同期的な分散グラフ処理におけるサブグラフ単位での負荷分散手法

山口　航†　宮崎　純†

† 東京工業大学　〒 152–8550 東京都目黒区大岡山
E-mail: †yamaguchi.w.aa@m.titech.ac.jp, ††miyazaki@cs.titech.ac.jp

あらまし　分散環境での同期的なグラフ処理では各ノードでの計算時間が均一であることが求められる。しかし、ヘテロジニアスな環境やアルゴリズムの性質によりノード間の計算時間に不均一が生じることがある。また、ワークロードをランダムに選択しノード間で移動させバランスを調整してもかえって通信量が増えることも考えられる。そこで本研究では、あらかじめ閉じたサブグラフを作成しておきそれらを単位としてワークロードを移動させることによってリアルタイムにノード間のバランスを調整する手法を提案し、既存のマイグレーション手法との比較により提案手法の評価を行う。

キーワード　分散グラフ処理, 負荷分散

1　はじめに

近年情報社会の発達により取得することができる情報の種類と量が増え、それらを高速に処理する必要性が高まっている。また、それらのデータ同士は複雑に絡み合いコンピューターサイエンス上ではグラフ構造で表すことで効率よく処理することができる。回路の集積面積を考えると 1 台の計算機で処理することには限界がきており大規模なデータに対しては複数台の計算機を用いて並列分散処理をすることが必要となっている。

並列処理を行う有名なフレームワークとして Google が提案した MapReduce [1] を挙げることができる。MapReduce はデータセットをクラスタ内のノードに分散させ、入力データから Key/Value ペアを生成する Map ステップと、それらのデータのソート及びグループ化を行う Shuffle ステップ、グループごとに Key/Value ペアを集約し、結果を算出する Reduce ステップの三つのステップに分けて並列計算を行う。この処理は大量の行データを処理することには適しているが、グラフ構造ではデータ同士が複雑に絡み合い分割することが難しいということと、グラフ処理では複数回のイテレーションを必要とするアルゴリズムが多いのに対し MapReduce は 1 回のイテレーションを基本としていることからグラフ処理には適していない。

そこで、2010 年に分散グラフ処理を目的とした計算モデルである Pregel [2] が提案された。Pregel は Bulk Synchronous Parallel(BSP) [3] をもとに作られた同期的な並列処理システムモデルである。Pregel は Vertex-centric フレームワーク [4] でありグラフアルゴリズムは頂点がローカルにあるデータ及びリモートのノードにあるデータを用いてどのように計算をするかで表現されている。Pregel ではスーパーステップの繰り返しによりアルゴリズムの終了条件にいたるまで計算が進む。スーパーステップ内ではそれぞれの頂点は並列に計算され、ユーザーが定義した全く同じアルゴリズムを実行する。アルゴリズム実行中は頂点が自分自身の値を更新する、エッジの値を更新する、1 つ前のスーパーステップのメッセージを受け取る、隣接頂点にメッセージを送る、グラフの形状を変更することによって計算を進めていく。実際に Pregel をもとにして作成された実装モデルは多く存在し、Giraph [5],GPS [6] がそれらの中でも有名な実装である。

このような同期的な分散処理においては、ノードの計算時間が不均一であると計算時間の一番遅いノードに全体の計算時間が依存してしまうため、すべてのノードが均等な計算時間で処理を行っていくことが求められる。Pregel をもとにした分散グラフ処理システムは様々提案されているが、これらのシステムはホモジニアスな分散環境及びグラフ全体を用いたアルゴリズムを実行していくことを前提として作られている。しかし、クラウド上でのプログラムの実行が増えた現在ではヘテロジニアスな環境で分散処理をする事例も増えており、また機械学習のアルゴリズムなどグラフアルゴリズムに関しても様々なパターンのアルゴリズムを処理する必要性が高まっている。このような状況下では既存のシステムではノード間の実行時間が不均一となり全体としての実行時間に遅れが出てしまう。

ノード間の時間の不均一を解消する方法として分散処理の途中にワークロードを移動させるマイグレーション手法が提案されている。しかし、既存のマイグレーション手法ではマイグレーションを行うかどうかの選定の曖昧さや、デフォルトでは HDRF [7] などの手法によりノード間の通信量が少なくなるように分割されているもののマイグレーション時にはマイグレーションするノードをランダムに選んでいることなど問題点が残っている。また、マイグレーションを行うタイミングがスーパーステップの最後だと、1 スーパステップ目の計算時にノード間の計算実行時間に大きな不均一が生じた場合は一番処理の速いノードは処理の一番遅いノードの計算終了を何もせずに待つ必要が生じてしまう。

そこで本研究では、あらかじめ閉じたサブグラフを作成しておきそれらを単位としてマイグレーションを実行し、マイグレーションをグラフ計算と並行してリアルタイムで行う新たなマイグレーション手法を導入した分散グラフ処理システムを提案する。提案手法の評価は既存手法と同じマイグレーション手

