メニーコア環境におけるグラフ処理エンジンの高速化

伊藤　竜一† 　藤森　俊匡†† 　鬼塚　真††

† 大阪大学工学部電子情報工学科　〒 565–0871 大阪府吹田市山田丘　2–1
†† 大阪大学大学院情報科学研究科　〒 565–0871 大阪府吹田市山田丘　1–5
E-mail: †{itou.ryuichi,fujimori.toshimasa,onizuka}@ist.osaka-u.ac.jp

あらまし　近年, ネットワークやストレージ, センサなどの進歩により, 大規模なグラフ構造のデータが一般的になり, それを如何に効率的に処理するかが技術課題となっている. 特に, グラフ処理の基盤となるエンジンの高速化は重要な要素である. しかしながら, 並列度が増加すると, 同時実行制御がボトルネックとなり, 実行効率が悪化することが知られている. そこで, 本稿では, 並列環境として単一マシンメニーコア環境に焦点を当て, 同時実行制御に Compare-and-Swap 命令や Hardware Transactional Memory を用いて最適化を図ることで並列化のオーバーヘッドを減らし, 高速な処理を実現した. ベースとして用いたグラフ処理エンジンである GraphChi と比較して, 同時実行制御に Compare-and-Swap 命令を利用した制御や Hardware Transactional Memory を用いた手法に置き換えることで, 処理全体にかかる時間を最大約 19%, 同時実行制御にかかる時間を最大約 76% 短縮することに成功した. また, Hardware Transactional Memory の利用最適化を行うことで, 更に最大約 8% の高速化に成功した.

キーワード　グラフ処理エンジン, メニーコア, 同時実行制御, Compare-and-Swap, Hardware Transactional Memory

1. はじめに

身の回りの様々な事象がデータとして蓄積されることが一般的になってきたことで, 結果として得られるソーシャルグラフや Web グラフといったグラフ構造のデータが大規模化している. 例えば, 2015 年 9 月時点の Facebook のユーザ数は 15 億 5000 万人にも及び [1], その中でのユーザ同士の繋がりやポストとそれに対するコメントなどが, 大規模且つ複雑なネットワークを形成している. このような莫大なグラフデータ源から, 有益な情報を如何に迅速に取り出すかが技術課題となっている. これまではプロセッサの動作周波数向上により処理速度を上げることが可能であったが, その成長が鈍化しつつあり, 増加するコアを活かした並列処理による高速化の重要性が増している.

並列処理へのアプローチとして, 単一マシン, 限られたコア, 共有メモリ環境であることを利用する方法がある. 実行環境のセットアップや管理が簡単であり, データのやり取りにネットワークを介することがないのでレイテンシが少ないといった利点がある一方, CPU の数が比較的少数で計算能力に制限があり, 巨大なデータを一度に全てメモリ上にロードすることができないといった欠点が存在する. そのため, 限られた計算資源で, memory-based なシステムと比較すると非常に低速である disk-based なシステムとなる. 単一マシン用グラフ処理エンジンの既存の研究として, GraphChi [2] や TurboGraph [3], VENUS [4], GridGraph [5], NXGraph [6] が挙げられる. いずれの研究も, アクセスの局所性を生かすデータ構造とアルゴリズムを設計することで, コアの利用率および IO 操作量の削減して高速化を図っている.

グラフ処理高速化の課題の一つとして, 更新処理時の一貫性を保つためにコストがかかるという問題が挙げられる. 一般的に, 現実世界で見られるようなグラフデータの多くには, 少数ながら, 多数の辺が集中する頂点が含まれることが知られている [7]. このような頂点は, 隣接頂点の多さから並列処理する際に更新操作の資源として参照される回数が非常に多くなるため, 一貫性を保つためのコストが高く, ボトルネックになりやすい. このような頂点に関する計算タイミングを分散させることで, 競合を解決するコストを抑えることができるが, グラフデータの局所性を十分に活かすことができなくなる. 一方, 計算をシリアル実行にすることで局所性を最大限に活かすことができるが, 並列環境を活かすことができなくなる, というトレードオフが発生する. 本稿の提案手法では同時実行制御に Lock-free である Compare-and-Swap 操作や Hardware Transactional Memory [8] を適用することで並列環境と局所性の両方を維持し, グラフ処理の高速化を図った. 並列計算モデルとして, 同時実行制御を必要とするがデータの競合発生率は低い, という Lock-free な手法で用いられる投機的実行と相性の良い特徴を持つ Gather-Apply-Scatter computation model [9] を利用する. 更に, Gather-Apply-Scatter computation model を構成する演算の一つである Sum 演算に関する部分が可換モノイドであるという性質に注目し, Sum 演算を多段階に分解することで投機的実行の失敗を抑制する最適化を行った.

本稿では Gather-Apply-Scatter computation model を利用できる単一マシングラフ処理エンジンとして, GraphChi をベースに提案手法の実装し, 評価実験を行った. 同時実行制御に従来の Mutex を用いた手法を Compare-and-Swap を用いた手法に置き換えることで処理全体にかかる時間を最大約 19%, 同時実行制御にかかる時間を最大約 76%, Hardware Transactional Memory を用いた手法に置き換えることで, 処理全体にかかる時間を最大約 10% , 同時実行制御にかかる時間を最大約

53% 短縮することに成功した. また, Hardware Transactional Memory の利用最適化を行うことで処理全体にかかる時間を更に最大約 8% 短縮することに成功した.

本稿の構成は以下の通りである. 2 章 で本稿の前提となる知識について述べる. 3 章 で提案手法を詳説し, 4 章 で提案手法の実験とその評価・分析を行う. 5 章 で関連研究を紹介し, 6 章で本稿全体を俯瞰しまとめる.

