DEIM2021 B13-4

GPU 並列 5 ノードサブグラフカウンティング手法の改良

菅波　柊也† 　天笠　俊之†† 　塩川　浩昭†† 　北川　博之 ††

† 筑波大学 理工情報生命学術院 システム情報工学研究群　〒 305–8577 茨城県つくば市天王台 1 丁目 1-1
†† 筑波大学 計算科学研究センター　〒 305–8577 茨城県つくば市天王台 1 丁目 1-1
E-mail: †suganami@kde.cs.tsukuba.ac.jp, ††{amagasa,shiokawa,kitagawa}@cs.tsukuba.ac.jp

あらまし　サブグラフカウンティングは，ターゲットとなるグラフから，高々数ノードで構成される小さなグラフ（パターン）の出現回数を求める問題であり，グラフ分析の基本的な手法の一つとして，コンピュータサイエンスやバイオインフォマティクスなどの様々な分野で利用されている．サブグラフカウンティングの手法として高速なものに ESCAPE がある．ESCAPE はパターンを分割し，組合せを利用することにより，5 ノードまでのパターンに対して効率的にサブグラフカウンティングを行う．しかしながら，ESCAPE は大規模なグラフに対しては，実行に多くの時間を必要とするという問題点が存在する．我々の先行研究により ESCAPE を（1）特定の構造について情報抽出，(2) 情報を集約し数え上げの 2 ステップに分け，ステップ 2 を GPU を用いて並列に実行することにより高速化を行った．本稿においては，先行研究で並列化を行っていないステップ 1 を GPU を用いて並列化することにより，サブグラフカウンティングの高速化を行う．また，実世界のグラフを用いた評価実験により，提案手法の有効性を示す．

キーワード　サブグラフカウンティング，GPGPU

1 はじめに

グラフ構造はデータをノードとエッジで表したデータ構造である．グラフは実世界において様々な場面で利用されており，グラフを分析することにより，様々な情報を抽出する技術が注目を集めている．一方で，近年ではデータは大規模化しており，ノード数が数億を超えるグラフも現れてきている．例えば，Instagram では 2018 の一ヶ月あたりのアクティブユーザー数が 10 億を超えており，ユーザをノード，ユーザ間のフォロー/フォロワー関係をエッジで表したグラフを考えると大規模なグラフになることが分かる [1]．今後もデータの規模が大きくなっていく事を考えると，大規模なグラフに対しても分析を行うためには，より高速な分析手法が必要となる．

グラフ構造の分析手法の一つとして，サブグラフカウンティングが挙げられる．グラフにはトライアングルやクリークのような様々な構造のサブグラフが存在する．サブグラフカウンティングでは，これらのサブグラフのグラフ中の出現回数を求める．サブグラフカウンティングの結果は，複数のグラフで同じ値を取りうる直径などのグローバルな特徴に比べ，よりグラフ特有のローカルな特徴を表すことができ，コンピュータサイエンス，バイオインフォマティクス，ソーシャルサイエンスなどの様々な分野で用いられている [2] [?].

しかし，既存のサブグラフの数え上げアルゴリズムの多くは実行に多くの時間を要するという問題点が存在する．この問題は特に，5 ノード以上のサブグラフの数え上げに対して顕著に現れる．これは，5 ノード以上のサブグラフの数え上げにおいて，組合せ爆発が起こることに起因する．エッジ数が数百万程度のグラフであっても，5 ノードの各サブグラフは数十億から数兆個存在する．さらに，サブグラフカウンティングを行うためには，実際のサブグラフの出現回数よりも多くの候補について，探索する必要がある．そのため，ノード数が 5 を超える場合，サブグラフの数え上げを行うためには特に多くの時間を要する．

5 ノードまでのサブグラフの数え上げを行う高速な手法として，Pinar らによる ESCAPE [3] がある．ESCAPE は組合せを用いることにより，5 ノードまでのサブグラフを効率よく数え上げる．また，我々の先行研究 [4] において，ESCAPE のアルゴリズムを (1) 特定のパターンについて情報抽出，(2) 情報を集約し数え上げの 2 ステップに分け，ステップ 2 を GPU を用いて並列に実行することにより高速化している．しかしながら，ESCAPE も我々の先行研究も依然として，グラフによっては探索に数時間から数日を要してしまう．そのため，より高速に 5 ノードまでサブグラフカウンティングを行う事が必要がある．

そこで，本稿では GPU を用いて 5 ノードまでのサブグラフカウンティングの高速化行う．より具体的には，ESCAPE の数え上げのアルゴリズムを基に，我々の先行研究において，並列化されていないステップ 1 を GPU を用いて並列化することにより高速化を行う．提案手法の性能を評価するため，実世界のグラフデータセットを用いて ESCAPE と我々の先行研究との実行時間の比較実験を行なった．その結果，提案手法は ESCAPE に比べ少なくとも最大で約 39 倍の高速化，我々の先行研究に比べて，少なくとも最大で 13 倍の高速化を達成した．

本稿の構成は以下の通りである．まず 2 節で前提となる知識について，3 節において GPU コンピューティングについて説明する．その後，提案手法である GPU を用いたサブグラフカウンティングについて 4 節で述べ，その評価実験を 5 節で行う．6 節で関連研究を紹介し，最後に 7 節で本稿のまとめを述べる．

