PARALLEL STREAMING GRAPH PARTITIONING UTILIZING MULTILEVEL FRAMEWORK

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Nazanin Jafari

August 2018

Parallel Streaming Graph Partitioning utilizing multilevel framework
By Nazanin Jafari
August 2018

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Cevdet Aykanat(Advisor)

Gökberk Cinbiş

Erman Ayday

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

PARALLEL STREAMING GRAPH PARTITIONING UTILIZING MULTILEVEL FRAMEWORK

Nazanin Jafari
M.S. in Computer Engineering
Advisor: Cevdet Aykanat
August 2018

Graph partitioning is widely used for efficient parallelization of a variety of applications. Streaming graph partitioning is a one pass partitioning solution provided to overcome high computation costs of offline graph partitioners. Even though these streaming algorithms can be used for successively repartitioning, aiming at further improvements in partitioning qualities, quality improvements is limited to few passes that make offline graph partitioning tools still a desirable solution for graph partitioning due to the generated high quality partitions.

We propose a multilevel approach using streaming algorithms that can alleviate tradeoff between quality and performance in graph partitioning problem. Moreover, our OpenMP based multi-threaded implementation, can generate fast and highly scalable solutions compared to *mt-metis*, a multi-threaded solution for METIS, the state-of-the-art offline high quality graph partitioning tool. Our results show that our method can produce up to fifteen times faster and more scalable results in large graph datasets. We also show that our method can improve quality of partitions significantly compared to state-of-the-art streaming graph partitioning algorithm LDG after repartitioning several times. On average we produce partitions with 29% better qualities than LDG algorithm.

Keywords: streaming graph partitioning, parallel computing,combinatorial scientific computing.

ÖZET

ÇOK DÜZEYLI YAPI KULLANARAK PARALEL AKIŞ ÇIZELGE BÖLÜMLEME

Nazanin Jafari
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Cevdet Aykanat
Ağustos 2018

Çizelge bölümleme, çeşitli uygulamaların verimli bir şekilde paralelleştirilmesi için yaygın olarak kullanılmaktadır. Akış grafiği bölümleme, çevrimdışı çizelge bölümleyicilerin yüksek hesaplama maliyetlerini aşmak için sağlanan bir geçiş bölümleme çözümüdür. Bu aktarım algoritmaları, bölümleme özelliklerinde daha fazla geliştirmeyi amaçlayan ardışık olarak yeniden bölümlendirme için kullanılabilir olsa da, kalite iyileştirmeleri, birkaç geçişle sınırlıdır. Çevrimdışı çizelge bölümleme araçlarını, oluşturulan yüksek kaliteli bölümler nedeniyle çizelge bölümleme için hala istenen bir çözüm halindedir.

Çizelge bölümleme probleminde kalite ve performans arasındaki dengeyi azaltabilen akış algoritmalarını kullanarak çok düzeyli bir yaklaşım öneriyoruz. Ayrıca Openmp tabanlı çok parçalı uygulamalarımız, son teknoloji ürünü çevrimdışı yüksek kaliteli çizelge bölümleme aracı olan METIS için çok iş parçacıklı bir çözüm olan emph mt-metis ile kıyaslandığında hızlı ve yüksek ölçeklenebilir çözümler üretebilir. Sonuçlarımız, yöntemimizin büyük çizelge veri setlerinde on beş kat daha hızlı ve daha ölçeklenebilir sonuçlar üretebildiğini göstermektedir. Ayrıca, yöntemin, birkaç kez yeniden bölümlendirildikten sonra en gelişmiş akış grafiği bölümleme algoritması LDG'ye kıyasla önemli ölçüde bölümlerin kalitesini artırabildiğini gösteriyoruz. Ortalama olarak LDG algoritmasından 29 % daha iyi niteliklere sahip bölümler üretiyoruz.

Anahtar sözcükler: Akış çizelge bölümleme, paralel hesaplama, kombinatoryal bilimsel hesaplama.

Acknowledgement

I would like to express my immense gratitude to my supervisor Prof. Dr. Cevdet Aykanat for his endless support, suggestions and valuable guidance throughout development of this study. I consider it an honour to work on this study under his supervision.

I am thankful to Asst. Prof. Gökberk Cinbiş and Asst. Prof. Erman Ayday for reading and reviewing this thesis.

I am especially thankful to Dr. Reha Oguz Selvitopi for his persistent support and valuable guidance in each and every step of developing this study.

I also acknowledge the scientific and Technical Research Council of Turkey (TÜBITAK) for supporting this study financially with project EEEAG-115E212.