2. 前提知識

本章では, グラフ処理における同時実行制御で必要となる前提知識と, 提案手法のベースに用いたグラフ処理エンジンと計算モデルについて述べる. 2.1 節 では同時並列的処理におけるデータの一貫性を保つためのモデルについて, 2.2 節 では 2.1 節 を実現するために用いられる同期プリミティブについて, 2.3 節 ではグラフ処理で用いられる計算モデルについて, 2.4 節 ではベースに用いたグラフ処理エンジンである GraphChi について述べる.

2.1 Data consistency model

従来はシングルマシン―シングルコア―シングルスレッドでの計算が主流であったが, 近年, コア単体での性能向上速度が必要とされる処理の大規模化に追いつかず, マルチマシン―マルチコア―マルチスレッドでの計算へと移行してきている. 大規模グラフ処理のように, 一度に巨大な共有資源を扱う処理をマルチマシン―マルチコア―マルチスレッドで, データの一貫性を保ちながら計算するためには, **同時実行制御** を行う必要がある. そのような処理では, 一度の更新処理に必要となる共有資源は共有資源全体からするとごく一部であることが多いため, 同時実行制御の対象範囲を変化させることで効率化を図ることができる. また, 多少の計算精度を犠牲にすることで処理をより高速化することができる場合もある. この, 同時実行制御の粒度や方法を定めたものを Data consistency model と呼ぶ.

グラフ処理で利用される同期実行で制御を行うための具体的な Data consistency model として, Bulk Synchronous Parallel(BSP) [9] がある. これは, 処理に superstep という同期単位を定め, superstep ごとに設けられた barrier のタイミングで全ワーカーのデータを一斉に同期させるというモデルである. 値を参照する場合は前回の superstep で得られた値を利用することで, 他のワーカーによるデータの変更の影響を受けず, 計算精度と一貫性を保つことができる. しかしながら, barrier のタイミングはすべてのワーカーが処理を終えたときと決められているため, 各ワーカーへ分散されたタスクに偏りが生じていると処理時間の長いワーカーに全体の処理時間が左右されてしまう問題がある.

一方, 分散グラフ処理エンジンである GraphLab などで用いられている完全な非同期実行型の Data consistency model として, vertex consistency / edge consistency / full consistency [10] がある. vertex consistency は注目頂点の値のみ, edge consistency は注目頂点と隣接する辺のみ, full consistency は注目頂点と隣接する辺と頂点のみに同時実行制御を行う. 当然ながら, 同時実行制御対象外の辺や頂点の値は任意のタイミングで他のワーカーから書き換えられてしまう可能性があるため, 計算精度が下がるが, 制御対象が少ないほど同時実行制御によるオーバーヘッドが抑えられ, また, 各ワーカーへのタスク分散に全体の実行時間が影響を受けなくなり, 実行時間を短縮することができる.

2.2 同時実行制御と同期プリミティブ

Data consistency model を導入するなどしてデータの一貫性を保つためには, 共有資源を他のワーカーから操作されないように同期プリミティブを利用して同時実行制御をする必要がある. この同時実行制御には多くの手法が存在しており, 大きく分けて Lock-based と Lock-free な手法があり, それぞれ 2.2.1 節 と 2.2.2 節で述べる.

2.2.1 Lock-based

Lock-based な手法として, 同期プリミティブに Lock 命令を使ったものが知られている. 特定の共有資源を利用する前にロックを取得することで, 他のワーカーが同時に同じ共有資源にアクセスしてしまうことを防ぐ排他的な方法である. コストがかかりやすいという問題点があるが, 仕組みが単純なため用いられることが多い.

2.2.2 Lock-free

Lock-based な手法とは異なり, ロックを用いない, つまり, 他のワーカーに影響を与えない Lock-free な手法が知られている. この手法は同期プリミティブに専用の機械語命令を用いることで, 大きく分けて 2 つのアプローチから利用されている.

1 つ目は特定ドメインに特化した Lock-free データ構造を作るというアプローチである. 同期プリミティブに Compare-and-Swap(CAS) という機械語命令を用いる. ロックを取ることなく値の変更を検知することでデータの一貫性を保つことが可能だが, 複数の値に対して一貫性が必要な場合, 実装が複雑になってしまうという問題がある.

2 つ目のアプローチとして, Transactional Memory(TM) [8] が挙げられる. TM とは, データベース分野におけるトランザクションをメモリに応用したものである. 一連の処理を行う際に, トランザクション内のすべての処理が完了するまでメモリの共有部分に反映させず, トランザクション内のすべての処理が終わった時点でトランザクション内で使われたデータとメモリの共有部分のデータをチェックし, T) 一貫性が保てる場合はトランザクションの結果をメモリに反映 (Commit) し, トランザクションが成功したことを通知する; F) 一貫性が保てない場合はトランザクションの結果を破棄 (Abort) し, トランザクションが失敗したことを通知する. このように一連のデータの読み込み・書き込みを 1 つのトランザクションとして扱うことで原子性・一貫性を, 各トランザクションを個別に管理することで独立性を保証する.(注1) TM を利用することで, 読み込み側責任で悲観的となる Lock-based な手法とは違い, 書き込み側責任で楽観的に処理できるため, トランザクションの Abort が発生しない限りでは同時実行制御のスループット向上を望むことができる. また, トランザクションという単位で扱うた

(注1)：揮発性メモリ上で行われるため, 耐久性については保証されない

め, 複数のトランザクションをまとめて 1 つのトランザクションとして利用することが容易であるという利点が挙げられる. TM の実装として, Software Transactional Memory と Hardware Transactional Memory がある. Software Transactional Memory(STM) とは, TM をソフトウェアレベルで実装したものである. 同期プリミティブとして前述の CAS 命令が用いられ, 1 つ目に挙げたアプローチの一種でもある. 一方, Hardware Transactional Memory(HTM) とは, TM をハードウェアレベルで実装したものである. 現時点では, Intel Haswell [11], IBM Blue Gene/Q [12]・POWER8 [13]・zEnterprise EC12 [14] などの一部のプロセッサに, それぞれ独自の命令として実装されている. STM と比較して, *) トランザクションの途中経過はすべて高速なキャッシュ上で管理される; *) ユーザランドにオブジェクトを生成しない; 等, ハードウェア資源を有効に利用できるためスループットが高い.