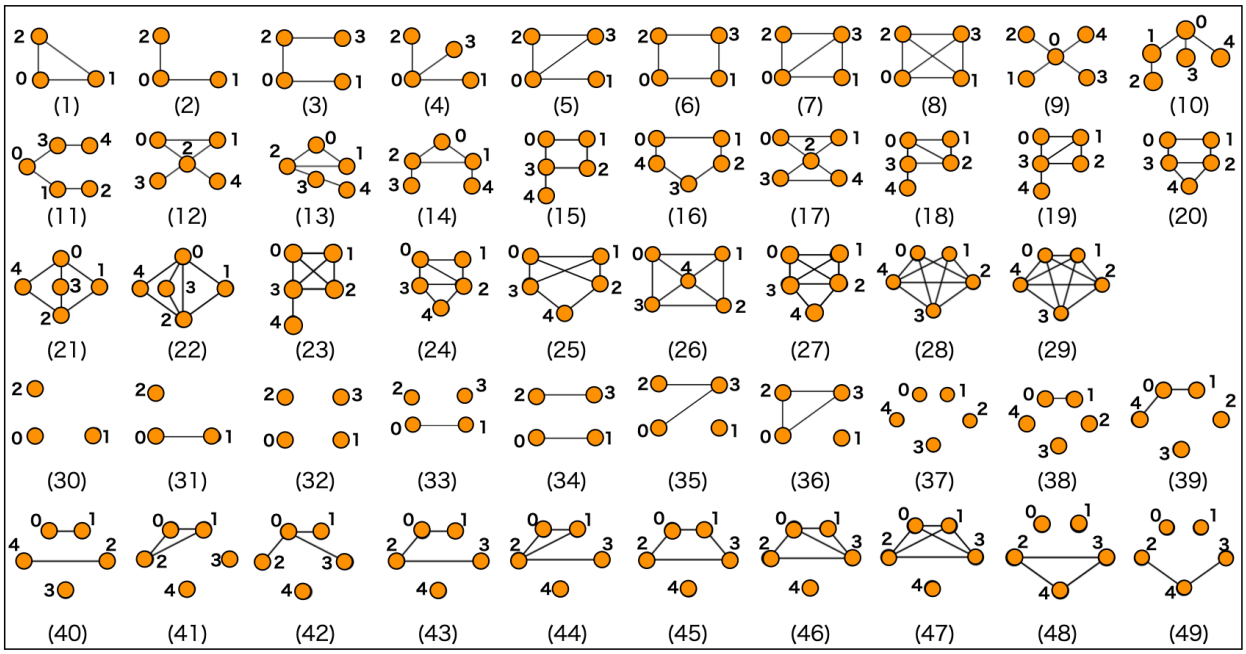


図 1: 3, 4, 5 ノードで作ることができる全てのグラフ

2 前提知識

本稿では連結なラベルなしのシンプルグラフ $G = (V(G), E(G))$ を対象とする．ここで $V(G), E(G)$ はそれぞれグラフ G 中のノード集合，エッジ集合であり，シンプルグラフとは自己ループ，重複エッジを持たない重みなし無向グラフである．ノード $v \in V(G)$ における隣接ノード集合を $N(v)$ と表す．また，数え上げを行う構造を表すグラフをパターンと呼び，$H = (V(H), E(H))$ と表す．ただし $V(H)$ はパターン H のノード集合，$E(H)$ はパターン H のエッジ集合である．図 1 に 5 ノードまでで作ることができる全てのグラフを示す．図 1 におけるグラフの i 番目のグラフを H_i と表す．例えば，図 1 の 8 番目のグラフである 4 部クリークは H_8 である．また，図 2 にいくつかのグラフの名称を示す．

次に，本稿で用いるいくつかの用語を定義する．

定義 1. (誘導部分グラフ) グラフ $G' = (V(G'), E(G'))$ を $G = (V(G), E(G))$ のサブグラフとする．$\forall u, v \in V(G')$ に対して，$(u, v) \in E(G)$ ならば $(u, v) \in E(G')$ を満たす時，G' を $V(G')$ により誘導される部分グラフと呼ぶ．

グラフ G において，ノード集合 V により誘導される部分グラフを $G[V]$ と表す．

定義 2. (DEGREE ORDERING) グラフ $G = (V(G), E(G))$ を考える．$u, v \in V(G)$ $(u \neq v)$ において，$|N(u)| < |N(v)|$，または，$|N(u)| \leqq |N(v)|$ かつ $u < v$（すなわち，ノード番号による比較）である時，$u \prec v$ と表し，$\prec$ を Degree ordering と呼ぶ．

ノード $u \in V(G)$ に対して，$v \prec u$ であるようなノードの集合，つまり，集合 $\{v \mid v \in V(G), u \prec v\}$ を $N^+(u)$ と表し，集合 $\{v \mid v \in V(G),\ v \prec u\}$ を $N^-(u)$ と表記する．

定義 3. (同型) 2 つのグラフ $G = (V(G), E(G))$，$G' = (V(G'), E(G'))$ を考える．$(u, v) \in E(G)$ である時に限り，$(\phi(u), \phi(v)) \in E(G')$ である全単射 ϕ が存在する時，G' は G と同型であるという．

定義 4. (自己同型) グラフ $G = (V(G), E(G))$ において，$(u, v) \in E(G)$ である時に限り，$(\phi(u), \phi(v)) \in E(G)$ である全単射 ϕ が存在する時，自己同型と呼び，G と自己同型なグラフの集合を $AUT(G)$ と表す．

例えば，図 1 のノード番号を用いると，H_5 には $\phi_1(0) = 0$, $\phi_1(1) = 1$, $\phi_1(2) = 2$, $\phi_1(3) = 3$ である全単射 ϕ_1 と $\phi_2(0) = 0$, $\phi_2(1) = 1$, $\phi_2(2) = 3$, $\phi_2(3) = 2$ である全単射 ϕ_2 が存在するため，$|AUT(H_5)| = 2$ である．

次に，本稿で対象とするサブグラフカウンティングを以下に定義する．

定義 5. (サブグラフカウンティング) 要素が互いに非同型なグラフ集合 $\mathcal{G}$ とグラフ G を考える．全ての $g \in \mathcal{G}$ に対して，G 中の同型な部分グラフの数を求めることをサブグラフカウンティングという．

