SIMD 命令を用いたグラフクラスタリングの高速化

塩川　浩昭† 　山室　健† 　藤原　靖宏† 　鬼塚　真†

† NTT ソフトウェアイノベーションセンタ　〒 180–8585 東京都武蔵野市緑町 3–9–11
E-mail: †{shiokawa.hiroaki,yamamuro.takeshi,fujiwara.yasuhiro,onizuka.makoto}@lab.ntt.co.jp

あらまし　大規模グラフデータに対して CPU 特性を考慮した高速なクラスタリング手法を提案する．グラフデータに対するクラスタリング手法として Newman 法や CNM 法，BGLL 法などが提案されてきたが，これらの手法を大規模なグラフデータに適用した場合，データのランダムアクセスと実行命令数の増大に伴い参照効率と実行効率の悪化が課題になることが判明した．そこで提案手法では，モジュラリティに基づくグラフクラスタリングに対して，CPU のキャッシュヒット率向上を考慮したデータ構造の適用，さらに近年注目されている CPU 内の SIMD 命令を活用することで従来手法の課題を解決する．

キーワード　グラフ，クラスタリング，並列化，SIMD

1. はじめに

グラフ構造はデータをノードとエッジで表現した基本的なデータ構造であり，情報推薦や情報検索，科学データ分析などの様々な分野で利用されている．特に近年では，数億ノードから構成される大規模なグラフ構造が登場し，このようなデータに対する高速な解析処理技術への需要が高まっている．例えば，Facebook では 2012 年に 1ヶ月当たりのアクティブユーザ数が 10 億人，また Twitter では一日当たりの投稿数が 3 億 4000 万を突破したと報告されている．このように，大規模なグラフ構造は現実に存在し，今後もその規模をさらに増大させていくことが考えられ，これらのグラフ構造に対する高速な解析手法は将来的に必要不可欠な技術となってくると言える．

グラフ構造解析のひとつとして，クラスタリング手法が挙げられる．グラフ構造には，クラスタと呼ばれる相互に密な接続を有する部分ノード集合が存在する．例えば，Web グラフではトピックや関心の近いページ集合が互いにリンクすることで，トピックや関心の類似したページ群がコミュニティを形成する傾向にある．このようにグラフ構造中のクラスタは互いに共通した性質を持つことから，グラフ構造の理解や様々なアプリケーションに利用され，非常に重要な要素技術となっている．このような背景からこれまで様々なクラスタリング手法の研究が行われてきた．

Modularity [1] を用いた手法は大規模なグラフ構造を高速にクラスタリングする手法として注目を集めている手法の一つである．Modularity は処理対象のグラフ構造をランダムグラフとみなしてモデル化し構築した指標であり，グラフ構造がランダムグラフから離れるほど良いスコアを示す．すなわち Modularity は，よりクラスタ内のエッジが密であり，かつ，クラスタ間のエッジが疎であるクラスタリング結果であるほど良い Modularity の値を示し，より適切にクラスタを抽出できていることを表す特徴を持つ．ゆえに，Modularity を用いた近年の研究 [1, 2, 3, 4, 5] では，高い Modularity の値を高速に求める課題となる．しかし，Modularity 値の最大化は NP 困難であることから，Modularity を用いたクラスタリング手法では高速にかつ可能な限り高い Modularity の値をいかにして求めるかという点が重要な研究課題となっている．

その中でも，Blondel らによる BGLL [5] は，数億ノード規模のグラフ構造に対して，高速かつ高い Modularity の値を示す手法として知られている．BGLL では，2 つのフェーズからなるパスを繰り返すことによりクラスタリングを行う．第 1 フェーズでは，グラフ構造を構成する各ノードがそれぞれ別のクラスタである状態から開始する．任意の順にノードを選択し，選択したノードに対して隣接するノードの中から最も Modularity を向上させる隣接ノードを 1 つ選択する．同一のクラスタとみなす操作を，Modularity の値が向上する限り繰り返す．これに続く第 2 フェーズでは，同一のクラスタと判定された全てのノードとエッジを 1 ノードへ集約する操作を行う．Modularity の値が向上する限り，これら 2 つのフェーズからなるパスを反復しクラスタの抽出を行う．従来手法 [2, 3, 4] ではクラスタ同士の統合を行う際に，クラスタに含まれる全てのノードとエッジに対し，Modularity の向上量を計算する必要があった．これに対し，BGLL では収束した処理結果を 1 ノードへ集約することで，処理に必要となるノードとエッジの参照数を削減させることに成功している．また BGLL は，任意の順に選択されたノードが，それぞれが隣接するノードに対してのみ統合の可能性を検証する．ゆえに，抽出されるクラスタサイズの偏りを防ぐことができ，その結果高い Modularity の値を示すことがわかっている．BGLL は 1 億ノード規模のグラフデータに対して約 2.5 時間程度でクラスタリング処理可能であることが Blondel らによる研究で報告されている [5].