2.3 Gather-Apply-Scatter computation model

グラフ処理で用いられるユーザ定義の計算アルゴリズムを記述する方法は処理エンジンにより様々である. しかしながら, 前述の Data consistency model を効率的に実現するため, ユーザが計算アルゴリズムを合成可能な形に分解することを前提としたインターフェースが利用されることが多い. その具体的な計算モデルとして, PowerGraph [9] で提案され, 他のグラフ処理エンジンでも類似モデルが使われることが多い Gather-Apply-Scatter computation model(GAS) が知られている. GAS では, 辺から値を取得し, 集約するための **Gather** と **Sum** , 頂点の値を更新するための **Apply** , 頂点の値から隣接する辺の値を更新するための **Scatter** と呼ばれる 4 つの演算で 1 つの計算アルゴリズムを定義する. GAS の処理フローの擬似コードを Algorithm 1 に示す. 1〜5 行目の `foreach` ループで Gather と Sum, Scatter 演算を行い, 全ての辺に対して値の割り当てと取り出しを完了させてから, 6〜8 行目の `foreach` ループで一括して Apply を実行し, 頂点データの更新をする. なお, これを非同期に計算する場合は, Sum 演算のみ同時実行制御の対象とすれば全体のデータの一貫性を保つことができ, 並列処理の恩恵を受けやすいという特徴を持つ.

表 1　Algorithm で使用されている主な記号の定義

$G_i(V,E)$	i イテレーション目の頂点集合 V, 辺集合 E からなるグラフ G
$get\_vertex(i)$	頂点番号 i から頂点データを取得する関数
$e.cur\_val$	BSP における現イテレーションの辺 e の値
$e.prev\_val$	BSP における前イテレーションの辺 e の値
$v.acc$	頂点 v の Sum 演算におけるアキュムレータ

2.4 GraphChi

本稿では, 提案手法を GraphChi [2] に組み込むことで, メニーコア環境に対応した高速なグラフ処理エンジンを提案する. そこで本節では, ベースとなるグラフ処理エンジンである GraphChi について概説する. グラフ処理を行うユーザプログラムのために, GAS に基づくインターフェースと, 1 つの更新関数を定義して利用するインターフェースを提供している.

Algorithm 1 Gather-Apply-Scatter computation model

Input: $G_n(V,E)$
Output: $G_{n+1}(V,E)$
1: **foreach** $e \in E$ **do**
2: $\quad v_{dest} \leftarrow get\_vertex(e.out)$
3: $\quad e.cur\_val \leftarrow Scatter(e)$
4: $\quad$ **lock**
5: $\quad\quad v_{dest}.acc \leftarrow Sum(v_{dest}.acc, Gather(e.prev\_val))$
6: $\quad$ **unlock**
7: **end for**
8: **foreach** $v \in V$ **do**
9: $\quad v.val \leftarrow Apply(v.acc)$
10: **end for**

前者の場合は, Data consistency model を選択することができ, Sum 演算は Mutex を使って同時実行制御が行われる. 後者の場合は, 基本的に非同期で処理が行われるが, 競合が起こり得るデータに関しては全てシリアル実行することで同時実行制御は行わない. ディスクアクセスには **Parallel Sliding Windows(PSW)** と呼ばれる, 実行時にランダムディスクアクセスを発生させないアルゴリズムを利用してグラフデータの読み書きを行っている. PSW は, 全てのグラフデータを一度にメモリに載せることが出来ず, disk-based となることを前提としている. グラフ $G=(V,E)$ に対して, 1〜3 からなるプリプロセッシングを行う. 1) V を P 個のインターバルに分割する (後述する shard がメモリに載るように調整する); 2) 各インターバルに含まれる頂点を終点とする全ての辺をそれぞれのインターバルに **shard** として関連付ける. つまり, 全ての辺は必ず唯一の shard に含まれる; 3) 各 shard に含まれる辺を始点でソートする; 実行時には, i〜iv からなる, shard に含まれる辺がソートされていることを利用した高速なシーケンシャルアクセスで処理を行う. i) 注目インターバルに関連付けられた shard を **memory-shard** としてメモリにロードする; ii) 注目インターバルに含まれる頂点を始点とする辺をディスク上の shard からメモリにロードする. 始点でソートされているため, インターバルごとに, 各 shard (Parallel) の対象となる辺集合 (Window) が前から後ろへとスライドしていく (Sliding); iii) 注目インターバルに含まれる各頂点に対し, ユーザ定義の更新処理を並列実行する; iv) i〜iii を残りのインターバルに対して繰り返す; このように実行することで, 1 インターバルにつき P 回のシーケンシャルアクセス, つまり, 全体で P^2 回のシーケンシャルアクセスで処理を完了することができる. P は グラフサイズ/メモリサイズ 程度となるため, 現実的な値と言える.