2.1 問題定義

本稿の目的は，グラフ G と図 1 に示した 5 ノードまでのグラフの集合 $\mathcal{H}$ に対してサブグラフカウンティングを行うことである．

先行研究 [3] によって，k ノードの全ての非連結なパターンのサブグラフカウントは，k ノードの全ての連結なパターンのサブグラフカウンティングの結果から，簡単な変換により求められることが示されている．そのため，本稿の目的を達成するためには，グラフ G と $\mathcal{H} = \{H_i \mid i \in \mathbb{N},\ 1 \leqq i \leqq 29\}$ に対して，

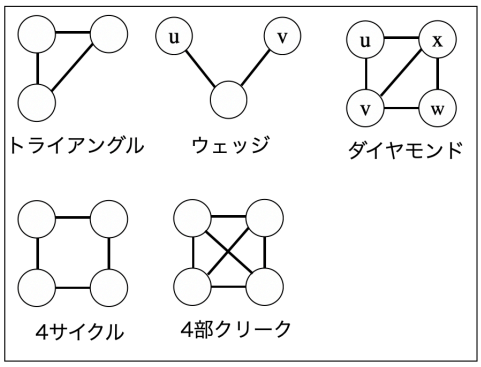


図 2: いくつかの基本的なパターンの名称

サブグラフカウンティングを行えば十分である．また，パターン H_i のサブグラフカウンティングの結果を f_i と表記する．

2.2 ESCAPE

本節では，提案手法のベース手法である ESCAPE について述べる．ESCAPE は，現在の 5 ノードまでのサブグラフカウンティングの state-of-the-art な手法であり，数え上げの効率よく数え上げを行うために 2 つのアイデアを利用する．

1 つ目のアイデアはパターンをより小さなパターンに分割する事である．クリークを除く全てのパターンにおいて，いくつかのノードを取り除くとそのパターンをいくつかの連結な k ノード以下のグラフに分割できるようなノード集合が存在する (これらのノード集合をカットセットと呼ぶ)．これらを数え上げ，組合せを利用することでパターンの数え上げを行うことができる．分割の詳細については 2.2.1 節で述べる．

2 つ目のアイデアは無向グラフ G を DAG (Directed Acyclic Graph) $G^{\rightarrow}$ へ変換し計算量を削減する事である．このアイデアは多くのトライアングルカウンティングのアルゴリズムにおいて用いられてきた [5]．ESCAPE では Degree ordering に従って，以下のように G を DAG $G^{\rightarrow}$ に変換する．エッジ (u, v) において，$u \prec v$ の時，u を始点，v を終点とする有向エッジに変換する．G のすべてのエッジについて，これを適用する事に $G^{\rightarrow}$ を得る．ここで，Degree ordering は全順序関係であるため，変換後の有向グラフが DAG である事は保証される．

ESCAPE ではこれらのアイデアを組み合わせ，パターンをより小さいパターンに分割し，その分割したパターンについて $G^{\rightarrow}$ を探索することで数え上げを行う．

2.2.1 分割フレームワーク

パターンをより小さなパターンに分割するフレームワークについて説明する．はじめにマッチ [1]を定義する．

定義 6. (マッチ) パターン $H = (V(H), E(H))$ とグラフ $G = (V(G), E(G))$ を考える．$S \subseteq V(G)$ とする．全単射 $\pi : S \to V(H)$ において，$\forall u, v \in S$ に対して $(\pi(u), \pi(v)) \in E(H)$ である時 $(u, v) \in E(G)$ であるならば π を H のマッチという．G 中の異なる H のマッチの集合を $match(H)$ と表す．

1 : 定理の一貫性を保持するため，ESCAPE におけるマッチの定義から一部を変更した．

定義よりパターン H のグラフ中のサブグラフとしての出現回数は $|\mathrm{match}(H)|/|\mathrm{AUT}(H)|$ である．$\mathrm{AUT}(H)$ は事前に求めることができるため，$match(H)$ を求めることができれば H のグラフ中のサブグラフとしての出現回数を求めることができる．また π が単射である，つまり $|S| < |V(H)|$ であるとき，π を部分マッチという．

定義 7. (拡張) 部分マッチ $\sigma : T \to V(H)$ とマッチ $\pi : S \to V(H)$ において，$T \subset S$ かつ $\forall t \in T, \pi(t) = \sigma(t)$ である場合，部分マッチ σ はマッチ π に拡張されるという．

定義 8. (H-*degree*) σ を G 中の H の部分マッチとする．σ を拡張し H のマッチとなる数を σ の H-*degree* といい，$\deg_H(\sigma)$ と表す．

次に H を小さなパターンに分割することにより得られるフラグメントについて定義する．

定義 9. (C-フラグメント) C をカットセットとする．カットセットの定義より，H から C を取り除くと H はいくつかの連結な要素 $S_1, S_2, \ldots, S_n$ に分割される．この分割されたそれぞれの要素と C の和集合によって誘導される H の部分グラフを H の C-フラグメントといい，C-フラグメントの集合を $\mathrm{Frag}_C(H)$ と表す．

ここで $H[C]$ のマッチ σ を考える．σ を H に拡張するためには，σ を $Frag_C(H)$ の全ての要素に対して拡張できれば十分である．σ を拡張した $Frag_C(H)$ の要素 F_i, F_j $(i \neq j)$ において C を除いた F_i と F_j のノード集合が互いに素である場合，σ を拡張した $F_1, F_2, \ldots, F_{|Frag_C|}$ を合併すると H のマッチとなる．一方で，F_i と F_j が互いに素でない場合，$F_1, F_2, \ldots, F_{|Frag_C|}$ を合併させた物は H とは異なるパターン H' となる．この H' を *shrinkage* と呼ぶ．

定義 10. (C-*shrinkage*) C をカットセット，$\mathrm{Frag}_C(H)$ の要素を $F_1, F_2, \ldots$ とする．パターン H を異なるパターン H' にする C-*shrinkage* を以下を満たす集合 $\{\sigma, \pi_1, \pi_2, \ldots, \pi_{|Frag_C(H)|}\}$ とする．