本稿では，より大規模な数億〜数十億ノード規模のグラフ構造に対する高速なクラスタリングの実現に向け，処理の並列化による BGLL の高速化手法を提案する．本手法では，BGLL に

含まれる Modularity 計算部分に対し，複数の比較処理を CPU の 1 サイクルで同時実行することが可能な SIMD 命令を活用することで処理の高速化を行う．さらに，クラスタリング処理時に生じるメモリ上のデータアクセスの効率化を実現する，キャッシュヒット率向上に向けたデータ構造を導入し，クラスタリング処理の更なる高速化を図る．SIMD 命令により Modularity 計算を並列実行する際に，Modularity 計算の定義式をより単純な命令の組み合わせに置き換え，SIMD 命令の実行数の増加を抑制する．本稿では，提案手法のプロトタイプを実装し実データを用いた評価実験を行う．評価実験により，BGLL に対して Modularity の値を同程度に示しつつ，高速に処理可能であることを示す．本稿の構成は以下の通りである．2 節で関連研究について述べ，3 節で本研究の前提となる概念と手法について説明する．4 節でキャッシュヒットを考慮したデータ構造について述べ，5 節で提案手法の詳細を説明する．6 節で提案手法の評価を行う．最後に 7 節で本稿のまとめと今後の課題について述べる．

2. 関 連 研 究

2.1 Modularity に基づくクラスタリング

Modularity を用いたクラスタリング手法について様々な研究が行われている．代表的な手法として Girvan–Newman 法 [1]，Newman 法 [2]，CNM 法 [3]，WT 法 [4]，BGLL [5] などが挙げられるが，いずれの手法においても本研究で対象とする大規模なグラフ構造を高速に処理することは難しい．

Girvan らはトップダウンにエッジを間引きグラフを分割する Girvan–Newman 法 [1] を提案した．グラフ構造全体を 1 つのクラスタとみなし，クラスタ間を横断する可能性の高いエッジから順に切り離すことにより小さなクラスタへと分割していく．エッジを切り離す度に各エッジに付与されたスコアを再計算することから，エッジが疎なグラフ構造に対してノード数を n とすると $O(n^3)$ の計算量を必要とする．計算量が膨大であるため，1 万ノード規模の処理には適用できないと報告されている．

Newman は，貪欲法によりボトムアップ式にクラスタを抽出する Newman 法 [2] を提案した．この手法では，各ノードがそれぞれ別のクラスタである状態から処理を開始し，Modularity が最も向上するノード対を貪欲法により同一クラスタへと統合していく．統合させるべきノード対を全探索することから，エッジが疎なグラフ構造に対してノード数を n とすると $O(n^2)$ の計算量を必要とする．これに対し，Clauset らは Newman 法に対してヒープ構造を導入することで高速化する CNM 法 [3] を提案し，計算量 $O(n \log^2 n)$ を達成した．また，脇田らはクラスタ統合時のクラスタサイズの不均衡による処理速度の低下を指摘し，新たに CNM 法のさらなる高速化を行う WT 法 [4] を提案している．WT 法ではクラスタ統合時に Modularity の向上量をクラスタサイズで正規化するヒューリスティクスを用いることでクラスタサイズの不均衡を解消する．これらの手法では最大で数百万ノード規模までの処理可能と報告されている．

Blondel らは，Newman 法の高速化手法として BGLL [5] を提案した．BGLL は Modularity の局所最適化とノードの一括集約により，クラスタリング処理におけるノードとエッジの参照数の削減に成功している．BGLL の最悪計算量は知られていないが，文献 [5] において従来手法に対する高速性が実験的に示されており，我々の知る限り最も高速にクラスタリング処理が可能な手法である．文献 [5] では，最大で数億ノード規模までの処理が可能であると報告されている．さらに，BGLL は任意の順でノードを選択することで抽出されるクラスタサイズに偏りが生じることを防ぎ，これまで述べた手法の中で最も良い Modularity の値を示すことも文献 [5] にて示されている．

本稿はこれらの手法に対し，より大規模なグラフ構造の高速なクラスタリング手法の実現に向け，SIMD 命令とキャッシュヒット率向上を考慮した BGLL のさらなる高速化に取り組む．

2.2 SIMD 命令を用いた高速化

2002 年頃以降，CPU や GPU の SIMD 命令を用いた各種手法の高速化に関する研究は盛んに行われている．グラフクラスタリングの高速化に関する従来研究 [10, 11] は存在するが，これらの研究ではラベル伝搬やランダムウォークに着眼するものである．計算量の大きなこれらの手法に対して，局所的なデータ並列化を行うことで処理性能の向上とスケーラビリティの確保を図っている．しかしながら，これらの手法の計算量は依然として大きく，本研究で対象とするような数億〜数十億ノード規模のグラフを対象としたクラスタリングは難しい．これに対し，Modularity による手法に着目し数億〜数十億ノードからなる大規模なデータに対するクラスタリングを行う処理に SIMD 命令を適用したものは本研究がはじめてである．

3. 事 前 準 備

3.1 クラスタリング指標 Modularity

本稿で提案するクラスタリング手法では，抽出結果の良さを示す指標 Modularity を対象とする．そこで，クラスタリング指標 Modularity について概説する．

Modularity はクラスタ内に含まれたノード間のエッジが密であり，クラスタ間に存在するエッジが疎となる程良い値を示す指標である．クラスタリング手法により取得したクラスタ集合を C，クラスタ i からクラスタ j へ接続されているエッジ数を e_{ij}，グラフ構造全体に含まれる総エッジ数を m とするとき，Modularity Q は定義 1 のように定義される．Modularity が負の値を取る場合は $Q = 0$ とし，常に Modularity Q は $0 \leq Q \leq 1$ の値を示す．