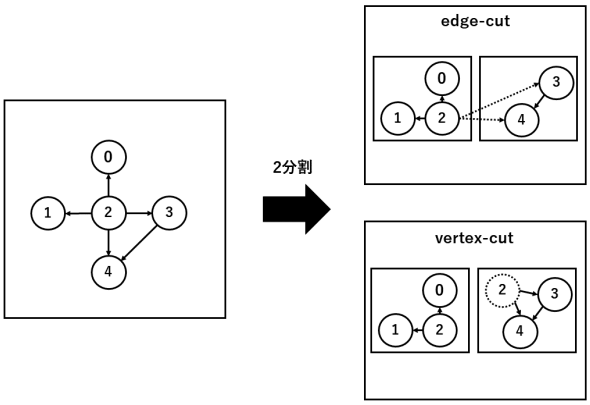

図 1　edge-cut と vertex-cut

法を同じ環境で再現し、ヘテロジニアスな環境でのグラフアルゴリズムの実行時間の比較を行う。

2　関 連 研 究

2.1　PowerGraph

Pregel が edge-cut の分散グラフ処理システムだったことに対して PowerGraph [8] は vertex-cut の分散グラフ処理システムである。図 1 に示すように edge-cut は頂点を各ノードに分配しエッジがノードをまたいで存在する場合はそこで通信コストがかかる。一方 vertex-cut ではエッジを各ノードに分配する。このとき同じ頂点が複数のノードにわたって存在するためそれらの一つをマスター頂点とし、残り頂点をコピー頂点とする。これらの頂点が値を同期するときに通信が生じる。

現実世界のグラフは skewed power-law distribution に従っている。skewed power-law distribution では頂点の次数の確率分布が定数 α を用いて式 (1) のように表される。

$$P(d) = d^{-\alpha} \tag{1}$$

edge-cut では次数の大きな頂点に対して多くのパーティションをまたぐエッジが存在する。この場合分散処理をするうえで以下の問題が発生する。

ノード間のバランス:　一部の頂点は数多くの隣接頂点を持っているため edge-cut ではノード間のワークロードに大きなインバランスが生じる恐れがある。Gather フェーズと Scatter フェーズの計算のコストは頂点の次数に比例しているのでワークロードのインバランスは計算時間のインバランスにつながってしまう。

通信:　高次数の頂点が原因で計算中の通信量が増える可能性がある。例えば、Pregel の場合高次数の頂点は多くのノードに多くの隣接エッジを持つため、これらの間での通信がボトルネックとなってしまう。

ストレージ:　各ノードおける頂点は隣接頂点に関する情報も保持しなければならない。高次元の頂点は隣接頂点が多いためすべての情報を 1 つのノードで保持することでメモリやストレージの容量を大量に消費してしまう。

vertex-cut は以上の問題を解決するうえで極めて有効である。PowerGraph では balanced p-way vertex-cut という概念が提案されている。balanced p-way vertex-cut ではエッジ $e \in E$ を p 個の計算機のどこかに割り当てる ($A(e) \in \{1, \ldots, p\}$)。このとき各頂点は複数個の計算機上にコピーを作ることになる ($A(v) \subseteq \{1, \ldots, p\}$)。式 (2) は各計算機にできる頂点のコピーを最小化することを目的にしている。しかしコピーの数を最小化するだけでは計算機間でワークロードのバランスが悪くなりかえって計算時間が遅くなってしまう場合がある。そこで式 (3) のようにエッジを一番分配された計算機がエッジを平等に計算機に分配したときの数の定数 λ 倍を下回るように分配することを条件としている。

$$\min_A \ \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{2}$$

$$s.t. \qquad \max_m \ |\{e \in E | A(e) = m\}| < \lambda \frac{|E|}{p} \tag{3}$$

2.2　GraphSteal

GraphSteal [9] はマイグレーション機能を搭載した vertex-cut の分散グラフ処理システムである。

PowerGraph や Pregel など提案されている多くの分散グラフ処理システムは使用するノードがすべて均一である前提で設計されている。そのため、初期分割をノード間で均一になるようにすることによってその後のグラフ計算もノード間でほぼ均一になるように行うことができる。しかし、異なるリソースを使用したクラスターやコアを共有したパブリッククラウドではメモリや CPU の性能において不均一が生じてしまう。このようなリソースによるノード間の不均一具合の大きさはアルゴリズムが原因による不均一具合よりも大きい傾向にある。このような、状況をメモリの使用量や計算時間のばらつきをモニタリングすることで判断しマイグレーションを実行するように設計されている。

スーパーステップのはじめに一つ前スーパーステップの実行時間などのモニタリング情報がリパーティショニングマネージャーに集約される。マネージャーは実行時間の平均値をとり平均値よりも早いノードをファストノード、遅いノードをスローノードに振り分ける。そして一番実行時間の遅いノードがファストノードの x 倍よりも遅い場合にマイグレーションを実行する。GraphSteal では実験の結果 x が 5 より大きい場合にマイグレーションが有効であることを示している。

GraphSteal ではグラフを CSR と CSC の形式で保持している。マイグレーションするペアが決まりマイグレーションするエッジ数が決まったときマイグレーションを行うノードでは CSR のリストの最後の部分から必要な数だけ切り取られ送られる。そのときに対応した CSC のリストも更新する。マイグレーション先では受け取ったエッジを CSR と CSC の適切な位置に導入することによって実際のマイグレーションは終了する。CSR と CSC の操作の例について図 2 に示す。