3. 提案手法

事前実験として, CAS 操作や HTM を含む, 同期プリミティブの特性評価を行った. 類似した先行ベンチマーク調査として, STAMP [15] [16] などが挙げられるが, 具体的なアプリケーションを対象としたものであり, 競合率や計算負荷といった基本的な指標に基づくベンチマークはなされていなかった. そこで, いくつかの指標に基づく同期プリミティブに関するベンチマー

クを作成した．結果として，図 1 から分かるように，ほぼすべての条件において CAS 操作が高速であることが，また，並列度が高い / データ競合率が低い / 同時アクセス要素数がある程度少ない / トランザクションサイズが小さい，といった条件下で HTM の性能が最大限発揮されることが確認された．ただし，CAS 命令を使った操作は，直接的には複数値に対する同時実行制御ができないという問題がある．

本稿の提案手法では，GAS に基づく単一マシン用グラフ処理エンジンの同時実行制御のコストを下げることでグラフ処理の高速化を実現する．GAS は，構成する 4 つの演算のうち，Sum の実行時のみ共有資源のデータの一貫性を考慮することで全体の一貫性を保つことができる計算モデルである．GAS による更新処理のうち，並列セクションにクリティカルセクションが占める時間は多くのアルゴリズムで 20〜30%，意図的に負荷を偏重させても 75% 程度と事前の計測により分かっている．多数のデータからごく一部のデータを非同期に取り出して扱っていることに注意すると，Sum は同時実行制御を必要とするが，データの競合が起こる確率は非常に低い演算ということが分かる．つまり，Lock-based な悲観的同期実行制御を用いると，lock/unlock によるオーバーヘッドが大きい．一方，Lock-free は楽観的同時実行制御であるため，データの競合が起こる確率が低い場合においてはほとんどコストが発生しない．そこで，提案手法では，Sum で既定の単一アキュムレータのみにアクセスする場合は CAS 操作を，それ以外のオブジェクトにもアクセスする場合は CAS では制御出来ないため，HTM を適用する．Algorithm 2 は同時実行制御に CAS 操作を，Algorithm 3 は HTM を適用した GAS による更新処理の擬似コードを示す．前者は 5〜12 行目で CAS 操作を利用することで共有資源である $v_{dest}.acc$ の一貫性を保つ．後者は 5〜7 行を HTM のトランザクションとすることで，他ワーカーからトランザクション内すべての共有資源の一貫性を保つ．どちらも Data consistency model に BSP を利用している．

3.1 Hardware Transactional Memory 適用の最適化

ソーシャルグラフのような現実世界によるグラフデータでは，そのデータが大規模化すると隣接編が極端に多い頂点が発生し得る．このような頂点が含まれると，GAS の更新処理が集中し，データの競合が非常に発生しやすくなる．特に，HTM においては，データの競合が発生した時点でトランザクションが Abort されるため，急激に同時実行制御のコストが増加する．そこで，本節では，隣接辺が多い頂点が含まれる場合でも，**仮想ノード**という実際の 1 つの頂点に対して複数の中間結果を保持する概念を導入することで，共有資源に対するアクセスを分散させてデータの競合発生率を抑える手法を提案する．

GAS 処理のうち，Sum は，注目頂点の値を更新する際に必要となる隣接辺データを集約するための演算である．隣接辺は複数存在する可能性があり，Sum による集約演算は隣接辺の数だけ呼び出される．この際，集約結果が処理対象となる辺の順序

(注2)：CAS の同時アクセス数が 2 以上の場合と No contrl は正確な同時実行制御が行われておらず参考値である

Algorithm 2 GAS with CAS operation

Input: $G_n(V, E)$
Output: $G_{n+1}(V, E)$

```
1: parallel foreach e ∈ E do
2:     v_dest ← get_vertex(e.out)
3:     e.cur_val ← Scatter(e)
4:     fetched ← Gather(e.prev_val)
5:     while true do
6:         loaded ← v_dest.acc.load()
7:         intermediate ← Sum(v_dest.acc, fetched)
8:         if acc_dest.acc.compare_and_swap(
9:                 loaded, intermediate) then
10:            break
11:        end if
12:    end while
13: end for
14: parallel foreach v ∈ V do
15:    v.val ← Apply(v.acc)
16: end for
```

Algorithm 3 GAS with HTM

Input: $G_n(V, E)$
Output: $G_{n+1}(V, E)$

```
1: parallel foreach e ∈ E do
2:     v_dest ← get_vertex(e.out)
3:     e.cur_val ← Scatter(e)
4:     fetched ← Gather(e.prev_val)
5:     transaction start
6:         v_dest.acc ← Sum(v_dest.acc, fetched)
7:     transaction end
8: end for
9: parallel foreach v ∈ V do
10:    v.val ← Apply(v.acc)
11: end for
```