［**定義** 1］ Modularity Q

$$Q = \sum_{i \in C} \left\{ \frac{e_{ii}}{2m} - \left(\frac{\sum_{j \in C} e_{ij}}{2m} \right)^2 \right\}$$

3.2 既存手法：BGLL

本稿では BGLL に対して，キャッシュヒットを考慮したデータ構造の導入と SIMD 命令を用いた並列化の導入により，処理のさらなる高速化を行う．そこで本節では，既存手法 BGLL について概説する．

BGLL は 2 つのフェーズから構成されるパスを繰り返すこ

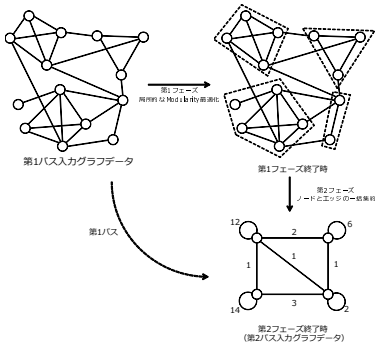

図 1　BGLL の処理の流れ

とでクラスタを抽出する手法である．第 1 フェーズにて，各ノードと隣接するノード間のみにおいて Modularity の値を準最適化し，第 2 フェーズにて，第 1 フェーズで得られた結果を 1 ノードに一括集約する．BGLL は Modularity の値が増加する限り，2 つのフェーズからなるパスを反復する．各フェーズの詳細について以下に説明する．

3. 2. 1　第 1 フェーズ：Modularity の局所最適化

第 1 フェーズではまず，入力されたグラフ構造に対して，任意の順（ランダム順）にノードを選択する．その後，選択されたノードの全隣接ノードに対して，クラスタを統合した際に上昇する Modularity の値 $\triangle Q$ を計算する．一般的に定義 1 で示した Modularity Q を求めるためには，全てのノードとエッジを参照する必要があるため効率的ではない．しかし BGLL では Modularity の変化量にのみ着目し，定義 1 から Modularity 変化量 $\triangle Q$ を導出し，計算の効率化を図っている．2 つのクラスタ i と j を統合した際の Modularity 変化量 $\triangle Q$ を定義 5.1 に示す．

［**定義** 2］ Modularity 変化量 $\triangle Q$

$$\triangle Q = 2\left\{\frac{e_{ij}}{2m} - \left(\frac{\sum_{k\in C} e_{ik}}{2m}\right)\left(\frac{\sum_{k\in C} e_{jk}}{2m}\right)\right\}$$

第 1 フェーズでは，上記の定義に従い $\triangle Q$ を計算し，最も Modularity 変化量 $\triangle Q$ が大きくなるノード対を選出する．その後，選出したノード対を同一のクラスタとしてラベル付する．この処理は Modularity の値が向上しなくなるまで継続する．

3. 2. 2　第 2 フェーズ：ノードとエッジの一括集約

第 2 フェーズでは，第 1 フェーズで得られた処理結果に対してノードとエッジを一括集約し，処理で扱うグラフ構造のサイズを縮小する．図 1 に示したように，同一のクラスタに含まれるノードは，1 つのノードにまとめられる．また，クラスタ内に存在するエッジについては，自己ループのエッジ 1 本へ集約し元のエッジ本数の 2 倍の重みをつける，クラスタ間に存在するエッジについては，同一のクラスタ対につきエッジ 1 本に集約し，エッジ本数分の重みをつける．第 2 フェーズで生成された集約された重み付きグラフ構造は，次回以降のパスで入力データとして与えられる．

3. 2. 3　BGLL の課題

BGLL は，第 1 パスの第 1 フェーズにおいて入力された全てのノードをランダム順に参照し，Modularity 変化量 $\triangle Q$ が増加する可能性を検証することで，クラスタの統合を行う．この処理は，Modularity 変化量 $\triangle Q$ が増加しなくなるまで継続されるため，入力されるデータのノード数増加に伴い，第 1 パスにおける処理時間は増加することとなる．これにより，より大規模なグラフ構造であるほど，第 1 パスにおける処理時間がボトルネックとなり，クラスタリング処理時間の短縮が困難となる．

表 1　BGLL における処理時間と Modularity 向上量の評価

パス	1	2	3	4	5	6	合計
Time(sec)	418.7	3.34	0.17	0.03	0.01	0.01	422.3
Modularity	0.669	0.06	0.01	0.01	0.01	0	0.759

具体例を表 1 に示す．表 1 では，ノード数 5,363,260，エッジ数 79,023,142 から構成されるグラフデータに対して Blondel らが公開している Louvain 法のプログラム(注1)を適用した際の処理時間と Modularity 値の変化の内訳を示している．表 1 のパスは BGLL を実行した際の反復回数，Time は各パスに消費した処理時間，Modularity は各パスで増加した Modularity の値を表している．表 1 からも分かる通り，BGLL では，第 1 パスにその処理時間の 99%以上を消費している．また Modularity については，第 2 パス以降では大きな向上は見られないことがわかる．この傾向は，予備実験で使用したデータセットのみならず，他のデータセットにおいても同様に見られるものである．このことから，我々は BGLL の第一パスを効率化することで，クラスタリング処理全体を大幅に高速化できると考えた．