I would like to thank my valuable friends Noushin, Mohammad, Hamed, Saharnaz, Ehsan, Mina, Pezhman, Nima, Zeynab and Wurya and all of my friends in Bilkent University for their emotional support and encouragement through my experiences.

I would like to express my gratitude to my Mother, Father, my little brother Amin for their love and encouragement. I would not be here without your kind and sincere support.

Lastly and more importantly very special thanks goes to my husband Iman Deznabi for his persistent support, understanding and endless love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph datasets are widely used in many areas of science such as social and biological networks, scientific computing and VLSI networks. Nevertheless, graphs grew in size rapidly in recent years, for instance twitter had over 40 million users with 1.4 billion interactions in 2009 [32]. Web graphs also increase in size up to trillions of links. Biological data such as protein or gene interactions can consist of millions of nodes and billions of edges. Considering these large datasets, essence of efficient computation becomes more vital. Graph partitioning is defined as clustering graph into different components of roughly equal size to process large graphs in distributed systems in parallel. Graph partitioning is beneficial for load balancing and reducing communication overhead. However, it is known to be an NP-hard problem [1, 2]. A good graph partitioning scheme should be able to partition a graph into roughly equal sized clusters of nodes (or number of edges) and at the same time reduce the number of edges (or nodes) between clusters. The former objective corresponds to load balancing and the latter one corresponds to reducing communication volume between processors in a parallel setting. These two objectives are usually in conflict with each other. This phenomenon is illustrated in figure 1.1

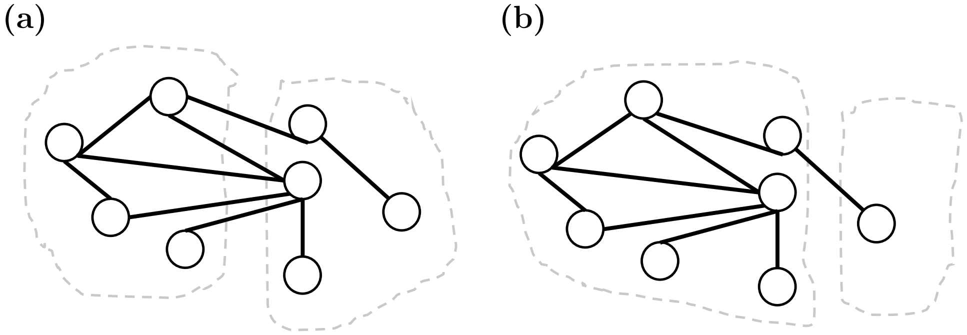

Figure 1.1: **(a)** A balanced partitioning with many edges between clusters , **(b)** Edges between clusters are minimized but partitioning is highly unbalanced

However, there exists several feasible solutions for this problem. Many offline graph partitioning tools are proposed such as Metis[4], Chaco [33], Patoh [3], Jostle[26] and Scotch[23]. There are also parallel offline graph partitioning tools such as Parmetis, [20],PT-Scotch[34], mt-metis[6]. However, these parallel graph partitioning algorithms are not sufficiently scalable. This is because the complexity of algorithms used in sequential version of these parallel graph partitioners, are usually not amenable for parallelizm and can generate performances in comparably low scalable setting leading to a need for more efficient algorithms in terms of performance.

Streaming graph partitioning algorithms have been proposed in order to overcome performance issues associated with offline frameworks. These algorithms, partition graphs as they arrive in the stream greedily, however, the quality of partitions in these algorithms are usually not comparable to the quality of partitions in offline multilevel tools. The reason is that, streaming algorithms greedily assign nodes (or edges) into parts with current information of the graph and they usually neglect whole graph structure in the beginning of stream. Moreover, these streaming graph partitioning algorithms partition graphs only once and they never move vertices (or edges) among parts targeting reducing communication between parts.

Restreaming graph partitioning [10] have been proposed for few streaming heursitics as a method to improve partitioning quality by repartitioning graphs. However, due to the nature of these greedy algorithms repartitioning does not improve the quality of partition after few passes and they stuck in local minimum. We propose a paradigm that aims at producing high quality partitions with streaming graph partitioning algorithms utilizing multilevel scheme. We also propose a parallel implementation of our framework using OpenMP library. This parallel version can yield very fast solutions in a more scalable fashion compared to shared-memory offline multilevel methods.