に依存しないようにするため，定義 1 に示すように，Sum を二項演算とする隣接辺データ集合が可換モノイドであることが必要条件である．

定義 1　Sum 演算における可換モノイド

$$\text{隣接辺データ集合} \quad S$$

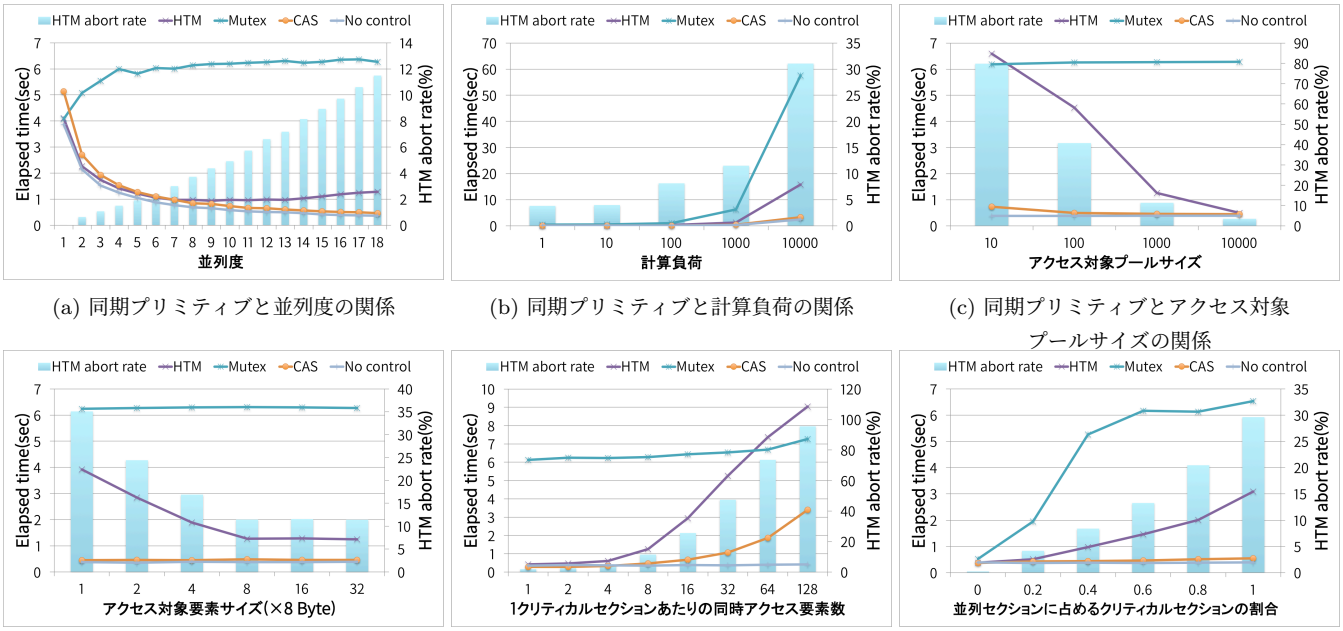
$$\text{二項演算 Sum} \quad \odot : S \times S \to S$$

$$\forall s_1 \forall s_2 \forall s_3 \in S \ ; \ (s_1 \odot s_2) \odot s_3 = s_1 \odot (s_2 \odot s_3)$$

$$\exists z \forall s \in S \ ; \ z \odot s = s \odot z = s$$

$$\forall s_1 \forall s_2 \in S \ ; \ s_1 \odot s_2 = s_2 \odot s_1$$

そこで，組 (S, Sum) が可換モノイドであることを利用して，Sum による集約演算を多段階に分割する．一旦隣接辺データを仮想ノードに集約し，その後，頂点単位で対応する仮想ノードを再度集約することで最終的な値を算出する．このように仮想ノードを挟んで部分集約をすることによって，辺が集中する頂点の演算が発生してもデータの競合発生率が上がらず，トランザクションの Abort 率が抑制されることで HTM の特性を活



(a) 同期プリミティブと並列度の関係

(b) 同期プリミティブと計算負荷の関係

(c) 同期プリミティブとアクセス対象プールサイズの関係

(d) 同期プリミティブとアクセス対象要素サイズの関係

(e) 同期プリミティブと 1 クリティカルセクションあたりの同時アクセス要素数の関係

(f) 同期プリミティブと並列セクション中に占めるクリティカルセクションの割合の関係

図 1　同時実行制御ベンチマーク (注2)

かした処理の高速化することができる.

3.2　GraphChi への実装

2.4 節 で述べた通り, GraphChi はユーザプログラムのために 2 つのインターフェースで提供しているが, 本稿での提案手法は GAS を対象とした手法であるため, GAS に基づくインターフェースに対して実装を行った. なお, 提案手法では, 単一マシンでスレッドプールを利用した並列処理を行うため, ロードバランスを保つことは容易なので Data consistency model には BSP に基づいたモデルを採用した. 隣接辺データを集約する Sum 演算は, メモリから shard をロードする際に, 辺単位で呼び出される. そこで, トランザクションの範囲を狭めるため, 共有資源である演算の中間結果に Sum でアクセスする部分のみに CAS 操作 と HTM を適用した. また, 各頂点に, スレッドごとに利用する中間結果を分けて持たせることで仮想ノードを実現した. これは, 仮想ノードがスレッドローカルになっているため, 同時実行制御を必要としないため高速な実行が可能である. 一方, 仮想ノードの個数が多くなり, 中間結果を保持するためのメモリの使用量とのトレードオフとなる. そのため, 各頂点ごとに, 流入辺数が閾値以上であり, データの競合が発生しやすい頂点のみに仮想ノードを実装し, それ未満の頂点に関してはデータの競合率は低いと見なして仮想ノードを用いずに HTM での同時実行制御を行う実装とした.

4.　実験・評価

提案手法の性能を評価するため, GraphChi における既存の実装である Mutex をベースラインとして検証を行った. グラフデータやアルゴリズムの特徴による差異を確認するため, 表 2 に示すグラフデータで, 複数のグラフ処理アルゴリズムを用いて実験を行った.

本実験で用いたグラフデータを概説する. Pokec [17] [18] は, SNS である Pokec 内でのユーザの繋がりを能わすグラフデータである. Power law exponent(注3) が高いという特徴を持つ. Flickr [19] [20] は Flickr という画像投稿コミュニティ内でのユーザの繋がりを表すグラフデータである. Power law exponent が低いという特徴を持つ. LiveJournal [21] [22] は, ブログサービスである LiveJournal でのユーザの繋がりを表すグラフデータである. Wikipedia, English [23] [24] は 英語版 Wikipedia 記事間のハイパーリンクを表すグラフデータである.

本実験で利用したグラフ処理アルゴリズムを概説する. PageRank は有向グラフの各頂点の重要度を決定するためのアルゴリズムである. 更新時の計算量が少なく, データの更新回数が多いという計算的特徴を持つ. 単一始点最短経路問題 (SSSP) は, 更新時の計算量が少ないという計算的特徴を持つ. 予め開始頂点を指定し, 収束するまで計算を行う. 加えて, 更新時のクリティカルセクション内計算量を意図的に増やした検証用アルゴリズム (heavy) を利用した.