4.　キャッシュヒットを考慮したデータ構造設計

本稿では，グラフ構造 $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ を隣接リスト表現で扱う．隣接リスト表現では，各ノード毎にノード u とそのノードに隣接するノード集合 $\Gamma(u)$ のみをリスト上にして表現したデータ構造であるため，隣接行列表現に対してより空間効率よくデータを保持することができる．しかしながら，Modularity によるグラフクラスタリングにおいてはメモリ上のデータに対する参照が頻繁に発生するため，各ノード毎に離散したメモリ番地へのアクセスが生じる可能性のある隣接リスト表現はキャッシュヒット率の低下を招き，参照効率が悪化する可能性がある．

そこで，本稿では隣接リスト表現をメモリ上において連続した領域に配置することで，データ参照時のキャッシュヒット率を向上させる．本データ構造の概要を図 2 に示す．

本データ構造は 2 つの連続した配列から構成される．一つはノード集合を格納した配列 V_a，もう一つは隣接ノード集合を格納した配列 E_a である．まず，配列 V_a では，配列の番地をノードの識別番号とし，その要素として配列の先頭から現在の番地までに含まれるノードの次数の合計値を格納する．ここで格納した次数は，配列 E_a のインデックスとして利用される．次に配列 E_a において，配列 V_a に格納したノードの順にそれぞれのノードの隣接ノードを格納する．

(注1)：https://sites.google.com/site/findcommunities/

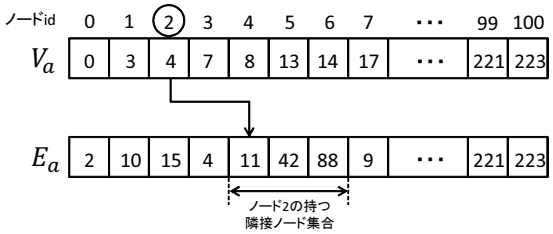

図 2　キャッシュヒットを考慮したデータ構造

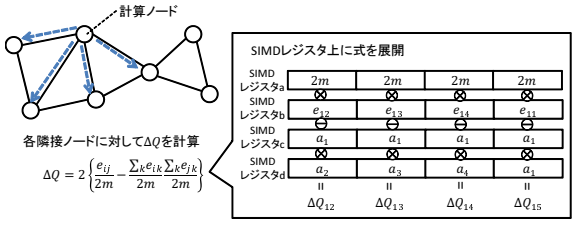

図 3　SIMD 命令による並列化の概要

このようにノードとエッジを格納することにより，全てのデータを連続領域に格納しつつ，それぞれのデータに対して効率良くアクセスすることが可能になる．例えば，あるノード u の隣接ノード集合 $\Gamma(u)$ を参照する場合を考える．この場合，まず，配列 V_a の番地 u にアクセスし，次数の合計値 $V_a(u)$ および $V_a(u+1)$ を取得する．この合計値 $V_a(u)$ および $V_a(u+1)$ は，配列 E_a に存在するノード u およびノード $u+1$ の隣接ノード集合を格納した領域の先頭番地を指すため，配列 $E_a(V_a(u))$ から $E_a(V_a(u+1))-1$ にアクセスすればノード u の隣接ノード集合 $\Gamma(u)$ が取得可能である．

例えば図 2 では，ノード 2 に隣接するノード集合を取得する場合を示している．ノード 2 の隣接ノード集合を取得する場合，まず配列 V_a から配列 E_a の参照先 $V_a(2)=4$ を取得する．次に，ノード 2 の持つ次数を取得する．この場合，$V_a(3)-V_a(2)=7-4=3$ となり，ノード 2 の持つ次数は 3 である．最後に配列 E_a 中の $E_a(3)$ から $E_a(3+3)E_a(6)$ までアクセスし，ノード 2 の隣接ノード集合 $\{11,42,88\}$ を取得する．

このデータ構造では，データをメモリ上の連続した領域の載せた状態で処理を行うことが可能となるため，キャッシュヒット率が向上しやすく，メモリ上のデータ参照が頻繁に発生するグラフクラスタリングにおいてデータアクセスを効率化できると考えられる．

5. SIMD 命令による Modularity 計算

本稿では，複数のデータに対する演算を CPU の 1 サイクルで同時実行することが可能な SIMD 命令を活用することで，Modularity によるクラスタリングをデータ並列化し高速化する．3.2 節で示したように，既存手法 BGLL は各パスの第一フェーズにおいて，全てのノードを走査する．そして，それぞれのノードの隣接ノード集合に対して Modularity 変化量 $\triangle Q$ を計算し，$\triangle Q$ を最大化するノードとクラスタの併合を繰り返すことで最終的なクラスタを抽出する．しかしながら，BGLL ではこの処理を全てのエッジに対して複数回イテレーションする必要があり，クラスタリングの処理時間に対して支配的となり，その性能を劣化させる原因となっている．

そこで，本稿では BGLL の第一フェーズにおける，各ノードの Modularity 変化量 $\triangle Q$ の計算に対して SIMD 命令を用いたデータ並列化を導入し，その処理時間を短縮する．提案手法における SIMD 命令を用いたデータ並列化の概要を図 3 に示す．