- $\sigma : H[C] \to H'$ が H' の部分マッチである．
- 各 $\pi_i : F_i \to H'$ が H' の部分マッチである．
- 各 π_i は σ を拡張したものである．
- H' の全てのエッジ (u, v) について，$\pi_i(a) = u$ かつ $\pi_i(b) = v$ であるような $F_i \in Frag_C(H)$ のエッジ (a, b) が存在する．

H からの C-*shrinkage* が少なくとも一つ存在するパターン H' の集合を $Shrink_{C(H)}$ とする．$H' \in Shrink_C(H)$ について，互いに素な C-*shrinkage* の数を $\mathrm{numSh}_C(H, H')$ と表す．

次に $\mathrm{match}(H)$ を求めるための補題を示す．この補題より全ての $H[C]$ の σ の $\deg_H(\sigma)$，全ての C-フラグメント $\mathrm{Frag}_C(H)$，起こりうる全ての shrinkage の出現回数が分かれば $\mathrm{match}(H)$ を求めることができる．

補題 1. C をカットセットとする．

$$match(H) = \sum_{\sigma \in match(H[C])} \prod_{F \in Frag_C(H)} deg_{F(\sigma)} - \sum_{H' \in Shrink_C(H)} numSh_C(H, H') \cdot match(H')$$

3 GPU コンピューティング

GPU コンピューティングとは，GPU (Graphics Processing Unit) を汎用の計算に用いることである．GPU は本来はグラフィック向けに開発されたデバイスであるが，CPU に比べ多くのコアが搭載されているため，高速に並列処理を行うことが可能である．そのため近年では，その並列性を用いて科学技術計算や機械学習など様々なアプリケーションにおいて，処理の高速化に貢献している．

一般に，CPU での逐次実行アルゴリズムを GPU を用いて並列に拡張することは簡単な問題ではない．GPU は CPU とは独立した特有の階層的なメモリを持ち，CUDA や OpenACC，OpenCL などの環境でコードを記述する必要がある．GPU コンピューティングにおいて，高い性能を発揮するためには，このような GPU の特徴を理解し，適切なプログラム設計を行う必要がある．

本稿では，NVIDIA 社の GPU と OpenACC を用いた．OpenACC は，GPU プログラミングにおいて広く用いられている CUDA に比べ学習コストが低く，また既存のコードに指示文を追加するだけでアクセラレータで実行できるため，可搬性や保守性，生産性に優れ，近年注目を集めている．以下では NVIDIA 社の GPU の構造と特徴，及び OpenACC を用いたプログラムについて説明する．

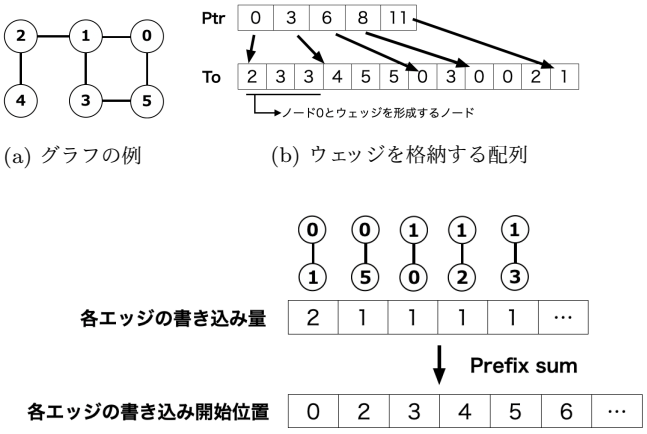
3.1 NVIDIA GPU

GPU は複数の SM (Streaming Multiprocessor) から成り，各 SM は数十から数百の SP (Scalar Processor) を搭載している．SP は CUDA コアや単にコアと呼ばれる．例えば NVIDIA Tesla V100 では，80 個の SM を持ち，各 SM は 64 個の FP32 コア，32 個の FP64 コアを含む．

GPU には大きく分けて，グローバルメモリ，シェアードメモリ，レジスタの 3 種類のメモリが存在する．グローバルメモリは 3 種類のメモリの中で最も容量の大きいメモリである．グローバルメモリは GPU 上の全ての SP からアクセス可能であるが，シェアードメモリやレジスタと比較するとアクセスが低速である．シェアードメモリはグローバルメモリよりも容量が小さく，同じ SM 上の SP からのみしかアクセスできないという制約はあるが，グローバルメモリよりも高速にアクセスできる．レジスタは 3 種類のメモリの中で最も高速なアクセスが可能であるが，最も容量の小さいメモリである．レジスタは同じ SM 上の SP からのみアクセス可能である．

3.2 OpenACC

OpenACC はメニーコアアクセラレータ向けの並列プログラミング規格である [6]．OpenMP と同様に，OpenACC はディレクティブベースのプログラミングモデルである．プログラマは C/C++・Fortran のコードに対して，並列に実行したい領域に対して指示文を挿入する．OpenACC に対応したコンパイラは，指示文に基づいて指定された領域に対して，GPU のようなアクセラレータによって並列実行が可能なコードを生成する．



(a) グラフの例　(b) ウェッジを格納する配列

(c) 書き込み位置の計算

図 3: ウェッジ配列の例

OpenACC の指示文には主に，並列領域指示文，データ指示文，ループ指示文の 3 つがある．並列領域指示文では並列に実行する領域を指定する．データ指示文では CPU とアクセラレータ間のデータの転送を指示する．一般にアクセラレータは CPU とは独立したメモリを持つ．そのためアクセラレータ上で実行するためには，事前に CPU 側からアクセラレータ上へデータを転送しておく必要がある．CPU とアクセラレータ間のデータ転送は実行のボトルネックになりうるため，ループ指示文は並列化の方法や並列の粒度を指示する．OpenACC の並列の粒度には gang，worker，vector の 3 段階が存在する．各ギャングは 1 つ以上の worker を持ち，各 worker は vector を用いてベクトル演算を行う．