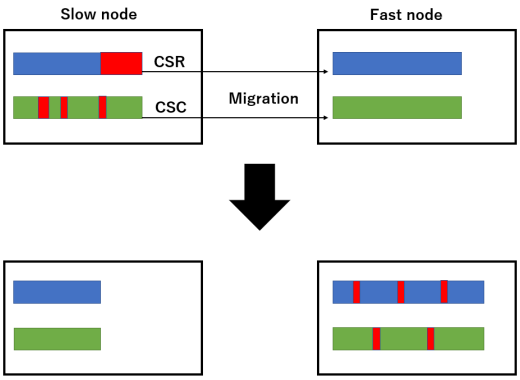

図 2　マイグレーションの例

3　提 案 手 法

3.1　既存手法の問題点

既存の分散グラフ処理システムにおいてマイグレーション機能を備えたものは複数存在する。それらはスーパーステップにおいて計算実行時間やメモリの使用量などをモニタリングし、その値を用いてマイグレーションを実行して不均一を解消しているが 2 つの問題が存在する。

マイグレーションエッジ選定

既存の多くのグラフ処理システムではマイグレーション時にエッジをランダムに選択して送信している。初期分割の精度が良くレプリケーションファクターが小さかったとしてもランダムなマイグレーションを行うことによって分割精度が落ちてしまう場合がある。また、マイグレーション時に精度が落ちないような閉じたサブグラフを見つけるのには時間的なコストがかかってしまいマイグレーションを行うコストに大きな影響を及ぼしてしまう。

マイグレーションを行うタイミング

既存の多くのグラフ処理システムではスーパーステップ全体の時間やメモリ量をモニタリングし、その値を用いてマイグレーションを行うためスーパーステップの最後にマイグレーションを実行している。しかし、1 スーパーステップ目でノード間に大きな不均一が生じた場合処理の一番早いノードはストラグラーを何もせずに待ち続けなければならない。また、マルチフェーズアルゴリズムのような各スーパーステップでアクティブになる頂点が変わる場合スーパーステップの最後にマイグレーションを行っても次のスーパーステップでは計算がノード間で均一化されるとは限らない。

そこで本研究では上記の 2 つの問題を解決する新しいマイグレーション手法を導入した分散グラフ処理システムを提案する。

3.2　提 案 手 法

3.2.1　概　　要

既存手法においてマイグレーションエッジをランダムに選んでいる問題に対しては、各ノードにおけるエッジの管理を閉じたサブグラフを用いることで解決する。既存のグラフ処理システムでは初期分割時に使用するノードと同じ数のサブグラフを作成する。提案手法では使用するノード数に対して十分大きな数のサブグラフを作成する。このように十分な数のサブグラフがあることでサブグラフを組み合わせ要求に合った数のエッジ群をつくることができ、マイグレーションの精度を高めることができる。サブグラフの粒度が高すぎると求められるマイグレーションのエッジ数をサブグラフの組み合わせで再現できない場合があるので、作成されるサブグラフの数と大きさには注意が必要である。また、これらのサブグラフは大きさが均一である必要はない。これらのサブグラフをノード間に均一に分配することで初期分割を行う。

サブグラフの分割には以下のことが求められる。

（1）　レプリケーションファクターが小さくなること

（2）　サブグラフの粒度が粗すぎないこと

サブグラフを動かしてもレプリケーションファクターが大きくならないように、最初の分割時のレプリケーションファクターが低くなることが求められる。また、サブグラフのサイズが大きすぎるとマイグレーション時に適切なサイズのグラフ群を選ぶことができないためマイグレーションを行えなくなる。

サブグラフ群を分配されたノードでは、これらのサブグラフを 1 つの単位としてグラフ計算やマイグレーションを行う。マイグレーションの実行時にはこれらのサブグラフを複数個送信することによってレプリケーションファクターの変化を最小限にし、また、適切なエッジ群を選択する時間もかからないようにした。

次に、マイグレーションを行うタイミングがスーパーステップの最後である問題に対してはスーパーステップの途中でグラフ計算と並行してマイグレーションを行うことで解決する。マイグレーションを行うタイミングは図 3 に示す。ローカルでの計算時に不均一が生じているが、それをリアルタイムで検知し、検知した段階でマイグレーションを実行している。このようにすることで、1 スーパーステップ目でノード間の実行時間に大きな差があったとしても処理速度の高いノードはストラグラーからワークロードをその時点で受け取って処理することができる。その後、マイグレーションがすべて終わったことを同期することで確認し次のステップへと進んでいる。マイグレーションの詳細は後述する。

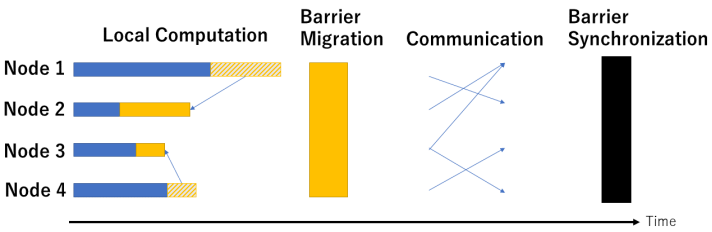

図 3　マイグレーションのタイミング

提案手法のアーキテクチャはマスター・ワーカーモデルである。提案手法ではノード群に対して 1 台をマスターノードと

して使用し、残りのすべてをワーカーノードとして利用している。マスターは計算初めのパーティションやマイグレーションに関わるタスクを行う。一方、ワーカーは与えられたサブグラフ群を使ってユーザーの定義したグラフアルゴリズムを実行する。また、マスターからマイグレーション命令があった場合は、指定されたノードに対してマイグレーションを実施する。アーキテクチャ全体イメージ図は図 4 に示す。黒色の四角はサブグラフを抽象的に表しており各ワーカーが様々なサイズのサブグラフを保有している。マスターとワーカーが通信し合うことによってサブグラフ単位でのリアルタイムなマイグレーションを実現している。