5.1 Modularity 変化量の並列計算

まず，任意のノードが与えられた際にその全ての隣接ノードに対して SIMD 命令を用いて並列に Modularity 変化量を計算する．提案手法では，BGLL と同様に任意の順にノード u を選択する．次に，選択されたノード u の隣接ノード集合 $\Gamma(u)$ を 4. 節に示した手順で取得する．そして，取得した隣接ノード集合 $\Gamma(u)$ に対して，SIMD 命令を用いて並列に Modularity 変化量の計算を実行する．

この際に，従来手法では定義で示した，Modularity 変化量 $\triangle Q$ を計算する必要がある．このとき，各計算に必要な e_{ij} や $\sum_{k\in C} e_{ik}$，$\sum_{k\in C} e_{jk}$ が事前に求められていた場合，SIMD 命令を用いた $\triangle Q$ の演算には下記の 6 つの演算が必要となる．

- e_{ij} と $2m$ の商
- $\sum_{k\in C} e_{ik}$ と $2m$ の商
- $\sum_{k\in C} e_{jk}$ と $2m$ の商
- $\sum_{k\in C} e_{ik}/2m$ と $\sum_{k\in C} e_{ik}/2m$ の積
- $e_{ij}/2m$ と $(\sum_{k\in C} e_{ik}/2m)(\sum_{k\in C} e_{ik}/2m)$ の差
- 2 と $e_{ij}/2m-(\sum_{k\in C} e_{ik}/2m)(\sum_{k\in C} e_{ik}/2m)$ の積

しかしながら，この計算で本来求める必要がある事項は，$\Gamma(u)$ に含まれるノードのうち Modularity 変化量 $\triangle Q$ を最大化し得るノードの発見である．すなわち，$\triangle Q$ に示した計算を全て実行する必要はなく，$\Gamma(u)$ に含まれる全てのノードが $\triangle Q$ に関する相対的な順位が求まれば良いと考えられる．そこで本稿では Modularity 変化量 $\triangle Q_{ij}$ をより容易な計算の組合せに変換した新たな Modularity 変化量 $\triangle Q'_{ij}$ を以下に定義する．

[**定義** 3]（簡略 Modularity 変化量 $\triangle Q'_{ij}$）

$$\triangle Q'_{ij} = 2me_{ij} - \sum_{k\in C} e_{ik} \sum_{k\in C} e_{jk}.$$

定義 3 で定義した Modularity 変化量 $\triangle Q'_{ij}$ から下記の補題が導出される．

[**補題** 1]（$\triangle Q'_{ij}$ と $\triangle Q_{ij}$ の大小関係） 簡略 Modularity 変化量 $\triangle Q'_{ij}$ は，Modularity 変化量 $\triangle Q_{ij}$ によって得られる値の大小関係と等価な大小関係を示す．

証明　定義 5.1 より，

$$\triangle Q_{ij} = 2\left\{\frac{e_{ij}}{2m} - \left(\frac{\sum_{k\in C} e_{ik}}{2m}\right)\left(\frac{\sum_{k\in C} e_{jk}}{2m}\right)\right\}$$

$$\frac{4m^2}{2}\triangle Q_{ij} = 2me_{ij} - \sum_{k\in C} e_{ik} \sum_{k\in C} e_{jk}.$$

したがって，

$$\triangle Q'_{ij} = 2me_{ij} - \sum_{k\in C} e_{ik} \sum_{k\in C} e_{jk},$$

となり，$\triangle Q'_{ij}$ は，$\triangle Q_{ij}$ と相対的に等価となる．　□

$\triangle Q'_{ij}$ を用いることで，Modularity 変化量 $\triangle Q$ では SIMD 命令上 6 回の演算が必要であったものを，下記の通り 3 回の演算で相対的に等価に求めることが可能になる．

- e_{ij} と $2m$ の積
- $\sum_{k\in C} e_{ik}$ と $\sum_{k\in C} e_{jk}$ の積
- $2me_{ij}$ と $\sum_{k\in C} e_{ik} \sum_{k\in C} e_{jk}$ の差

提案手法では，SIMD レジスタ上において $\triangle Q'_{ij}$ を用いて並列に計算を行うことにより，従来の BGLL よりも少ない CPU サイクルでクラスタを求めることができる．

5.2 Modularity 変化量最大化ノードの選択

前述した Modularity 変化量からその値を最大化する隣接ノードを SIMD 命令を用いて取得する．本手法では Modularity 変化量を最大化するノードの取得を行うために，4 つの SIMD レジスタ，*reg_id*，*reg_result*，*reg_max_id*，*reg_max*，*reg_mask* を使用する．*reg_id* は 5.1 節に示した手順によって計算対象となっている隣接ノード集合の ID を格納した SIMD レジスタ，*reg_result* は *reg_id* に対応した Modularity 変化量の計算結果が格納された SIMD レジスタ，*reg_max_id* は Modularity 変化量を最大化する可能性のある隣接ノード ID が格納された SIMD レジスタ，*reg_max* はノード u と *reg_max_id* に存在する隣接ノードによって得られる Modularity 変化量を格納した SIMD レジスタ，そして *reg_mask* は最大化ノードを取得するために利用するマスク用の SIMD レジスタである．