1.1 Contributions

We introduce a novel multilevel graph partitioning approach. Our method benefits fast and lightweight streaming graph partitioning algorithm, Linear Deterministic Greedy (LDG), best performing heuristic proposed by Stanton and Kliot [9], in partitioning graph in multilevel approach. We show that average gain of our algorithm over LDG can achieve 29% in partitioning quality given the same running time. Moreover,we propose parallel version of our multilevel framework using OpenMP, a shared memory parallel programming platform. Our method is more scalable than *mt-metis* within shorter running times on a given dataset. In the following section we provide an explanation on notations we use in this study.

1.2 Definitions

A Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined as a set of vertices $\mathcal{V} = \{v_1, v_2, ..., v_n\}$ and a set of edges $\mathcal{E}$, such that an edge represents connection between pair of vertices i.e., $e = (v_i, v_j)$. Vertices and edges can be associated with weight, i.e., $\omega(v_i)$ and $\omega(v_i, v_j)$ denote vertex and edge weight respectively.

$\Pi = \{V_1, V_2, ..., V_K\}$ is called a K-way partitioning of graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that each part is nonempty and parts are pairwise disjoint. In a partition Π an edge $e = (v_i, v_j)$ said to be a *cut* if v_i and v_j are assigned to different parts. The cutsize of a partition is defined as the sum of weights of edges that are *cut* and it is shown as $cutsize(\Pi)$. Weight of each part,V_k, $\omega(V_k)$ is equal to the sum of the weights of vertices, $\omega(v_i)$ in that partition i.e:

$$\omega(V_k) = \sum_{v_i \in V_k} \omega(v_i) \tag{1.1}$$

A partition Π is said to be balanced if it satisfies the balance constraint:

$$\omega(V_k) \leq (1 + \epsilon)\mathcal{W}_{avg} \tag{1.2}$$

Where ϵ is the maximum imbalance ratio and $\mathcal{W}_{avg}$ is equal to:

$$\mathcal{W}_{avg} = \Sigma_{v_i \in V} \frac{\omega(v_i)}{K} \tag{1.3}$$

It is important to note that in this study, index notations for vertices are given as $\{i, j, ..\}$ while adressing parts we use index notations $\{k, l, m, ..\}$.

Chapter 2

Background and related work

In this chapter, we discuss different solutions on graph partitioning problem. Our proposed method can be related to existing solutions in two ways: 1) offline graph partitioning methods, and 2) online lightweight streaming graph partitioning heuristics. We also discuss parallel versions of these graph partitioning methods. Several number of methods are available for offline graph partitioning. Among these methods multilevel scheme is proven to obtain high quality partitions, consequently many methods use this paradigm.

2.1 Multilevel frameworks

Multilevel partitioning is a successfull paradigm widely used in several state-of-the-art graph/hypergraph partitioning tools (Metis [4],Patoh [3], Scotch [23], Jostle [26], Chaco [33] and KaHIP[27]). Several parallel offline graph partitioning tools have been proposed that can alleviate performance overhead of sequential offline graph partitioning tools. Akhremtsev et al. [7] propose a shared memory multilevel graph partitioning by parallelizing label propagation algorithm [24] in coarsening phase and introducing parallel version of k-way multi try local search[27]. ParMetis [20], as a distributed memory graph partitioning tool and *mt-metis*, as the shared memory version of Metis are among the well known parallel graph partitioners. PT-Scotch [34] is a parallel version of Scotch [23], the well-known offline multilevel graph partitioning tool. However, these methods are not very scalable because algorithms used in partitioning and uncoarsening phases of offline multilevel graph partitioning are usually not amenable for parallelism and may incur problems in maintaining balance because of the sequential nature of the coarsening and refinement algorithms utilized [6].

General structure of a multilevel framework consists of coarsening the graph into smaller graphs then applying a graph partitioning algorithm on the coarsest graph and project it back to the original graph by performing a refinement algorithm on each finer graph. Since finer graphs have more degree of freedom, refinement can effectively increase quality of partition[6]. In the coarsening phase original graph $\mathcal{G}^0 = (\mathcal{V}^0, \mathcal{E}^0)$ transform into coarser graphs in a sequence. i.e., $\mathcal{G}^1, \mathcal{G}^2, ..., \mathcal{G}^L$. Several algorithms have been used for this phase, among which edge matching algorithms [4, 21, 22, 23] and label propagation algorithm [24] are mainly chosen. The coarsest graph is partitioned in *initial partitioning* phase. Various algorithms can be selected for this stage. Spectral Bisection (SB), KL algorithm, graph growing partitioning algorithm and greedy graph growing partitioning algorithm are among the most common paartitioning algorithms which are introduced in [4]. At each level of uncoarsening, a refinement algorithm [25] is applied aiming to improve the quality of the edgecut produced in the previous level partitioning phase. Due to the expensive computations costs of these offline graph partitioning tools, recently streaming graph partitioning algorithms have been proposed. In the next section we will argue the nature of these algorithms.