本実験では異なる特徴を持った 2 つの環境で行った. 1 つ目はコンシューマ向けの性能を持つデスクトップパソコン (PC) を利用した. CPU は TSX をサポートしている Intel Core i5-6600, 動作周波数は 3.3GHz で 物理 4 コアである. メモリは 16GB, ストレージとして HDD を利用した. HDD の書き込み速度は約 350 MB/sec, 読み込み速度は 約 210 MB/sec であった. ソフトウェア環境として, OS に Fedora23(kernel-4.2.3), コンパイラに g++5.3.1 を利用した. 2 つ目の環境として, ハイエンドサーバ (SV) を利用した. CPU は TSX をサポートしている Intel Xeon E7-8890v3, 動作周波数は 2.5 GHz で物理 18 コア

(注3)：グラフデータが従う冪乗則の指数. 数値が高いと辺が集中する頂点の割合が下がる.

表 2　データセット

Name	Nodes	Edges	Power law exp.	shard 数 (PC)	shard 数 (SV)
Pokec [17] [18]	1,632,803	30,622,564	3.0810	2	2
Flickr [19] [20]	2,302,925	33,140,017	1.7110	2	2
LiveJournal [21] [22]	4,847,571	68,993,773	2.6510	3	3
Wikipedia, English(dbpedia) [23] [24]	18,268,992	136,537,566	2.3710	6	5

である．メモリは 2TB, ストレージとして HDD を利用した．HDD の書き込み速度は 約 1.3 GB/sec, 読み込み速度は 約 200 MB/sec であった．ソフトウェア環境として, OS に Red Hat Enterprise Linux7(kernel-3.10.0), コンパイラに g++5.3.0 を利用した．どちらの環境も -O3 オプションを用いることでコンパイル時最適化を行っている．並列度はそれぞれの環境における物理コア数とした．また, グラフデータの環境別 shard 数は, GraphChi の機能により, 表 3 のように定められた．

(A) 様々な環境における性能特性, (B) 並列度による性能変化, (C)HTM に仮想ノードを用いた最適化を行った手法についての実験をそれぞれ行った．

(A) 同時実行制御方法の処理全体の時間の比較を 図 2 に示す．また, 既存手法の Mutex によるオーバーヘッドが処理全体の時間に占める割合を 表 3 に, 提案手法の同時実行制御にかかる時間の Mutex との比率を 表 4 に示す．図 2, 表 3, 表 4 や PC 環境の結果より, CAS 操作を利用した場合は全て, HTM でもほとんどの場合において既存の Mutex を使った実装に対して高速化され, 処理全体の時間が最大 19% , 同時実行制御にかかる時間が最大 76% 短縮されたことが分かる．一部の条件において, HTM を利用した場合の方が Mutex を利用した場合に比べて処理全体の時間が増加する結果となった．表 4 と照らし合わせると, 元々の Mutex による同時実行制御コストが処理全体に対して小さい計算の場合は性能の向上があまり望めず, 場合によっては性能が劣化する傾向が見受けられた．また, 横軸が CPU 使用率を表すヒストグラムを 図 3 に示す．図 3(a) と図 3(b) を比較すると, 提案手法が既存手法に比べて最も CPU 使用率が高く理想的な状態を表す階級である Ideal の占める割合が増加したことから, 並列環境を活かすことができたと分かる．PC 環境においても同様の傾向が見られた．

(B) データセットを Wikipedia, アルゴリズムを PageRank に固定した環境で, コア数を制限することで並列度を変化させた時の性能比較を 図 4 に示す．図 4 より, 常に HTM が Mutex に対して優勢であることが分かった．その中での傾向として, 並列度が上がることで HTM の性能は確かに向上しているものの, HTM と Mutex の差異の減少が見受けられる．これは, HTM は並列度が高くなるとデータの競合が発生してしまい, 投機実行に失敗する確率が増加するためと考えられる．図 1(a) の結果を踏まえると, 並列度を更に上げるとやがて実行性能が悪化する可能性があるため, 高並列環境においては 3.1 節 で提案したような最適化の必要性が再認された．

(C) データセットを Wikipedia, アルゴリズムを PageRank に固定した環境で, 既存手法と 実験 (A) での提案手法に加え, いくつかの閾値における仮想ノードを用いた手法の性能比較と仮想ノードの使用メモリ量を 図 5 に示す．横軸ラベルの仮想ノードに続く数値が閾値を示す．図 5 や PC 環境での結果より, HTM にスレッドローカル仮想ノードを利用した最適化を行うことで, 処理全体の時間を更に最大約 8% 短縮することに成功した．閾値を下げる, つまり, スレッドローカル仮想ノードによる実行の割合を増やすと, 同時実行制御のコストを抑えることができ, 実行性能を上げることができるが, 多量のメモリを必要とする．PageRank や SSSP のような Sum 演算の集約対象が単一値であるアルゴリズムでは影響は少ないが, 多値を取るアルゴリズムの場合は必要メモリ量が非常に多くなり, 1 shard あたりのサイズを減少させ, 処理全体のパフォーマンスを下げる可能性がある．一方, 閾値を上げる, つまり, HTM での実行の割合を増やすと, スレッドローカル仮想ノードによる高速化分を初期化処理のオーバーヘッドが上回り実行性能が低下してしまう．図 5 からこのトレードオフが見受けられ, アルゴリズムやデータの特徴に合わせて適切な閾値を設定する必要があることが分かった．

表 3　(A) Mutex にかかる時間が全体に占める割合 (SV)(%)

アルゴリズム	Wikipedia	Flickr	LiveJournal	Pokec
PageRank	14.948	19.839	17.039	19.521
SSSP	11.623	13.761	11.222	12.970
heavy	9.710	12.186	11.153	12.252

表 4　(A) Mutex 比の同時実行制御にかかる時間 (SV)(%)