まず，5.1 節に示した手順によって，ノード u の隣接ノード集合 $\Gamma(u)$ のうち，1 つの SIMD レジスタにデータ入力可能な分の隣接ノード部分集合 *reg_id* に対する Modularity 変化量計算が終了した場合を考える．この時，*reg_result* には各隣接ノードに対する Modularity 変化量の値が入力されている．次に図 4 のように，*reg_result* と *reg_max* の 2 つの SIMD レジスタ間の大小比較を行い，*reg_result* の方が大きな値を持っている SIMD レジスタ上のアライメントのビットを全て 1，そうでないアライメントのビットを全て 0 とし，その結果を *reg_mask* で保持する．

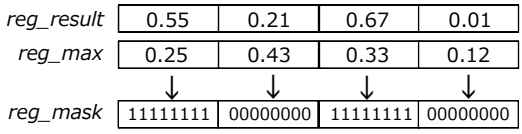

図 4　*reg_result* と *reg_max* から *reg_mask* の作成

reg_mask 取得後，このマスクの値と *reg_id* および *reg_max_id* を用いて，Modularity 変化量が大きくなったノード ID を下記のように抽出する．

$$reg\_max\_id = (!reg\_mask \,\&\, reg\_max\_id) \mid (reg\_mask \,\&\, reg\_id)$$

ノード u に対して全ての $\Gamma(u)$ に対して処理が終了するまで，上記の手続きを繰り返し，*reg_max_id* および，*reg_max* に Modularity 変化量を最大化する可能性のある隣接ノードの ID 及び Modularity 変化量を更新し保持し続ける．Modularity 変化量の計算が終了後，*reg_max_id* から *reg_max* が最大値となるようなアライメントを操作により取得することで，ノード u に対して Modularity 変化量を最大化する隣接ノードを取得する．

6. 評価実験

提案手法の有効性を評価するために，Blondel らによる BGLL に対して，本提案手法を適用し比較実験を行う．本実験で下記の実データを利用し評価を行った．

- Live [12] (注2)：LiveJournal と呼ばれる 1999 年に開始された SNS のユーザの友達関係からなるネットワーク．ノード数は 5,363,260 であり，エッジ数は 79,023,142 である．
- uk-2005 (注3)：2005 年にクロールした uk ドメインの Web グラフのスナップショット．ノード数は 39,459,925 であり，エッジ数は 936,364,282 である．
- webbase-2001 [12] (注4)：WebBase クローラ(注5)によって 2001 年にクロールされた Web グラフのスナップショット．ノード数は 118,142,155 であり，エッジ数は 1,019,903,190 である．

これらのデータセットの次数分布を図 5 に示す．

また，本実験では，DynamIc Graph Generator (DIGG) (注6)を用いて，人工のグラフデータを生成した．DIGG では，生成するグラフのノード数とグラフ中に含まれるエッジの偏りを表すパラメータ β の値を指定することで，グラフデータを生成することが可能である．β の値は大きな値を持つほど，ノード間の次数に偏りが生じ，小さな値を持つほどノード間で均一な次数をもつグラフデータが生成される．本実験では，DIGG により生成したノード数を 10,000 ノードから 10,000,000 ノードまで 10 倍ずつ変化させた 4 つのグラフデータを利用し評価を行った．また，DIGG で生成した全てのグラフデータは $\beta = 2.0$ と設定されている．本実験では，これらのデータセットに対して，クラスタリング処理が終了するまで処理を行った際の処理時間を示し，比較を行う．本実験には CPU が Intel Xeon Quad–Core L5640，メモリが 144GB の Linux サーバを利用した．また，本実験で使用した SIMD レジスタのサイズは 128bit であり，計算に利用するノード ID やエッジ数は 32bit のデータ型として扱った．

6.1 実データによる評価

本実験では，前節で述べた実データに対して提案手法と BGLL の処理速度，処理制度を比較する．

図 6 に処理速度の比較結果を示す．図 6 の Original，Cache，Cache+SIMD はそれぞれ，BGLL，キャッシュを考慮したデータ構造のみを用いた BGLL，キャッシュを考慮したデータ構造と SIMD による並列化を行った BGLL を示している．図 6 からわかるように，Cache および，Cache+SIMD は全てのデータセットにおいて，Original よりも高速に処理できていることがわかる．Cache は Original に対して約 5〜8 倍程度高速

(注2)：http://law.di.unimi.it/webdata/ljournal-2008/
(注3)：http://law.di.unimi.it/webdata/uk-2005/
(注4)：http://law.di.unimi.it/webdata/webbase-2001/
(注5)：http://diglib.stanford.edu:8091/~testbed/doc2/WebBase/
(注6)：http://digg.cs.tufts.edu/