2.2 Streaming graph partitioning paradigms

In contrast to offline graph partitioning methods, online methods use lightweight streaming graph partitioning algorithms that can generate partitions in a sufficiently good quality without needing to keep all the graph information in the memory. These algorithms assign vertices or edges on parts as they arrive in the stream.

Stanton and Kliot [9] propose 10 heuristics among which Linear Deterministic Greedy (LDG) algorithm produce best results. While this heuristic greedily assigns vertices to the partitions with most number of vertices sharing common edge, it penalizes full partitions with a linearly weighted penalty function. Its linear penalty function can produce highly balanced partitions without neglecting the graph structure. Fennel [8], being another streaming graph partitioning algorithm, is a modularity maximization framework that approximately balances parts while assigning vertices to these parts greedily. Tsourakakis et.al [13] propose streaming graph partitioning in a planted partition model with higher length walks. Besides streaming vertex based partitioning, edge based streaming partitioning also got attention. Petroni et al. [14] proposed replicated streaming edge based graph partitioning mostly focusing on power-law graphs.

As the quality of partitioning in streaming graph partitioning algorithms are usually not comparable to the qualities produced by offline multilevel graph partitioners, Nishimura and Ugander [10] propose a multipass solution for two single pass streaming algorithms, Fennel [8] and linear deterministic greedy algorithm proposed by Stanton and Kliot [9]. aiming at improving partitioning quality. In this work a graph partitioned with state-of-the-art streaming graph partitioning algorithms can be repartitioned several times, even though repartitioning can be effective in increasing quality of graph partitioning in only few passes, the quality does not improve in the later passes.

Besides these heuristics, Firth and Missier [11] propose workload aware streaming graph partitioning algorithm. They use frequent patterns seen in the graph as workloads and partition the graph considering these motifs using the LDG algorithm [9]. Grasp [12] proposes distributed-memory streaming graph partitioning. For this purpose, it uses MPI to parallelize the framework proposed in Fennel [8].

As our proposed method is based on the lightweight greedy algorithm, LDG, in the rest of the section the selected graph partitioning algorithm will be briefly outlined. Our multilevel graph partitioning framework exploits this fast streaming algorithm for partitioning in both *coarsening* and *uncoarsening* phase. LDG algorithm is defined as a heuristic that assigns vertices into parts where it has maximum number of neighbors even though it penalize full parts with a linear penalty function. Equation 2.1 introduces LDG heuristic.

$$\arg\max_{k \in K} \{|V_k \cap Adj(v)|(1 - \frac{\omega(V_k)}{C})\} \tag{2.1}$$

V_k represents the set of vertices in part k. v is a random vertex in the stream that is ready to be assigned to a part. $Adj(v)$ represents the set of vertices that share same edge with v. C is the capacity constraint and K refers to the number of parts respectively.

This LDG partitioning heuristic computes affinity of a vertex v to parts considering its neighbors and assigns the vertex into a part with the maximum affinity score while penalizing full parts with the capacity constraint C.

Chapter 3

Multilevel Streaming Graph Partitioning Framework

In this chapter, we provide an extensive explanation on our paradigm. The general structure of our framework is simple. First the original graph is partitioned with a streaming graph partitioning algorithm, LDG. Then generated partition creates structure of the next level coarser graph i.e. each part produced in previous level partitioning become coarse vertices of the next level and part weights of previous level correspond to coarse vertex weight of the next level. The cut edges between each pair of parts are coalesced into a single edge whose weight is set to be equal to the sum of the weights of its constituent edges. The process of partitioning and coarsening is repeated based on the number of levels that was previously computed. Then in the last level, parts with coarsest vertices project back to the original graph in several steps of decomposing coarse vertices into their corresponding finer vertices in previous levels while **repartitioning** finer graph using the LDG algorithm. In the following sections different phases of our multilevel approach will be thoroughly discussed.