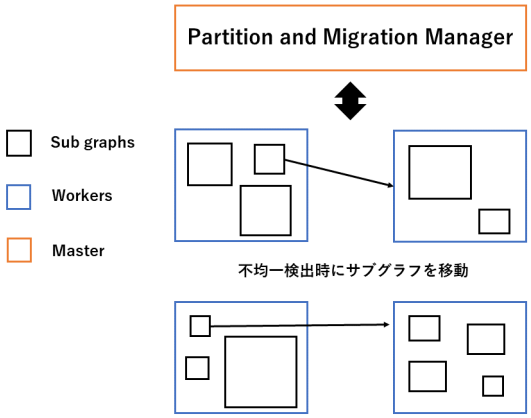

図 4　提案手法全体図

3.2.2　サブグラフへの分割

提案手法ではサブグラフへの分割を行う上で gscarf [10] を利用する。しかし、gscarf は edge-cut であるためこれを vertex-cut に変換する必要がある。各サブグラフがもつエッジ数を記録しておきサブグラフをまたぐエッジがある場合はそのエッジをエッジ数の少ないサブグラフに追加する。エッジが追加されたサブグラフにはコピー頂点が作成される。これをすべてのサブグラフをまたぐエッジに対して行うことによって vertex-cut を実現する。

3.2.3　グラフ構造

マイグレーションを効率よく行う上でグラフの構造をメモリ上で連続して表現することによってマイグレーション時のシリアライズとグラフ受取時のデシリアライズの時間を短縮することができる。そこで、提案手法では図 5 のようにグラフを表現する。グラフ構造は主に 4 つのパートから構成されている。Graph info ではサブグラフ ID やサブグラフの中に何個エッジがあり何個頂点があるかの情報を保持している。頂点とエッジの関係を表すために本研究では CSR と CSC を用いる。GAS モデルでは頂点に入ってくるエッジを中心として計算を実行したり、頂点を出ていくエッジを中心として計算を実行する場合があるため CSR と CSC のどちらの形式も保持している。また頂点をイテレーションする場合もあるので頂点 id のセットを保持している。グラフ構造は頂点同士の関係を表しているだけで実際の頂点の値は別のデータ構造で保持している。

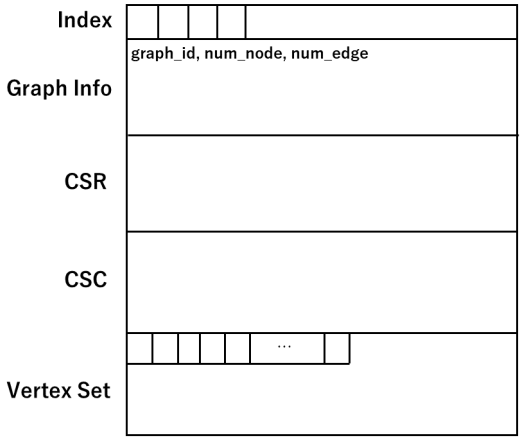

図 5　グラフの表現

3.2.4　計 算 手 順

提案手法の計算手順は GAS モデルをもとにしている。GAS には計算時間のモニタリング機能とマイグレーション機能がないのでそれらの機能を組み込んでいる。以下に計算手順を示す。

Gather:　すべての頂点は隣接頂点から値を収集しその値を用いて自身の値を更新する。GAS モデルの場合はこのステップでノード間の通信が発生するが提案手法のモデルではここで発生する通信を次のステップのはじめにまとめている。各ノードはユーザーが指定したエッジ数 P について先に計算を進め、エッジ数 P を処理するのに要した時間をモニタリングしておく。ここでモニタリングされる時間は、保有しているワークロードの量、アルゴリズムの性質によるアクティブな頂点の偏り、CPU やメモリなどのリソースに依存する。記録した処理時間と現在保有している総エッジ数をマスターノードに集約しマスターはマイグレーションをするかどうかの決定及びマイグレーションを実行する場合はマイグレーションペアとマイグレーションするエッジ数を計算し、ワーカーにブロードキャストする。その後ワーカーはマイグレーションが必要な場合はマイグレーションを実行し、必要でない場合は残りのエッジの処理を進める。

Apply:　はじめに Gather ステップで更新した情報をコピー頂点がマスター頂点に集約する。マスター頂点では集約した値を用いてグラフ全体の頂点の値として自身の値を更新する。その値をコピー頂点にブロードキャストし、コピー頂点とマスター頂点の同期が終了する。apply 終了時にはすべてのワーカーに存在するノードはノード id が同じであるならば同じ値を保持している。

Scatter:　すべての頂点は Apply ステップで同期した値を隣接頂点に送る。値をもらった頂点はその値を用いてユーザーの定義したタスクを実行する。

3.2.5　マイグレーション

まず初めにマイグレーションを行うタイミングについて記述する。スーパーステップの Gather 実行時に処理の一番速いノードの id を f 処理の一番遅いノードの id を s とする。また Gather にかかった時間をそれぞれ T_f, T_s とする。スーパステップの最後にマイグレーションを行った場合はノードの実行時間の不均一具合が大きいと $T_s - T_f$ の時間 T_f は何も処理を

することなく待機する無駄な時間が生じてしまう。そこで提案手法では Gather 時に不均一を検知し Gather 時にグラフ計算と並行してマイグレーションを行う。