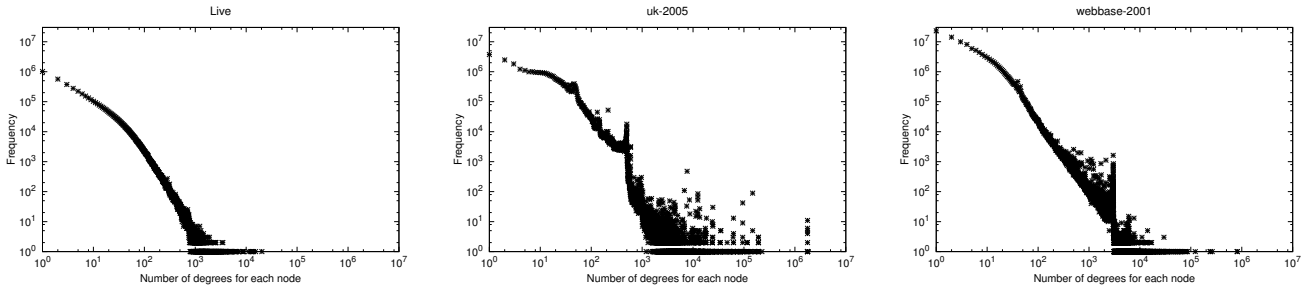

図 5　各データセットの次数分布 (両対数グラフ)

化に成功している．また，Cache+SIMD は，いずれのデータセットに対しても Cache に対して約 4 倍の高速化に成功している．本実験で用いた実験環境では SIMD レジスタのサイズ 128bit に対して 32bit のデータ型を利用した実装を行ったため，Modularity 計算部分の並列化上限値は 4 である．図 6 の結果はその並列化の上限値に達した結果となっている．このような結果が得られた理由として，キャッシュを考慮したデータ構造の導入が挙げられる．従来の BGLL では，大量のノードとエッジに対する参照と Modularity 計算の両方に計算コストが生じていたが，本稿ではキャッシュを考慮したデータ構造を導入したことによりノードとエッジの参照コストを削減することに成功している．そのため，BGLL のボトルネックが参照と演算の 2 つから演算のみにシフトし，演算部分の並列化による高速化が処理全体に寄与する割合が大きくなっているからだと考察できる．

また図 6 では，特に webbase-2001 のようにノード数やエッジ数が多く，次数の分布が広いデータセットに対してより高い高速化性能を示していることがわかる．これは，SIMD 演算を用いて Modularity 変化量の計算を行う際に，SIMD レジスタ上に展開できるノード数の上限を超えるような次数を持つノードが多く存在する場合に，SIMD 命令による並列化された Modularity 計算が効果的に実行できるからだと考えられる．このような場合，SIMD レジスタの領域を残すことなくノードが展開されやすくなるため，1 回の CPU サイクルで同時に計算が可能となる隣接ノードの数が増加しやすくなる．ゆえに，webbase-2001 や uk-2005 のようなエッジ数が多く，かつ，広く分布したグラフデータである程，高速化しやくすなる傾向にあると考えられる．

次に表 2 に提案手法 (Cache+SIMD) と BGLL によるクラスタリング結果の Modularity 値を比較した結果を示す．表 2 からわかるように，いずれのデータセットに対してもほぼ同程度の Modularity 値を示していることがわかる．提案手法はデータ構造の変更と SIMD による並列化を行なっているものの BGLL 法と本質的に同様の処理を行う手法である．ゆえに表 2 にも示されたように，最終的に得られるクラスタリング結果は BGLL と同程度のものが得られる．

6.2　人工データによる評価

本実験では，次数の分布傾向を示す DIGG のパラメータ β を 2.0 に固定し，ノード数が増加した時の本手法と BGLL のスケーラビリティを評価する．図 7 に実験結果を示す．図 7 より，ノード数が 10,000 ノードや 100,000 ノード程度の時には提案手法 (Cache+SIMD) と BGLL はほぼ同程度の処理時間を要している．これはデータの規模が小さいことにより，グラフ全体に占める次数の大きなノードが少なく，並列化による高速化の効率が低下していることが理由として考えられる．しかしながら，1,000,000 ノードや 10,000,000 ノード規模のより大きなデータセットに対してはいずれのデータセットに対しても約 3 倍から 4 倍程度の高速化性能を示していることが図 7 よりわかる．この理由は 6.1 節で述べた理由と同様に，規模の大きなデータセットになるほど，大きな次数を有するノードがグラフ全体に占める割合が大きくなり，SIMD 命令による Modularity 計算によって得られる計算の並列度が高くなりやすくなるからだと考えられる．これより提案手法は，よりノード数やエッジ数が比較的大きなグラフデータに対して，従来手法である BGLL よりも約 3 倍から約 4 倍程度高速に処理可能であると考えられる．

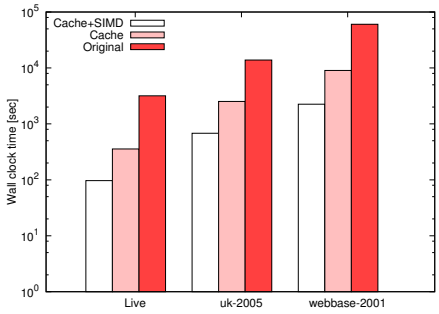

図 6　実データにより実行速度の比較

表 2　Modularity 値の比較

	Live	uk-2005	webbase-2001
Proposed	0.755	0.977	0.980
BGLL	0.754	0.979	0.981

7. おわりに

本稿では，グラフデータに対する高速なクラスタリングの実現に向けた，SIMD 命令を用いたグラフクラスタリングの並列化手法を提案した．本稿ではまず，既存のクラスタリング手法である BGLL のボトルネックが処理の第 1 パスにあることを明らかにした．そして本手法では，第 1 パスの計算時間を短縮するために，複数の命令を 1 つの CPU サイクルで実行可能な