4 提 案 手 法

本節では，GPU を用いた 5 ノードまでのサブグラフの数え上げについて述べる．提案手法では，前節で述べた ESCAPE の分割フレームワークに基づいた数え上げアルゴリズムを，GPU 用いて並列に実行することにより，5 ノードまでのサブグラフの数え上げを高速化する．提案手法は，我々の先行研究 [4] と同様に (1) 特定のパターンについて情報抽出，(2) 情報を集約し数え上げの 2 ステップから成る．異なる点は我々の先行研究がステップ 1 を CPU，ステップ 2 を GPU で行っているのに対し，提案手法はどちらのステップも GPU で行うことである．さらに，提案手法では，一部のパターンおいて，ステップ 2 おけるアルゴリズムを見直すことで，より高速なカウンティングを実現している．

4.1 ステップ 1

このステップではトライアングル，ウェッジ，ダイヤモンドの情報をグラフから抽出する．より具体的には各ノード，エッ

ジ毎にトライアングルとなるノード，各ノード毎にウェッジを形成するノード，各ウェッジ毎にダイヤモンドを形成するノードを記録する．ここで，あるノードとウェッジを形成するノードとは図 2 のウェッジにおいて，ノード u に対してノード v にあたるノードである．ただし u, v 間の接続は問わない．また，あるウェッジにおけるダイヤモンドを形成するノードとは図 2 のダイヤモンドにおいて，ウェッジ (u, v, w) に対してノード x にあたるノードである．ただし u, w 間の接続は問わない．実際の探索は 2.2 節で述べたように，DAG へ変換したグラフに対して行う．そのため，トライアングルやウェッジ，ダイヤモンドについても，エッジの方向を考慮する必要がある．しかしながら，エッジの方向を考慮した場合でも同様の考え方を適用することができるため，説明の平易化のために，このステップの以降においてはエッジの方向を考慮しない．

4.1.1 データ構造

グラフと同様に，トライアングル，ウェッジ，ダイヤモンドは CSR (Compressed Sparse Row) 形式に基づいた表現法により表す．CSR はグラフを表現するために一般的に使用されているデータ構造であり，重みなしグラフの場合 Ptr 配列と To 配列からなる．To 配列は各頂点の隣接ノードを格納しており，Ptr 配列はある頂点の隣接ノードが To 配列中のどこに格納されているかを表す．例えば，図 3 (a) のグラフに対してウェッジは図 3 (b) のように表される．図 3 (a) のグラフにおいて，ノード 0 はノード 2 と $(0, 1, 2)$ の一つのウェッジを形成し，ノード 3 と $(0, 1, 3)$, $(0, 5, 3)$ の二つのウェッジを形成する．そのため，図 3 (b) における To 配列のノード 0 とウェッジを形成するノードは 2，3，3 となる．

4.1.2 配列の書き込み手順

並列に To 配列を作成するために，各スレッドは共有の To 配列に書き込む必要がある．しかしながら，単純にエッジ毎にスレッド並列化を行い，各スレッドが共有の配列に書き込みを行うと，スレッド間の動作は非同期であるため，書き込み位置が衝突する可能性がある．

提案手法では 2 つの手順によって，予め各スレッドの書き込み位置を求めることでこれを回避する．ここではスペースの都合上，ウェッジの配列についてのみ述べるが，トライアングル，ダイヤモンドについても同様のアイデアを適用することができる．ウェッジの配列の作成では，各エッジ単位でスレッド並列化を行う．そのため，一つ目の手順として，図 3 (c) の上の配列のように各エッジ毎の書き込み量を計算する．ウェッジにおける各エッジ毎の書き込み量は，エッジ (u, v) において，始点ノードを u とすると $(|N(v)| - 1)$ で求めることができる．例えばエッジ $(0, 1)$ において，ノード 1 の次数は 3 であるから，このエッジの書き込み量は 2 となる．続いて図 3 (c) の下の配列のように一つ目のステップで求めた書き込み量に対して Prefix sum を求める．ここで Prefix sum は配列 $[a_0, a_1, \ldots, a_{n-1}]$，二項演算子 $\oplus$ を入力として受け取り，配列 $[I, a_0, a_0 \oplus a_1, \ldots a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}]$ を返す [7]．I は単位元である．本稿においては，Prefix sum といった場合，二項演算子として加算を与えた Prefix sum を表す．この結果は，各スレッドの書き込み開始位置を表している．

Algorithm 1 ウェッジの作成

Input: ノード集合 V, エッジ集合 E
Output: Ptr, To

```
 1: for each u ∈ V do in parallel
 2:     Ptr[u] ← 0 // 初期化
 3: end for
 4: // 手順 1
 5: for each e_i = (u, v) ∈ E do in parallel
 6:     Ptr[u] ← Ptr[u] + 1
 7:     wedgeNum[e_i] ← |N(v)| - 1
 8: end for
 9: // 手順 2
10: Ptr ← Prefix sum(Ptr)
11: writePos ← Prefix sum(wedgeNum)
12: // To 配列の作成
13: for each e_i = (u, v) ∈ E do in parallel
14:     writeIndex ← writePos[e_i]
15:     for each w ∈ N(v) − u do
16:         To[writeIndex] ← w
17:         writeIndex ← writeIndex + 1
18:     end for
19: end for
20: // To 配列をソート
21: To ← Segmented sort(Ptr,To)
```