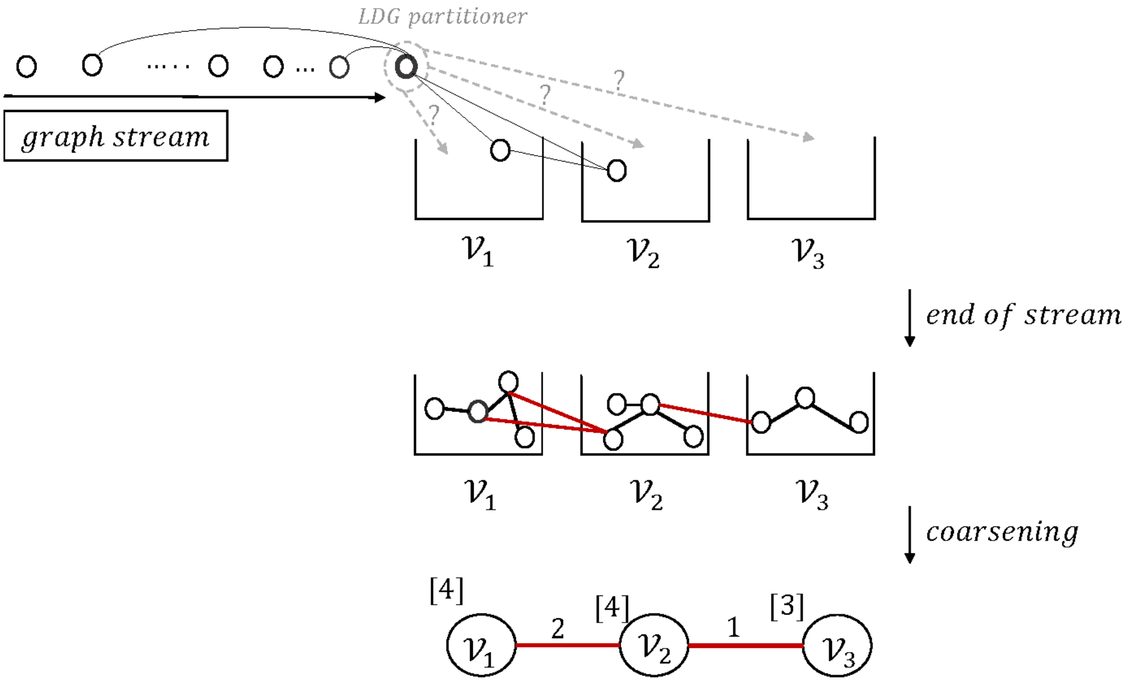

Figure 3.1: Graph G^ℓ is partitioned in a streaming order. At the end of the stream the partition with fully loaded graph constructs coarser graph.

3.1 Partitioning and coarsening

At level ℓ graph $\mathcal{G}^\ell$ is partitioned with the LDG algorithm into K^ℓ parts, then the generated partition is utilized to determine the structure of coarser graph $\mathcal{G}^{\ell+1}$. Let $\Pi^\ell = \{V_1^\ell, V_2^\ell, ..., V_{K^\ell}^\ell\}$ denote the set of K^ℓ way partitioning generated by the LDG algorithm at level ℓ. Partition Π^ℓ determines $\mathcal{G}^{\ell+1}$ with K^ℓ vertices. i.e., part k of Π^ℓ constitutes the coarse vertex v_k^c of $\mathcal{G}_{\ell+1}$, where the vertices of V_k^ℓ of Π_ℓ correspond to the constituent vertices of coarse vertex v_k^c of $\mathcal{G}_{\ell+1}$. Vertex weight of coarse vertex v_k^c of $\mathcal{G}^{\ell+1}$ is equal to the sum of the weights of constituent vertices of part k of Π^ℓ i.e.,

$$\omega(v_k^c) = \sum_{v_i \in V_k^\ell} \omega(v_i) \tag{3.1}$$

The edge-set of $\mathcal{G}_{\ell+1}$ is constituted as follows:

$$\begin{aligned} &(v_k^c, v_m^c) \in \mathcal{E}^{\ell+1} \quad \text{if} \\ &Adj(V_k^\ell) \cap V_m^\ell \neq \emptyset \end{aligned} \tag{3.2}$$

The adjacency relation of a set of vertices is given by:

$$Adj(V) = \bigcup_{v \in V} Adj(v) - V \tag{3.3}$$

The weight of edge (v_k^c, v_m^c) is determined as

$$\omega(v_k^c, v_m^c) = \sum_{\substack{v_i \in V_k^\ell \\ v_j \in V_m^\ell}} \omega(v_i, v_j) \tag{3.4}$$

The number of coarsening levels , L is computed in Equation 3.5.

$$L = \left\lfloor \log_\beta \frac{|\mathcal{V}^0|}{K} \right\rfloor \tag{3.5}$$

Here K is the number of parts to be partitioned at the end and β refers to the average number of vertices in a part and it determines the number of parts of each level i.e., $K^\ell = |\mathcal{V}^\ell|/\beta$.