詳細の流れは図 6 で示す。それぞれの矢印は処理の順番及びスレッドを表しており、赤色の矢印は通信を表し赤色の網線はワーカー全体での通信を表している。この例では Worker n がストラグラーとしてグラフを送信し、それ以外のワーカーはグラフを受信する。グラフを送るノードは送るグラフを選択するときに保持しているグラフ群の構造に変更を加えるためグラフ計算と並列することが不可能である。そのため先に送るグラフを選択してから、グラフ計算を続行している。また、グラフを受け取ったノードは元々持っていたグラフ群と受け取ったグラフ群を区別するため元々持っていたグラフ群の処理が終わった後に受け取ったグラフ群の処理をする。マイグレーション終了後の Apply ステップでコピー頂点とマスター頂点が同期するためこの時点で頂点がどこのワーカーに存在するかのハッシュテーブルの更新を行う必要がある。

頂点の位置管理のためのハッシュテーブルはコピー頂点用とマスター頂点用の 2 つが存在する。コピー頂点のハッシュテーブルはキーを頂点 id としバリューは存在するワーカー id のリストになっている。コピー頂点のハッシュテーブルの例を表 1 に示す。頂点 1 はワーカー 1,2,3 に存在し、頂点 2 のコピー頂点は存在せず、頂点 3 はワーカー 1 に存在することを示している。マスター頂点のハッシュテーブルは頂点 id をキーとしマスタ頂点が存在すワーカー id をバリューにしている。マスター頂点は各ワーカーに 1 つしか存在しないのでバリューはリストにはならない。マスター頂点のハッシュテーブルの例を表 2 に示す。頂点 1 と 2 のマスター頂点はワーカー 1 に存在し、頂点 3 のマスター頂点はワーカー 2 に存在することを示している。すべてのワーカーはこれらのハッシュテーブルをローカルで保持しており、Apply のコピー頂点とマスター頂点の同期時にハッシュテーブルを用いて通信を行う。

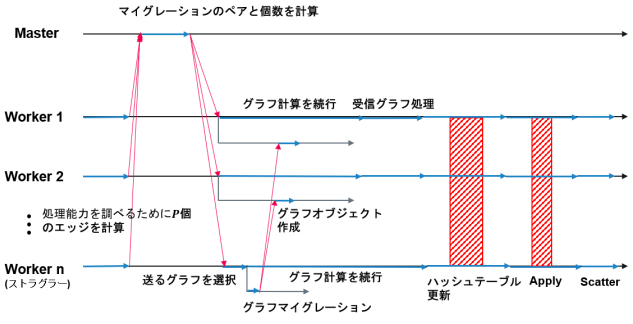

図 6　マイグレーションのシーケンス

表 1　コピー頂点のハッシュテーブルの例

頂点 ID	ワーカー ID のリスト
1	[1,2,3]
2	[]
3	[1]

表 2　マスター頂点のハッシュテーブルの例

頂点 ID	ワーカー ID
1	1
2	1
3	2

次にマスターノードで行う処理について記述する。各ワーカーは Gather ステップのはじめに P 個のエッジに到達するまで Gather 処理を進める。そして処理したエッジ数が P に到達した時点で、マスターノードに P 個のエッジ処理にかかった時間 T_i と現在そのノードが持っているエッジ数 $|E_i|$ を送信する (i はノードの id)。

マスターノードでは、ワーカーの処理能力 μ_i(sec/個) を $\mu_i = \frac{T_i}{P}$ で求めることができるため、ワーカーにおける Gather の実行時間は $\mu_i * |E_i|$ で表現される。このとき一番処理の遅いノードと一番処理の速いノードの差 $\mu_s * |E_s| - \mu_f * |E_f|$ がユーザーの指定する定数 γ より小さい場合はノード間に不均一が生じていないものとしてマイグレーションを行わないためマイグレーションが発生しないことをワーカーにブロードキャストする。マイグレーションを行う場合はマイグレーションを行うペアとエッジ数を計算する。

マイグレーションを行うことで各ワーカーの保有エッジ数がそれぞれ $|E'_i|$ になったことを考える。このとき式 4 が成り立てばノード間でバランスがとれたと言える。式 4 を解くことによって、計算時間が均一になるようなエッジ数が式 5 のように求めることができる。

$$\mu_1 * |E'_1| = \mu_2 * |E'_2| = \cdots = \mu_n * |E'_n| \quad (4)$$

$$where.\ \ |E'_1| + |E'_2| + \cdots + |E'_n| = |E|$$

($|E|$ はグラフ全体のエッジ数)

$$|E'_i| = \frac{|E|}{\mu_i * (\frac{1}{\mu_1} + \cdots + \frac{1}{\mu_n})} \quad (5)$$

$|E'_i| - |E_i|$ を求めることでマイグレーションすべきエッジ数を算出できる。例えば、この値が正の場合はグラフを受け取る側のノードとなり、負の場合はグラフを送る側のノードとなる。

また、Gather ステップ時に各ワーカーは P エッジを処理し、その時間をモニタリングするためすべてのワーカーは最低でも P エッジを保有する必要がある。そのため、マスターがエッジ数 P を下回るようなマイグレーション指示を出したとしても、ワーカーはエッジ数 P を下回る時点でマイグレーションを終了するように設計されている。よって、マイグレーション予測と実際のマイグレーションの挙動に差が生じることがある。

次にマイグレーションを行う数をもとにマイグレーションを行うペアを作成していく。マイグレーションを行うべきエッジ数をもとにエッジを送るノードとエッジを受け取るノードの 2 グループにノードを分割し、それぞれの中でマイグレーションエッジ数を昇順にソートする。それぞれのグループを先頭からイテレーションする。このとき送るべきエッジ数を $\#send$、受け取るべきエッジ数を $\#recv$ とすると、マイグレーションするエッ