そのため，これを用いることにより，各スレッドの書き込み位置が衝突することなく，並列に書き込みを行う事ができる．アルゴリズム 1 にウェッジ作成のアルゴリズムを示す．アルゴリズム 1 の 6 行目から 9 行目において，Ptr 配列には各ノードの始点ノードとしての出現回数，$wedgeNum$ 配列には各エッジ毎のウェッジの数を格納する．その後 11 行目，12 行目において，Ptr 配列と $wedgeNum$ 配列それぞれに対して Prefix sum を求める．その結果，Ptr 配列は，To 配列における各ノードのウェッジ開始を，$writePos$ 配列は To 配列における各エッジの書き込み開始位置を表す．13 行目から 19 行目において， 各エッジ毎にスレッド並列化し，ウェッジを探索することで To 配列を作成する．14 行目の変数 $writeIndex$ は各スレッド毎のプライベートな変数であり，各スレッドの書き込み位置を表す．最後に 21 行目において，To 配列に対して Segmented sort を行う．ここで Segmented sort は 2 つの配列 $S = [s_0, s_1, \ldots, s_m]$ と $A = [a_0, a_1, \ldots, a_{n-1}]$ を入力として受け取り，A の各部分配列 $[a_{s_i}, a_{s_i+1}, \ldots, a_{s_{i+1}-1}]$ をそれぞれソートする．21 行目では Ptr 配列，To 配列はそれぞれ S，A に相当する．

ステップ 1 では To 配列を並列に作成するために，あらかじめ各スレッド毎の書き込み量を求め，Prefix sum を行う必要があるため，逐次実行により To 配列を作成する時に比べ，追加のコストを要する．しかしながら，各スレッド毎の書き込み量は，スレッド毎に独立であるため並列に求めることができる．さらに，Prefix sum は GPU において，高速に動作することが知られており [7]，これらの処理のオーバヘッドは小さいと言える．そのため全体でみると，これらのオーバヘッドに比べ，To 配列を並列に作成することにより得られる高速化の方が大きいため，ステップ 1 を高速化することができる．

4.2 ステップ 2

このステップでは，ノード数やエッジ数，次数などのグラフの基本的な情報とステップ 1 の情報を利用して数え上げを行う．パターン H の探索において，カットセットを C とすると，探索はグラフ中の全ての $H[C]$ のマッチに対して行う．例えば，$H[C]$ がエッジの場合，グラフ中の全てのエッジが $H[C]$ のマッチとなるため，全エッジについて探索を行う．提案手法では，この探索が各 $H[C]$ のマッチ毎に独立であることに着目し，基本的には $H[C]$ 単位でスレッド並列化を行う．ただし，グラフの次数分布の偏りを考慮し，$H[C]$ が単一のノードであってもエッジ単位で並列化することができる場合，エッジ単位でスレッド並列化を行う．また $H[C]$ が非連結となる場合は，ノードまたはエッジ単位でスレッド並列化を行う．各パターンにおいて，どのノードをカットセットとするかは探索に大きな影響を与えるため，適切なノードをカットセットとして選択する必要がある．提案手法では全てのパターンで，ESCAPE においてカットセットされているノードをカットセットとした．我々は，5 ノードまでの各パターンに対するカウンティングアルゴリズムを構築しているが，スペースが限られているため，ここでは H_{15} についてのみ述べる．

H_{15} の数え上げ: H_{15} の数え上げについて述べるために，図 1 における (15) のノード番号を用いる．H_{15} において，カットセットを $C = \{3\}$ すると，C-フラグメントはエッジ $(3,4)$ と 4 サイクル $\{(0,1),(1,2),(2,3),(3,0)\}$ の 2 つであり，$Shrink_C(H_{15}) = \{H_6, H_7\}$ である．$numShc(H_{15}, H_6) = 2$, $numShc(H_{15}, H_7) = 1$ であるため，補題 1 より，

$$match(H_{15}) = \sum_{u \in V(G)} |N(u)| \cdot 2 \cdot C_4(u) - 2 \cdot match(H_6) - match(H_7)$$

となる．ただし，$C_4(u)$ はノード u が含まれる 4 サイクルの数を表す．同様の議論から，$match(H_6) = 8 \cdot f_6$ であり，$match(H_7) = 4 \cdot f_7$ であることが容易に導けるため，$8 \cdot f_6 = 2\sum_{u \in V(G)} C_4(u)$ に注意すると，

$$\begin{aligned} match(H_{15}) &= \sum_{u \in V(G)} 2 \cdot C_4(u) \cdot (|N(u)| - 2) - 4 \cdot f_7 \\ &= \sum_{u \in V(G)} \sum_{v \in W_u} 2 \cdot \binom{W(u,v)}{2} \cdot (|N(u)| - 2) - 4 \cdot f_7 \end{aligned}$$

となる．ただし，W_u はノード u とウェッジを形成するノード集合であり，$W(u,v)$ はノード u がノード v と形成するウェッジの数を表す．例えば，図 3 (a) のグラフにおいて，ノード 0 はノード 3 と 2 つのウェッジを形成するため $W(0,3) = 2$ である．$|AUT(H_{15})| = 2$ であることから，

$$f_{15} = \sum_{u \in V(G)} \sum_{v \in W_u} \binom{W(u,v)}{2} \cdot (|N(u)| - 2) - 2 \cdot f_7$$

を得る．ステップ 1 のウェッジの結果から任意の $u, v \in V(G)$ について $W(u,v)$ は求めることができ，f_7 についてもステップ 1 で求めたダイヤモンドの結果から計算することができる．そのため，ステップ 1 の情報を用いて，これらの探索を並列に行うことで f_{15} を求めることができる．