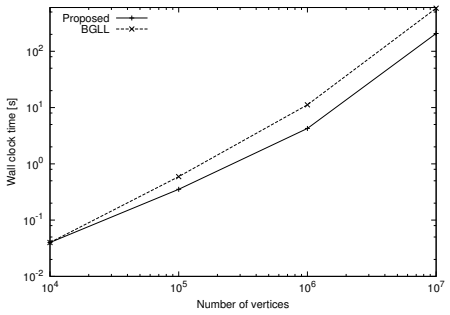

図 7　ノード数の変化による処理速度の比較

SIMD 命令をクラスタリングの Modularity 計算部に適用することによる計算時間を短縮し，メモリ上のデータ参照に対するキャッシュヒット率向上を目的としたデータ構造および索引構造を導入することでデータアクセスを効率化した．本稿では，128bit の SIMD レジスタを利用して実データセット及び人工データセットに対して計算速度の評価を行い，数百万以上のノードを持つグラフデータに対しては高速化の理論上限値である 3 倍から 4 倍程度の高速化が可能であることを確認した．本手法は，SIMD レジスタのサイズの大きさに伴い，その処理の並列度を増加させることが可能であり，将来普及する可能性のあるより大きな SIMD レジスタを搭載した計算機においてさらなる高速化が期待できる．

本研究の今後の課題として次の点が挙げられる．

a） 並列度の向上

本稿の評価実験で示したように，SIMD 命令による並列化はキャッシュを考慮したデータ構造と組み合わせることにより，並列化上限に近い高速化倍率を得られることがわかっている．そこで本研究ではこの特性を利用して，SIMD レジスタ上での並列化上限を向上させることによる本手法のさらなる高速化を検討する．具体的な方針として下記の 2 つのアプローチを検討している．

i） ハードウェアによるアプローチ

本実験で用いた CPU の SIMD レジスタのサイズは 128bit であった．しかしながら，近年発売されている各種ハードウェアには SIMD レジスタが 256bit のものや 512bit のものが存在する．今後はこれらのハードウェアを利用したさらなる並列度の向上を検討する．

ii） データ圧縮による並列度の向上

本稿ではデータの圧縮などは考慮せず，SIMD レジスタ上にノードを展開し Modularity 変化量の計算を行った．そのため，SIMD 命令による Modularity 変化量計算の並列度は SIMD レジスタのサイズと各ノード ID や各ノードの持つ次数のデータ型に依存して決定してしまう．例えば，本稿の評価実験では 128bit の SIMD レジスタに対し 32bit のデータ型を利用したため 4 並列が理論上現値であった．これに対し，ノード ID や各ノードの次数に対して圧縮を行うことで SIMD レジスタ上に展開可能なノード数の数を増やすことが可能になり，さらなる高速化が期待できる．本研究では，今後 SIMD レジスタ上での並列度向上を目的とした，グラフデータの圧縮技術について検討を進める．

b） グラフデータの特性による性能の評価

本稿では，SIMD 命令を用いたグラフクラスタリングの並列化の初期検討として，3 つの実データと 4 つの人工データを用いた性能の評価を行った．本稿の評価結果より，次数の分布の傾向や平均エッジ数などに依存して処理の並列度が影響を受ける可能性が考えられる．今後の課題として，ノードの計算順序や，グラフデータの傾向に即した並列化手法の検討を行う必要がある．

文　献

[1] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, Vol. 69, p. 026113, Feb 2004.

[2] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev. E*, Vol. 69, p. 066133, Jun 2004.

[3] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, Vol. 70, p. 066111, Dec 2004.

[4] Ken Wakita and Toshiyuki Tsurumi. Finding community structure in mega-scale social networks. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*, pp. 1275–1276, 5 2007.

[5] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, Vol. 2008, p. P10008, October 2008.

[6] Xianchao Zhang, Liang Wang, Yueting Li, and Wenxin Liang. Extracting local community structure from local cores. In *DASFAA Workshops*, pp. 287–298, 4 2011.

[7] Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Generalized louvain method for community detection in large networks. In *Intelligent Systems Design and Applications*, pp. 88–93, 11 2011.

[8] Qi Ye, Bin Wu, Zhixiong Zhao, and Bai Wang. Detecting link communities in massive networks. In *Advances in Social Network Analysis and Mining*, pp. 71–78, 7 2011.

[9] Nam P. Nguyen, Thang N. Dinh, Ying Xuan, and My T. Thai. Adaptive algorithms for detecting community structure in dynamic social networks. In *IEEE INFOCOM*, pp. 2282–2290, 3 2011.

[10] Di Wu, Tianji Wu, Yi Shan, Yu Wang, Yong He, Ningyi Xu, and Huazhong Yang. Making human connectome faster: Gpu acceleration of brain network analysis. In *16th International Conference on Parallel and Distributed Systems*, 2010.

[11] Jyothish Soman and Ankur Narang. Fast community detection algorithm with gpus and multicore architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*, 2011.

[12] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*, pp. 595–601, 5 2004.