る平均 6 次数のランダムグラフに対しては BFS を 1 秒で SSSP を 1.5 秒で解くことに成功している．但し，スケールフリー性があるグラフや DIMACS チャレンジの実道路ネットワークでは CPU に対する優位性がほとんど得られていない．また，マルチノード上，マルチ GPU 上での MapReduce に関する研究も行われている[22][23]．これらの実装を考慮して今後マルチ GPU 化を行った上で，メモリ階層の管理などについて探求する予定である．

8. お わ り に

GPU を搭載する環境で Mars に GIM-V アルゴリズムを適用し，MapReduce の反復計算によってグラフ処理を行った．PageRank，Random Walk with Restart，Connected Components の各グラフ処理アプリケーションを複数の MapReduce を組み合わせることによって実装し，PEGASUS との比較実験を行った．反復処理 1 回分の平均時間を測定した結果，PageRank で 2.17〜9.53 倍，RWR で 2.18〜5.47 倍，Connected Components で 2.41〜8.46 倍の性能向上を確認することができた．

今後の課題としては，性能の改善，メモリあふれへの対処，マルチ GPU 化などが挙げられる．GPU のメモリ量には制限があるため，大規模グラフの場合にはメモリ階層の管理が必要となる．GPU デバイスメモリ，CPU のメモリ，SSD，ストレージなどのメモリ階層の効率的な管理を今後行っていく予定である．またグラフ処理は頂点ごとの計算量が少なくメモリアクセスの局所性が少ないなど並列化の難しさがあるため，これらを考慮した効率的なマルチ GPU 処理についても探求する予定である．

謝辞 本研究の一部は 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) David A. Bader, Jonathan Berry, S. K. R. M. E. J. R.: The Graph 500 List, Graph500.org (online), available from ⟨http://www.graph500.org/⟩
2) 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).
3) Kang, U., Tsourakakis, C.E. and Faloutsos, C.: PEGASUS: A Peta-Scale Graph

Mining System- Implementation and Observations (2009).
4) 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).
5) John, N., Ian, B., Michael, G. and Kevin, S.: Scalable Parallel Programming with CUDA, *Queue*, Vol.6, No.2, pp.40–53 (online), DOI:http://doi.acm.org/10.1145/1365490.1365500 (2008).
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) Fang, W., He, B., Luo, Q. and Govindaraju, N.K.: Mars: Accelerating MapReduce with Graphics Processors, *IEEE Transactions on Parallel and Distributed Systems*, Vol.22, pp.608–620 (online), DOI:http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.158 (2011).
9) Brin, S. and Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine, *Seventh International World-Wide Web Conference (WWW 1998)*, (online), available from ⟨http://ilpubs.stanford.edu:8090/361/⟩ (1998).
10) yu Pan, J., Yang, H., Duygulu, P. and Faloutsos, C.: Automatic multimedia cross-modal correlation discovery, *In KDD*, pp.653–658 (2004).
11) 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).
12) He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q. and Sander, P.: Relational joins on graphics processors, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, New York, NY, USA, ACM, pp.511–524 (online), DOI:http://doi.acm.org/10.1145/1376616.1376670 (2008).
13) Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C. and Ghahramani, Z.: Kronecker Graphs: An Approach to Modeling Networks, *J. Mach. Learn. Res.*, Vol.11, pp.985–1042 (online), available from ⟨http://portal.acm.org/citation.cfm?id=1756006.1756039⟩ (2010).
14) Chakrabarti, D., Zhan, Y. and Faloutsos, C.: R-MAT: A recursive model for graph mining, *In SDM* (2004).
15) Cevahir, A., 額田　彰, 松岡　聡：Efficient PageRank on GPU Clusters, 情報処理学会研究報告, Vol.HPC-128, pp.1–6 (2010).
16) Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N. and Czajkowski, G.: Pregel: a system for large-scale graph processing, *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, New York, NY, USA, ACM, pp. 6–6 (online), DOI:http://doi.acm.org/10.1145/1582716.1582723 (2009).
17) Gregor, D. and Lumsdaine, A.: The parallel bgl: A generic library for distributed graph computations, *In Parallel Object-Oriented Scientific Computing (POOSC* (2005).
18) (cray, J.F., (uc, J.G., Microsystems, S., (georgia Tech, K.M., (formerly Of, B.M. and (mit/ll, T.M.: HPCS Scalable Synthetic Compact Applications 2 Graph Analysis (2006).
19) Brandes, U.: A Faster Algorithm for Betweenness Centrality, *Journal of Mathematical Sociology*, Vol.25, pp.163–177 (2001).
20) 西井俊介，鈴村豊太郎：データストリーム処理によるインクリメンタルグラフ処理に向けて，電子情報通信学会データ工学研究会 (2011).
21) Harish, P. and Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA, *Proceedings of the 14th international conference on High performance computing*, HiPC'07, Berlin, Heidelberg, Springer-Verlag, pp. 197–208 (online), available from ⟨http://portal.acm.org/citation.cfm?id=1782174.1782200⟩ (2007).
22) Stuart, J.A. and Owens, J.D.: Multi-GPU MapReduce on GPU Clusters, *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, IEEE (2011).
23) Plimpton, S. J. and Devine, K. D.: MapReduce in MPI for Large-scale Graph Algorithms, *Parallel Computing*, Vol. In Press,, No. February, pp. 1–39 (online), available from ⟨http://linkinghub.elsevier.com/retrieve/pii/S0167819111000172⟩ (2011).