It is worth noting that for each LDG partitioning we relax the strict capacity constraint of $1+\epsilon$ of balancing in the beginning of coarsening phase with a value equal to $\epsilon + \epsilon'$ such that $\epsilon \leq \epsilon'$. Reducing the strict burden of balancing constraint aids us in better decisions in partitioning. At each level we reduce the value of ϵ' by a range equal to $\frac{|\epsilon - \epsilon'|}{L}$. In uncoarsening phase capacity of parts are computed with $1+\epsilon$ constraint.

3.2 Uncoarsening and repartitioning

In the last level, when the graph $\mathcal{G}^L$ is partitioned in K-ways, our multilevel framework begin uncoarsening followed by repartitioning. Partition of last level Π^L

contain parts that hold coarse vertices of graph G^L. Number of these coarse vertices in the last level is very low while weight of each vertex is very high therefore, moving them across parts for repartitioning is nearly impossible because presence of approximately high capacity constraint for very limited number of vertices with heavy weight limits ability to move vertices among parts. Therefore, decomposition of these heavy weighted vertices are necessary for repartitioning to have impact on improvements. The partition $\Pi^\ell = \{V_1^\ell, V_2^\ell, ..., V_{K^\ell}^\ell\}$ initially obtained at level ℓ is projected back to a finer partition $\Pi^{\ell-1} = \{V_1^{\ell-1}, V_2^{\ell-1}, ..., V_{K^{\ell-1}}^{\ell-1}\}$ at level $\ell - 1$ as follows:

Consider a coarse vertex v_k^c mapped to part V_k^ℓ at Π^ℓ. Then each constituent vertex set $\mathcal{V}^{\ell-1}$ of coarse vertex v_k^c is assigned to part $V_k^{\ell-1}$ of $\Pi^{\ell-1}$. Then we perform a number of repartitioning phases using LDG algorithm on this initial partition $\Pi^{\ell-1}$ which is projected back from the resulting partition of Π^ℓ.

Chapter 4

Implementation details: Parallel Multilevel Streaming Graph Partitioning

In this chapter we broadly explore the details of our parallel multilevel streaming graph partitioning framework. Each phase of our framework is parallelized using OpenMP. In the following sections we will thoroughly argue parallel implementation details of each phase of our scheme.

4.1 Multi Threaded LDG

Simple structure of Linear Deterministic Greedy algorithm provides a platform very amenable for parallelism. Vertices of the graph are divided equally among threads. Each thread is responsible for assigning its vertices into parts. As vertex u arrives in stream, we compute its affinity to the parts that its neighbors have already been assigned. Among them a part is chosen depending on its affinity score and its capacity. Despite being simple, parallel version of LDG algorithm in shared-memory platform can be challenging in some aspects.

In our implementation, part information of each vertex of graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is shared among threads and all threads can read or write these information. Part information consists of weight value for each part k, $\omega(V_k)$ and a *part* vector showing part information for each vertex, *part* is a vector with length equal to $|\mathcal{V}|$. For every $v \in V_k$ we can define $part[v]$ in the following way:

$$part[v] = \begin{cases} k, & \text{if } v \textit{ is already assigned to } V_k.\ for\ 1 \leq k \leq K \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

Since $\omega(V_k)$ is shared among threads concurrent reading and/or writing can cause race condition. For instance, if thread T_1 assigns vertex u into part k and at the same time thread T_2 assigns another vertex v into the same part k then weight of part k might not be correctly updated since both of the threads access $\omega(V_k)$ at the same time and add *only* their own vertex weights to the part weight. In this case weight value for part k is updated erroneously. In order to prevent such race conditions, updating the part weight is done in an *atomic section* (i.e., one thread at a time can update part weight values.) The pseudocode for Multi-Threaded LDG algorithm is shown in Algorithm 1. aff is a vector that computes the affinity of adjacent vertices of vertex u to the parts. Each vertex and its neighboring vertices are divided among threads and each thread is responsible for assigning these vertices into parts in this algorithm. In our implementation we track the part information of neighbors of vertex u using *part* vector and add part info of these neighbors into *partset*, a set to store distinct parts of neighbors of u. After processing each neighbor of vertex u, using *partset* and affinity score of each part which is stored in *partset* LDG assigns vertex u into the part that has highest score according to equation 2.1 while considering the capacity constraint. However, in cases when none of the neighbors of vertex u is assigned to parts or the weight of the parts of neighbors exceed the capacity, vertex u is assigned into parts randomly.