ジ数は $max(\#send, \#recv) - min(\#send, \#recv)$ で求めることができる。つまり、受け取ることができるエッジ数が送るべきエッジ数を超えている場合はすべてのエッジを受け取ることができるが、受け取ることができるエッジ数が送るべきエッジ数よりも小さい場合は送るべきエッジの一部しか受け取ることはできない。マイグレーションするエッジ数が決まったらそれらの値を $\#send$ と $\#recv$ から差し引き、値を更新して、値が 0 になった方はイテレーションを次に進める。この作業をすべてが 0 になるまで繰り返すことでペアを作成してく。図 7 の例ではマイグレーションのペアが、$(5 \rightarrow 3), (4 \rightarrow 3, 1), (2 \rightarrow 1), (6 \rightarrow 1)$ のように求めることができる。

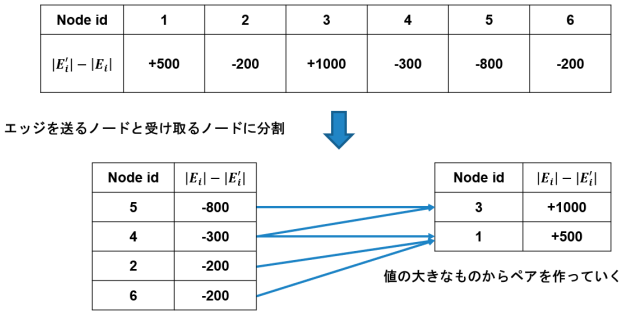

図 7　マイグレーションペアの作成例

4　評 価 実 験

評価実験では提案手法と比較手法を用いて PageRank アルゴリズムを 15 イテレーション行い、それらの全体の実行時間を比較することで評価する。1 イテレーションの時間はストラグラーノードの実行時間と同じになる。つまり 15 スーパステップでの合計時間は各スーパーステップのストラグラーノードの実行時間の合計となる。これらの時間を測定するときに、全体のグラフをサブグラフに分割する時間及びノードにおいてサブグラフをメモリ上にロードする時間は実行時間に含めない。また、キャッシュヒット率や OS のタスクスケジューリングによって計算時間が変化するためそれぞれの手法は 5 回実行し、その平均値を算出する。実験結果の棒グラフに使用しているエラーバーは 68%の標準誤差を表したものである。

また、提案手法にで用いているパラメータについては、$P = 5000000$ と $\gamma = 5$ を使用した。

4.1　実験の前提

4.1.1　比 較 手 法

提案手法の性能を評価するために複数個の比較手法を同じ環境に実装した。表 3 に比較手法の表記名と詳細を示す。

表 3　比 較 手 法

手法名	詳細
gscarf	マイグレーションを行わない
gscarf + mig	マイグレーションを最後に行う。 既存手法のマイグレーションタイミングを再現している。
gscarf + random_mig	マイグレーションするエッジ群をランダムに選択する。 既存手法のマイグレーションエッジ選択方法を再現している。

4.1.2　実 験 環 境

クラスタ型スーパーコンピューターの TSUBAME [12] を使用して実験を行う。TSUBAME のスペックは表 4 に示す。物理コア 4 つと 30(GB) を 1 つのノードとして使用し、5 台、9 台、17 台で実験を行う。ノードの個数に関係なく 1 台をマスターノードとして使用し、残りの台数をすべてワーカーとして使用する。またこれらのノード間の通信には OpenMPI [13] のバージョン 3.1 を利用する。

表 4　TSUBAME のスペック

CPU	Intel Xeon E5-2680 V4 Processor (Broadwell-EP, 14 コア, 2.4GHz) × 2 Socket
RAM	256GiB
Storage	Intel DC P3500 2TB (NVMe, PCI-E 3.0 x4, R2700/W1800)
Network	Intel Omni-Path 100Gb/s × 4

各ノードのリソースの不均一を再現するために浮動小数点の割り算を無限に行うスレッドを用意し、このスレッドを無効スレッドと呼ぶ。ワーカーノードのうち 1 台を無効スレッド 200 個、ワーカーノードのうち他の 1 台を無効スレッド 400 個起動することでストラグラーを作成する。その他のノードでは無効スレッドは起動していない。本研究の実験では無効スレッドとグラフの計算を行うスレッドが使用するメモリ量を合わせてもメインメモリの大きさを超えないように考慮されているためスワップ領域を使用することはない。

4.2　データセット

データセットは orkut [15] のソーシャルネットを使用する。表 5 にデータセットの詳細を示す。また、orkut を gscarf で分割したときのクラスターに関する情報を表 6 に示す。

表 5　orkut 詳細

頂点数	3072441
エッジ数	117185083
直径 (頂点間の最短距離の最長)	9

表 6　orkut の gscarf による分割結果

サブグラフ数	18659
平均保有エッジ数	27454
最大保有エッジ数	6003500
保有エッジ数の中央値	24

4.3　実 験 結 果

図 8 に実験結果を示す。実験の結果 5 ノードの場合提案手法がマイグレーションをしない場合に対して 2.80 倍、スーパーステップの最後にマイグレーションを行った場合に対して 1.25 倍高速化し、9 ノードの場合提案手法がマイグレーションをしない場合に対して 2.34 倍高速化し、スーパーステップの最後にマイグレーションを行った場合に対して 1.30 倍高速化し、17 ノードの場合提案手法がマイグレーションをしない場合に対して 1.73 倍高速化し、スーパーステップの最後にマイグレーショ