表 1: データセット

データセット	$\|V\|$	$\|E\|$	$\|T\|$
socfb-B-anon	2.94M	41.9M	52.0M
socfb-A-anon	3.10M	47.3M	55.6M
socfb-uci-uni	58.8M	184M	6.38M
socfb-konect	59.2M	185M	6.38M
soc-lastfm	1.19M	9.04M	3.95M
soc-pokec	1.63M	22.3M	32.6M
soc-sinaweibo	58.6M	522M	213M
wiki-topcats	1.79M	50.9M	52.1M
wiki-en-cat	1.85M	7.59M	2.54K
wiki-Talk	2.39M	9.32M	9.20M
com-amazon	335K	1.85M	667K
com-youtube	1.13M	5.97M	3.06M
com-orcut	3.07M	234M	628M
web-wiki-ch-internal	1.93M	8.5M	18.2M
web-hudong	1.98M	14.4M	21.6M
web-baidu-baike	2.14M	17.0M	25.2M
tech-as-skitter	1.69M	28.8M	28.8M
tech-ip	2.25M	21.6M	2.3M

5 評価実験

本節では，実世界のグラフを用いた実験により，提案手法の性能を評価する．

5.1 実験設定

本実験では比較手法として，現在の CPU による 5 ノードのサブグラフの数え上げの state-of-the-art な手法である ESCAPE と ESCAPE の一部を GPU を用いて高速化した我々の先行研究（以下，ESC-GPU と表記する．）との実行時間の比較評価を行う．ESCAPE は著者らによってソースコードが公開されており，ESCAPE の実行にはそのソースコードを利用した [8]．また本実験には CPU として Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz, 64GB，GPU として NVIDIA Tesla V100，32GB を搭載している Linux サーバを使用した．提案手法と ESC-GPU は C++（OpenACC）と一部 CUDA C++ を用いて実装し，コンパイルには `pgc++ (PGI) 19.10`，`nvcc 9.2.148` を用いた．ESCAPE は C++により実装し，コンパイルには `g++ (GCC) 4.8.5` を用いた．我々は ESCAPE に対して，`pgc++`コンパイラ，`g++`コンパイラのそれぞれでコンパイルし，実行時間を比較し，その結果 `g++`コンパイラの方が高速であったため，実験には `g++`コンパイラを用いた．データセットとして，Citation Network Dataset [9] と SNAP [10] のグラフデータセットを用いた．グラフはエッジの方向を無視し，重複したエッジと自己ループを取り除いて使用する．実験に使用したデータセットの詳細を表 1 に示す．表 2 に ESCAPE と ESC-GPU，提案手法の実行時間を示す．ただし，実行時間が

6 時間を超えた場合タイムアウトとし – と表記した．

5.2 実行時間の比較

本節では比較手法と提案手法の実行時間の比較について述べる．

ESCAPE との比較: 表 2 に ESCAPE と提案手法の実行時間を示す．表 2 の結果から，すべてのデータセットに対して，ESCAPE に比べ提案手法が高速化している．提案手法は ESCAPE と比較すると，ノード数 3.10M，エッジ数 47.3M の socfb-A-anon において最も高速化しており，ESCAPE は実行に約 55 分要しているのに対し，提案手法は 1 分 30 秒程度で終了しており，約 39 倍の高速化を達成している．

ESC-GPU との比較: 表 2 に ESC-GPU と提案手法の実行時間を示す．表 2 の結果から，すべてのデータセットに対して，提案手法は ESC-GPU と比較して高速化していることが確認できる．平均すると提案手法は ESC-GPU に比べ，約 5 倍の高速化を達成している．また，ノード数 335K，エッジ数 1.85M の com-amazon に対して最も高速化しており，提案手法は約 13 倍高速にサブグラフカウンティングを実行することができている．

また，表 2 から wiki-en-cat において，ESCAPE が 17.3 秒であり，ESC-GPU が 17.6 秒，com-amazon において，ESCAPE が 2.22 秒，ESC-GPU が 4.66 秒であり，どちらのグラフに対しても ESC-GPU の方が ESCAPE よりも低速であることが確認できる．ESC-GPU はステップ 1 とステップ 2 のうち，ステップ 1 は CPU で行い，ステップ 2 のみを GPU で行う手法である．そのため，ステップ 1 に比べてステップ 2 の比率が小さい場合や，ステップ 2 において十分な並列性が確保できない場合などでは，並列に実行するためのデータの格納方式の変更やデータの転送，カーネル起動などの GPU で実行するため必要となるオーバヘッドが，ステップ 2 を並列化し GPU で実行して得られる高速化よりも大きいため，ESCAPE と比較して同程度や低速な結果を示すと考えられる．それに対して，提案手法では表 2 より，ESC-GPU では ESCAPE に対して低速化していた wiki-en-cat，com-amazon においても ESCAPE と比較して高速化していることが分かる．このことから，提案手法は既存の GPU 並列 5 ノードサブグラフカウンティング手法である ESC-GPU では高速化が難しかったグラフに対しても高速化することが可能だと言える．

表 2: 5 ノードサブグラフカウンティングの実行時間（単位は全て秒）

データセット	ESCAPE	ESC-GPU	Proposal
socfb-B-anon	2.49K	249.1	71.7
socfb-A-anon	3.24K	292.6	82.8
socfb-uci-uni	287.3	245.8	73.4
socfb-konect	255.8	215.4	64.5
soc-lastfm	212	37.5	7.2
soc-pokec	1.79K	174.1	50.1
soc-sinaweibo	-	-	5.19K
wiki-topcats	4.26K	1.25K	326.6
wiki-en-cat	17.3	17.6	3.25
wiki-Talk	1.02K	184.3	33.7
com-amazon	2.22	4.66	0.35
com-youtube	118.9	33.5	6.59
com-orcut	-	-	2.88K
web-wiki-ch-internal	1.73K	210.4	49.3
web-hudong	2.55K	396.7	80.7
web-baidu-baike	3.61K	596.7	168.0
tech-as-skitter	1.27K	321.5	112.9
tech-ip	-	18.1K	3.42K

6 関 連 研 究

サブグラフカウンティングはコンピュータサイエンスやバイオインフォマティクスなどの多くの分野において重要なタスクであり，様々な手法が提案されてきた [11]．その中で，最も基本的なサブグラフカウンティングであるトライアングルカウンティングは盛んに研究が行われており，その領域は広大となるが，スペースが限られているため，ここでは本稿と特に関連強い 4 ノード以上のサブグラフカウンティング手法について述べ，トライアングルカウンティングについては言及しない．