Algorithm 1 multi-threaded LDG

```
**Input:** graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of parts $K$, capacity $C$
**Output:** $part$: part information vector for each $v \ \in \mathcal{V}$, $\omega(V)$: partition weight
    for each part $\in K$,$\omega(v)$: vertex weight for each $v \in \mathcal{V}$
for each $v \in \mathcal{V}$ in parallel do
    $part[v] = UNPROCESSED$
for $k = 1 \rightarrow K$ in parallel do
    aff $[k] \leftarrow 0$ {vector to compute affinity score for each part}
for each $u \in \mathcal{V}$ in parallel do
    for each $v \in Adj(u)$ do
        if $part[v] = UNPROCESSED$ then
            $continue$
        $k \leftarrow part[v]$
        if aff $[k] = 0$ then
            $partset \leftarrow partset \cup \{k\}$ {set of parts holding neighbors of $u$}
        aff $[k] \leftarrow$ aff $[k] + \omega(u, v)$
    $k_{max} \leftarrow 0$
    $\alpha_{max} \leftarrow 0$
    for each $k \in partset$ do
        $\alpha \leftarrow$ aff $[k] * (1 - \frac{\omega(V_k)}{C})$;
        if $\alpha > \alpha_{max}$ then
            $k_{max} \leftarrow k$
            $\alpha_{max} \leftarrow \alpha$
        aff $[k] \leftarrow 0$
    while $k_{max} = 0$ do
        $k_\xi \leftarrow rand(1, k)$
        if $\omega(V_{k_\xi}) + \omega(u) \leq C$ then
            $k_{max} \leftarrow k_\xi$
    $part[u] \leftarrow k_{max}$
    $\omega(V_{k_{max}}) \leftarrow \omega(V_{k_{max}}) + \omega(u)$; {atomic operation due to concurrent writes}
```

Considering the fact that our β parameter determines the number of levels, number of parts and capacity constraint for each level (section 4.1), choosing very small value for parameter β may result in high number of parts and low capacity constraint leading to poor edgecut quality in parallel platform as the number of threads increase. The reason is that due to the nature of streaming graph partitioning algorithm at the beginning of the stream, there is no part information about adjacent vertices of vertex u, thus the algorithm inevitably assigns vertices to parts randomly. As the neighbors of vertex u already appear in parts in stream order, LDG algorithm arrives into better decisions on vertex u.

However, in multi-threaded environment this random assignment can happen even more frequently. The reason is that since in a streaming order, vertices are divided among threads, adjacent vertices might be processed by different threads. Lets consider two adjacent vertices u and v $(u, v) \in \mathcal{E}$ that are processed near in time by two threads T_1 and T_2 respectively. Let T_1 to assign u to part V_k while T_2 is performing affinity computation regarding the assignment of v to a proper part. T_2 might miss the part assignment decision on its neighbor u and consequently assign it to a part other than V_k producing an edge-cut that in sequential runs this might not occur. This process is illustrated on figure 4.1.

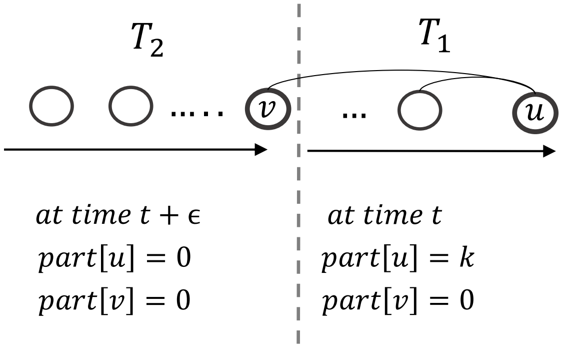

Figure 4.1: vertex u arrives into stream at time t and thread T_1 assigns vertex u into part V_k, however, thread T_2 have not received the updated part information for adjacent vertices of vertex u and assigns it randomly.

It is worth noting that disregarding the value of parameter β this phenomenon usally exist in the parallel platform of our framework. However, coarsening and uncoarsening phases significantly recover quality of partition. Nevertheless, depending on the structure of graphs being partitioned, the value of parameter β can have severe or mild effect on the quality. The effect of the value of β can be eliminated in graphs with irregular patterns (i.e., each vertex has distinct degrees). In such graphs, the probability of lack of part information for nodes reduce and threads can have better decisions on which part to assign vertices. On the other hand, in regular graphs such as meshes where each vertex in the graph have nearly the same degree, this probability increase therefore, more random assignments occur.