ンを行った場合に対して 1.24 倍高速化した。

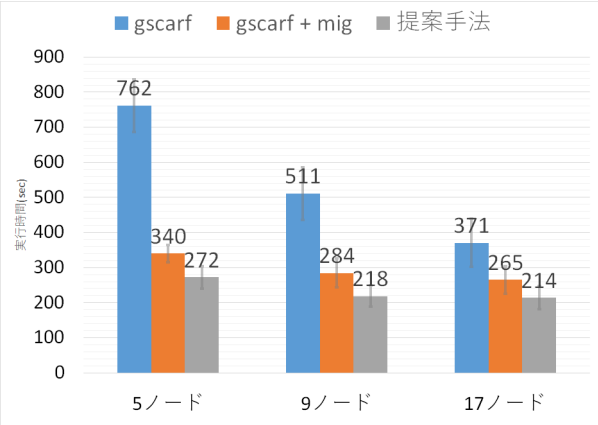

図 8　提案手法の実行結果

表 7 にスーパステップ 5 までの各ステップの実行時間を示す。提案手法は 1 スーパステップ目で不均一を計算中に検知し計算の並行してマイグレーションを実施することで 1 スーパステップ目の実行時間が 5 ノードの場合 1.34 倍高速化し、9 ノードの場合 1.38 倍高速化し、17 ノードの場合 1.22 倍高速化した。9 ノードの場合を考えると、マイグレーションを行わない場合 Gather 最速のノードは 0.1(sec)、最遅のノードは 13.6(sec) であり 1 スーパステップ目で最速のノードは最遅のノードを待たなければならなかったが、それを待たずにマイグレーションを行うことで高速化している。また、Gather に続く Apply フェーズでも処理能力のバランスがとれることによって 1 スーパーステップ目の高速化に寄与している。今回のアルゴリズムでは Scatter フェーズの計算はないが、Scatter フェーズに重い計算がある場合は、提案手法がより有効になると考える。

表 7　5 スーパーステップの各実行時間 (単位:sec)

	スーパーステップ	1	2	3	4	5
5 ノード	gscarf + mig	177	11.7	11.1	12.0	11.3
	提案手法	131	9.87	10.12	10.20	10.02
9 ノード	gscarf + mig	125	13.4	11.9	11.7	10.6
	提案手法	90.2	9.49	9.39	9.03	9.30
17 ノード	gscarf + mig	105	12.6	11.2	11.6	11.2
	提案手法	86.0	9.57	9.13	8.93	9.07

9 ノードの場合のマイグレーション前後での各ノードの保有エッジ数の変化を表 8 に示す。ノード ID が 7 のノードは無効スレッドが 200 個動いていて、ノード ID が 8 のノードでは無効スレッドが 400 個動いている。提案手法にあるマイグレーションエッジ数とペアを決めるアルゴリズムを用いることでノード 7 とノード 8 がストラグラー群であることを検知しマイグレーションを行えていることがわかる。

表 8　マイグレーションによるエッジ数の変化 (単位:10^6 個)

ノード ID	mig 前	エッジ移動数の予測	mig 後
1	16	8.3	24
2	14	2.3	17
3	15	3.9	18
4	15	2.5	21
5	15	6.9	22
6	15	4.3	15
7	15	-14	0.51
8	15	-14	0.51

図 9 にマイグレーションにかかるコストの詳細を示す。マイグレーションのコストは CPU と MPI の時間、またマイグレーションそのものにかかる時間とハッシュテーブルの更新時間に分けることができる。

ノード数が増えるごとに CPU 時間が小さくなっているのがわかる。ノード数が増えることでストラグラーが保有するエッジ数が減りマイグレーション時の送信エッジ数が減っているためストラグラーのマイグレーション準備の時間的コストが減るためだと考えられる。また、どのノード数においてもハッシュテーブルの更新にかかるコストが大きいため、ハッシュテーブルの管理を工夫する必要がある。

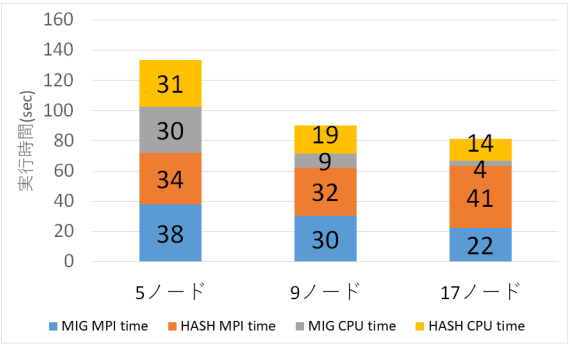

図 9　マイグレーションコスト

次に閉じたサブグラフ単位でのマイグレーションの評価を行う。図 10 に gscarf+mig と gscarf+random_mig のマイグレーション後のレプリケーションファクターの違いを示す。赤線はマイグレーション前のレプリケーションファクターを示している。閉じたサブグラフのままマイグレーションを行うことによってマイグレーション後のレプリケーションファクターがランダムにエッジを送信する場合と比べて 5 ノードの場合 0.16、9 ノードの場合 0.16、17 ノードの場合 0.39 小さくなった。レプリケーションファクターが小さくなるように保つことによって図 11 に示すようにマイグレーション後の Apply の時間もランダムにエッジを選びマイグレーションする場合に比べて早くなっていることがわかる。