アルゴリズム	手法	Wikipedia	Flickr	LiveJournal	Pokec
PageRank	HTM	94.008	73.696	46.680	52.050
	CAS	65.779	59.451	58.294	43.142
SSSP	HTM	103.755	57.188	51.183	60.815
	CAS	81.217	39.549	64.143	54.082
heavy	HTM	190.068	84.201	66.535	55.249
	CAS	97.568	60.363	94.153	75.727

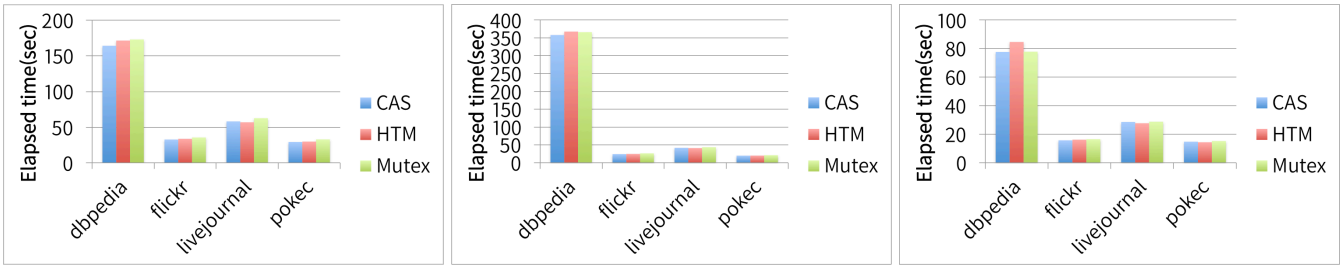
5. 関 連 研 究

5.1 単一マシン用グラフ処理エンジン

GraphChi [2] を参考に設計された単一マシン用グラフ処理エンジンとして, TurboGraph [3] や VENUS [4], GridGraph [5], NXGraph [6] などが挙げられる．

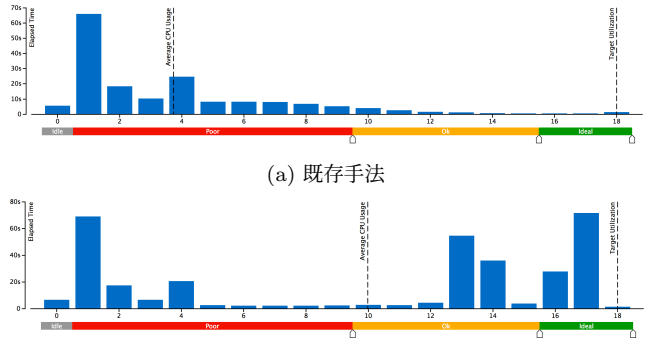
TurboGraph はデータの保存単位を細分化し, Pin-and-Slide という方法でデータのロードを行うことで, IO と CPU のオー

(注4)：Intel VTune Amplifier 2016: https://software.intel.com/en-us/intel-vtune-amplifier-xe



(a) PageRank (b) SSSP (c) heavy

図 2 (A) 同時実行制御方法のデータセットごとの性能比較 (SV)



(a) 既存手法

(b) 提案手法の一例 (HTM)

図 3 (A) CPU 使用率ヒストグラムの一例 (SV) (注4)
灰: Idle, 赤: Poor, 黄: Ok, 緑: Ideal

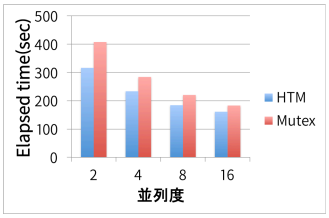


図 4 (B) 同時実行制御方法ごとのコア数と性能の関係 (SV)

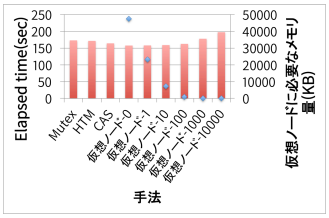


図 5 (C) HTM と仮想ノードを用いた手法の性能とメモリ使用量 (SV)

バーラップ効率を上げている. Pin-and-Slide では, ページと呼ばれる単位でメモリにロード (Pin) し, 次のページのロードをしている間にロード済みのページ内のデータに関する更新処理を行う. ロードが終わり次第更新処理に移り (Slide), 別スレッドで次のページのロードを行う. このように, 処理を細分化することで, ディスク IO 待ち / 更新処理待ち を減らしている. また, ディスクとして SSD を使うことで, ディスク IO のレイテンシを減らしている.

VENUS は, 多くのグラフ処理アルゴリズムにおいて, 隣接頂点から伝搬させる値を計算する際に, 伝搬値を一時的に対象頂点間の辺の値として保持する必要が無いことに着目している. 対象頂点の更新処理に隣接頂点の値を直接利用することで, 中間データをメモリ上に置く必要性やディスクに書き出す必要性をなく, 処理時間の短縮を行っている. しかしながら, 辺に中間データを保持させる必要がアルゴリズム, 例えば 2 ホップ以上先の頂点の値を利用するようなアルゴリズム, を実行することができないという制限がある. 辺データの読み出しと更新処理をオーバーラップさせることで, IO と CPU を効率的に利用している.

GridGraph は, 頂点データを 1 次元に, 辺データを始点と終点に基づく 2 次元に分割して管理することで, 従来の vertex-centric や edge-centric な手法と比べ, 1 イテレーション当たりの辺データのロード回数を 1 回限りに抑えている. また, グラフ処理のスケジューリングをユーザ定義のアルゴリズムに合わせて選択できるようにすることで, 不必要な辺データのロードを防ぎ, 高速化を行っている.