Milo らによる MFINDER [12] や Wernicke らによる FANMOD [13] などの，初期のサブグラフカウンティング手法では，k ノードサブグラフカウンティングにおいて，グラフ中から全ての連結な k ノードのパターンを探索し，その後それらに対して同型判定を行うことで各パターンを分類していた．このアプローチでは，グラフのサイズや k が増加するとカウンティング難しくなってしまう．そのため，より高速にサブグラフカウンティングを行うために，Ribeiro らによるトライ木を用いた手法 [14,15] や，Hočeva らによるパターン間の関係を利用し，行列演算によりカウンティングを行う ORCA [16]，その他の様々なアプローチを用いた手法 [17,18] が提案されてきた．

しかしながらこれらの手法は，依然として実行に多くの時間を必要とし，グラフが数百万ノード程度の規模になると，4 ノードのパターンのサブグラフカウンティングであっても現実的な時間で実行する事ができなかった．この規模のグラフに対する初の 4 ノードサブグラフカウンティング手法として，Ahmed らによって PGD [19] が提案された．PGD は CPU 並列によるサブグラフカウンティング手法であり，各エッジ毎にいくつかのパターンをカウントし，その結果と各パターンの関係を利用して数え上げを行う．PGD は数百万ノードのグラフに対して，効率的にサブグラフカウンティングを行う事ができるが，数えあげを行う事ができるパターンのサイズは 4 ノードまでと制限があった．前述したように，5 ノード以上のパターンに対するサブグラフカウンティングは，組合せ爆発により多くの探索を必要とする．そのため，この規模のグラフに対して，5 ノード以上のサブグラフカウンティングは困難であると考えられていた．

近年，ブレイクスルーとなる手法である ESCAPE [3] が Pinar らにより提案された．2.2 節で述べたように，ESCAPE

は 5 ノードまでのサブグラフカウンティング手法であり，数百万ノード程度のグラフに対して，数時間から数十時間で 5 ノードまでのパターンの数え上げを行う事ができる．さらに，ESCAPE は PGD に比べ，4 ノードまでのサブグラフカウンティングを高速に行う事ができる事が示されている [3]．

また，高速化のために GPU を用いた並列計算を利用した手法も存在する．Milinkovič らは，ORCA を GPU を用いて高速化した手法 [20] を提案している．しかしながら，この手法は計算量が大きく，大規模なグラフに対して適用する事ができない．Rossi らの手法 [21] は，PGD のアルゴリズムをベースにマルチ CPU-GPU を用いて，並列に数え上げ行う．また，我々の先行研究 [4] により，ESCAPE の一部を GPU を用いて並列化することにより，ESCAPE を高速化している．

7 ま と め

本稿では，GPU を用いた並列な 5 ノードサブグラフカウンティング手法を提案した．提案手法では，既存の GPU による 5 ノードサブグラフカウンティング手法における逐次実行部分に着目し，その部分を GPU により並列化することにより，サブグラフカウンティングを高速化をする．実世界のグラフを用いた評価実験により，我々の手法は 5 ノードのサブグラフカウンティングにおいて，state-of-the-art な CPU 手法と比べ，少なくとも最大で約 39 倍の高速化，我々の先行研究である既存の GPU を用いた手法と比べ，少なくとも最大で 13 倍の高速化することを確認した．

謝辞 この成果は，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP20006）の結果および筑波大学計算科学研究センターの学際共同利用プログラムによるものです．

文　献

[1] Xin L. Wong, Rose C. Liu, and Deshan F. Sebaratnam. Evolving role of instagram in# medicine. *Internal medicine journal*, 49(10):1329–1332, 2019.

[2] Wayne Hayes, Kai Sun, and Nataša Pržulj. Graphlet-based measures are suitable for biological network comparison. *Bioinformatics*, 29(4):483–491, 2013.

[3] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. ESCAPE: efficiently counting all 5-vertex subgraphs. *CoRR*, abs/1610.09411, 2016.

[4] Shuya Suganami, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Accelerating all 5-vertex subgraphs counting using gpus. In *International Conference on Database and Expert Systems Applications*, pages 55–70. Springer, 2020.

[5] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer, 2005.

[6] Openacc-standard.org. `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf`.

[7] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[8] Escape. https://bitbucket.org/seshadhri/escape.

[9] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[10] Jure Leskovec and Andrej Krevl. SNAP Datasets : Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[11] Comandur Seshadhri and Srikanta Tirthapura. Scalable subgraph counting: The methods behind the madness. In *Companion Proceedings of The 2019 World Wide Web Conference*, pages 1317–1318. ACM, 2019.

[12] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[13] Sebastian Wernicke and Florian Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[14] Pedro Ribeiro and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1559–1566, 2010.

[15] Pedro Ribeiro and Fernando Silva. G-tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28(2):337–377, 2014.

[16] Tomaž Hočevar and Janez Demšar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[17] Dror Marcus and Yuval Shavitt. Rage–a rapid graphlet enumerator for large networks. *Computer Networks*, 56(2):810–819, 2012.

[18] Sahand Khakabimamaghani, Iman Sharafuddin, Norbert Dichter, Ina Koch, and Ali Masoudi-Nejad. Quatexelero: an accelerated exact network motif detection algorithm. *PloS one*, 8(7):e68073, 2013.

[19] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining*, pages 1–10. IEEE, 2015.

[20] Aleksandar Milinković, Stevan Milinković, and Ljubomir Lazić. A contribution to acceleration of graphlet counting. In *Infoteh Jahorina Symposium*, volume 14, pages 741–745.

[21] Ryan A. Rossi and Rong Zhou. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1783–1792. ACM, 2016.