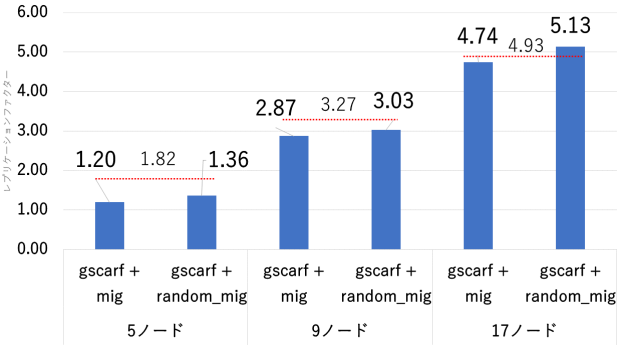

図 10　レプリケーションファクターの変化

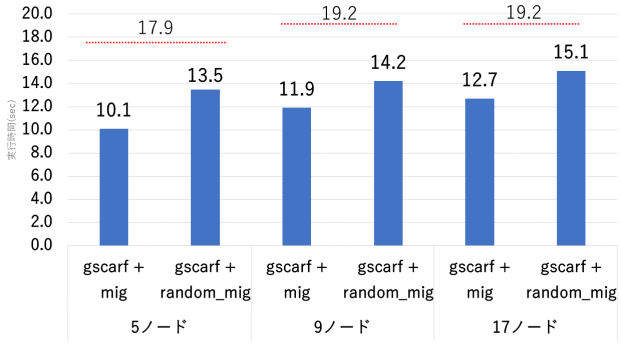

図 11　Apply 時間の変化

5　おわりに

本研究では、リソースが不均一な環境における同期的な分散グラフ処理において、マイグレーション機能付きの既存のグラフ処理システムのマイグレーションエッジの選び方とマイグレーションを行うタイミングの問題点を解決するために、グラフをサブグラフ単位で管理して、グラフ計算と並行してマイグレーションを行うことによって計算速度を向上させる手法を提案し、評価を行った。

実験を行う上で計算するアルゴリズムはページランクを使用し、5 ノード、9 ノード、17 ノードを用いて計算を実行した。不均一な環境を再現するために複数のノードでグラフ計算とは関係のないスレッドを複数個起動した。

実験の結果、提案手法がマイグレーションを行わない場合に対して 2.34 倍高速化し、マイグレーションをスーパーステップの最後に行った場合に対して 1.30 倍高速化した。1 スーパーステップ目の Gather 時に最速ノードが最遅ノードの実行の終わりを待たずにマイグレーションを行えることと、1 スーパーステップ目の Apply フェーズでそれぞれのノードでの処理速度が均一化されたことが高速化した要因である。また、エッジをランダムに選びマイグレーションした場合に比べて提案手法のマイグレーション後のレプリケーションファクターが 0.39 小さくなる結果になり、スーパーステップの実行時間削減に寄与した。

今後の課題としては、マイグレーションのコストの削減方法の模索やより大きな台数での実験などが挙げられる。

謝　辞

本研究の一部は、JSPS 科研費 (18H03242, 18H03342, 19H01138A) の助成を受けたものである。

文　献

[1] Jeffrey Dean and Sanjay Ghemawat, ”MapReduce: Simplified Data Processing on Large Clusters”, 6th USENIX Symp. on Operating Syst. Design and Impl, pp. 137–150, 2004.

[2] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, ”Pregel: A System for Large-scale Graph Processing”,Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD ’10),ACM, New York, pp. 135-146, 2010.

[3] Leslie G. Valiant, ”A bridging model for parallel computation”, Communications of the ACM, Volume 33 Issue 8, 1990.

[4] R. R. McCune, T. Weninger, and G. Madey, “Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing, ACM Comput. Surv., vol. 48, no. 2, p.p. 1–39, Oct. 2015.

[5] http://giraph.apache.org (2021 年 2 月 10 日アクセス)

[6] S. Salihoglu and J. Widom, ”GPS: A Graph Processing System”, SSDBM ’13, p.p. 22:1–22:12, 2013

[7] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, Giorgio Iacoboni,“HDRF: Stream-Based Partitioning for Power-Law Graphs”, CIKM’15. Melbourne, 2015.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, ”PowerGraph: Distributed graph-parallel computation on natural graphs”,Proceedings ofthe 10th USENIXConference on Operating Systems Design andImplementation (OSDI ’12), 2012.

[9] D. Kumar, A. Raj, and J. Dharanipragada, ”Graphsteal: Dynamic re-partitioning for efficient graph processing in heterogeneous clusters”, 2017 IEEE 10th International Conference on Cloud Computing, IEEE, pp. 439–446, 2017.

[10] Hiroaki Shiokawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. ”Scaling Fine-grained Modularity Clustering for Massive Graphs.”, Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019), pp. 4597–4604, 2019.

[11] Newman MEJ, Girvan M, ” Finding and evaluating community structure in networks”, Phys Rev E 69, 026113, 2014.

[12] https://www.t3.gsic.titech.ac.jp/ (2021 年 2 月 10 日アクセス)

[13] https://www.open-mpi.org/ (2021 年 2 月 10 日アクセス)

[14] S. Brin and L. Page, ”The anatomy of a large-scale hypertextual Web search engine”, Computer Networks and ISDN Systems, pp. 107–117, 1998.

[15] http://www.orkut.com/index.html (2021 年 2 月 10 日アクセス)