NXGraph は, 辺の始点と終点を考慮する Destination-Sorted Sub-Shard という手法を用いることで, 限られたメモリ空間でグラフの局所性を利用し, 更新処理時に発生する書き込みによるデータの競合発生率を抑えている. また, グラフデータのサイズと利用可能なメモリサイズに基いて更新処理に Single-Phase Updating と Double-Phase Updating, Mixed-Phase Updating の 3 種類を使い分けることでディスクアクセスを減らしている.

5.2 Hardware Transactional Memory の応用

HTM は古くから提案されてきたが [8], 近年ようやく商用のプロセッサにも実装され, 普及し始めたばかりの若い技術である. しかしながら, 同時実行制御は分散処理やデータベース等, 広い分野で重要な技術であるため, その有用性は注目を集めており, 本稿で提案した単一マシングラフ処理エンジンだけでなく, 既に分散処理におけるメッセージパッシング [?] やインメモリデータベース [25] などにも応用されている. これらの既に応用化が進んでいる分野の特徴として, 大部分は並列的に行われるものの, 同時実行制御が真に必要となるような複数アクセスが発生する割合が低い処理を扱っているということが挙げられる. データの競合が少ない限り, 同時実行制御としてのオーバーヘッドが非常に少ない, という HTM の特性を活かすことが重要であると分かる.

5.3 MapReduce との類似点

本稿で提案した仮想ノードへの部分集約に類似する概念として, 分散処理モデルである MapReduce [26] における combiner による部分集約が挙げられる. MapReduce は基本的なフローは, 値を mapper で処理してからネットワーク越しで reducer に渡し, 集約するというものである. そこに combiner を導入し, reducer での集約の前に mapper のノード上で部分的に集約をすることで通信量を削減して高速化を図ることができる [27].

提案手法では同時実行制御を減らすため, MapReduce では通信量を減らすため, と, 部分集約を利用する目的は異なるが, どちらも分割統治法の集約ストラテジを最適化することで, ボトルネックになり得る特定の処理を減らしている.

6. まとめ

本稿では, 単一マシンメニーコア環境でのグラフ処理エンジンにおいて, 同時実行制御方法を従来の Mutex から CAS 操作や HTM に置き換えることで高速化を行った. 更に, GAS の特性を考慮し, HTM とスレッドローカル仮想ノードを組み合わせる高速な処理手法を提案した. また, 提案手法を GraphChi に実装し, 性能評価を行った. 同時実行制御に Mutex を用いた実装と比較して, CAS 操作に置き換えることで処理全体にかかる時間を最大約 19%, 同時実行制御にかかる時間を最大約 76%, HTM を用いた手法に置き換えることで, 処理全体にかかる時間を最大約 10% , 同時実行制御にかかる時間を最大約 53% 短縮することに成功した. また, HTM とスレッドローカル仮想ノードを用いた最適化を行うことにより, 処理全体にかかる時間を更に最大約 8% 短縮することに成功した.

謝辞

本研究の実験環境を提供して頂いたヒューレット・パッカード社に深謝申し上げます.

文献

[1] Facebook. Facebook company info. `http://newsroom.fb.com/company-info/`.

[2] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a pc. In *Proceedings of OSDI*, Vol. 12, pp. 31–46, October 2012.

[3] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of SIGKDD*, pp. 77–85. ACM, August 2013.

[4] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John CS Lui, and Cheng He. VENUS: Vertex-centric streamlined graph computation on a single pc. *Proceedings of ICDE*, April 2015.

[5] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX ATC*, pp. 375–386, July 2015.

[6] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. NXgraph: An efficient graph processing system on a single machine. *Proceedings of ICDE*, May 2016.

[7] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Proceedings of WAPA*, Vol. 11, pp. 985–1042, September 2010.

[8] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, Vol. 21. ACM, 1993.

[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of OSDI*, pp. 17–30, Hollywood, CA, October 2012. USENIX.

[10] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. GraphLab: A new framework for parallel machine learning. *Proceedings of UAI*, July 2014.

[11] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation intel core processor. *Proceedings of IEEE*, No. 2, pp. 6–20, December 2014.

[12] Ruud Haring, Martin Ohmacht, Thomas W Fox, Michael K Gschwind, David L Satterfield, Krishnan Sugavanam, Paul W Coteus, Philip Heidelberger, Matthias A Blumrich, Robert W Wisniewski, et al. The IBM Blue Gene/Q compute chip. *Proceedings of IEEE*, Vol. 32, No. 2, pp. 48–60, December 2012.

[13] B Sinharoy, JA Van Norstrand, RJ Eickemeyer, HQ Le, J Leenstra, DQ Nguyen, B Konigsburg, K Ward, MD Brown, JE Moreira, et al. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, Vol. 59, No. 1, pp. 2–1, January 2015.

[14] C Kevin Shum, Fadi Busaba, and Christian Jacobi. IBM zEC12: The third-generation high-frequency mainframe microprocessor. *Proceedings of IEEE*, No. 2, pp. 38–47, June 2013.

[15] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of IISWC*, pp. 35–46. IEEE, September 2008.

[16] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of ISCA*, pp. 144–157. ACM, June 2015.

[17] Pokec network dataset – KONECT, May 2015.

[18] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present DTI*, May 2012.

[19] Flickr network dataset – KONECT, May 2015.

[20] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the Flickr social network. In *Proceedings of COSN*, pp. 25–30, March 2008.

[21] LiveJournal network dataset – KONECT, May 2015.

[22] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of WWW*, pp. 695–704, April 2008.

[23] Wikipedia, English network dataset – KONECT, May 2015.

[24] Sren Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Proceedings of ISWC*, pp. 722–735, October 2008.

[25] Viktor Leis, Alfons Kemper, and Tobias Neumann. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of ICDE*, pp. 580–591. IEEE, March 2014.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Proceedings of ACM*, Vol. 51, No. 1, pp. 107–113, March 2008.

[27] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, Vol. 3, No. 1, pp. 1–177, 2010.