Moreover, random vertex process ordering is an important feature of streaming graph algorithms. However, this random processing order should be toned in a wise way to preserve spatial locality. Graph representation in our multilevel framework is based on CSR format. Vector vtx with length $n+1$ represent vertices of graph $\mathcal{G}$ and points on the adjacent vertices of graph $\mathcal{G}$. (i.e., $vtx[u], vtx[u+1]$ means that adjacent vertices of vertex u can be accessed in vector $adjncy$ in the range of $[adjncy[vtx[u]], adjncy[vtx[u+1]])$. Considering this format, accessing adjacency of each vertex should be done in a sequential order to respect spatial locality. i.e., $adjncy[vtx[u]], adjncy[vtx[u+1], adjncy[vtx[u+2]], adjncy[vtx[u+$

3]], . . . However, accessing vertices and their adjacent vertices in *random order* in streaming platform disturbs the spatial locality. i.e., in a random order instead of accessing vertices in sequential order $u, u+1, u+2 \dots$, we trace the vertices in a random order $(u, v, ...)$. This problem can destroy the advantage of parallelism as accessing *adjncy* vector in cache level would be impossible. We can permute vertices in stream order such that the spatial locality will be preserved. After permuting the order of vertices in stream, we accordingly update *adjncy* vector to coalesce with the new permutation.

4.2 Multi-threaded coarsening

The process of creating the next level coarser graph with part information of current level is shown in Algorithm 2. In order to create coarse graph in next level we need to trace vertices inside each part separately. As the only information we have as an input is the *part* vector that provides part of each $v \in \mathcal{V}^\ell$ tracing each vertex inside particular part is not possible using this vector. Hence sequentially we create a set $\mathcal{P}$ and for each part k we add vertices of part k into this set. It is crucial for this step to be implemented in sequential order since if we divide vertices of $\mathcal{V}^\ell$ among threads it is mostly probable for $\mathcal{P}_k$ to be accessed and accordingly updated by several threads as there is several number of vertices that share same part k.

Hence after obtaining the set $\mathcal{P}$ we begin parallel implementation and in this implementation parts and their corresponding vertices are divided among threads and each thread process vertices of those parts that are assigned to it. In T number of threads, each thread performs computation on approximately $\frac{K^\ell}{T}$ number of parts. As discussed in previous chapter, part weights of each part in current level correspond to vertex weight of coarse vertices in next level. Hence in designing our parallel implementation, we can simply map part weights to their corresponding vertex weights in parallel. i.e., weight of part k, $\omega(V_k^\ell)$ is equal to the weight of coarse vertex k of next level. In parallel fashion each thread reads its own part weights and write these weights into the corresponding coarse vertices. Next, coarse vertex v_k^c is added into set of vertices of level $\ell + 1$ in parallel.

Cut edges among parts create adjacency set and adjacency weight of each coarse vertex in new graph, i.e., if there is an edge between vertex u and v corresponding to part p and part k respectively, then coarse vertex v_k^c is added to the adjacency set of coarse vertex v_p^c in level $\ell + 1$, $Adj(v_k^c)$ and edge weight between coarse vertex p and k is the same as edge weight between vertex u and vertex v. Thus, in order to detect these cut edges, we trace neighbors of vertices in each part and find part information of these adjacent vertices using *part* vector. Then we can find out if vertices and their neighbors do not share same part. After

Algorithm 2 Multi Threaded coarsening

```
Input: graph $\mathcal{G}^\ell = (\mathcal{V}^\ell, \mathcal{E}^\ell)$, number of parts $K^\ell$, $part$: part information vector
    for each $v \in \mathcal{V}^\ell$, $\omega(V_k^\ell)$: partition weight vector for each part $k$
Output: coarse graph $\mathcal{G}^{\ell+1} = (\mathcal{V}^{\ell+1}, \mathcal{E}^{\ell+1})$
for each $u \in \mathcal{V}^\ell$ do
    $k \leftarrow part[u]$
    $\mathcal{P}_k \leftarrow \mathcal{P}_k \cup \{u\}$
for $k = 1 \rightarrow K^\ell$ in parallel do
    $\omega(v_k^c) \leftarrow \omega(V_k^\ell)$
    $V^{l+1} = V^{l+1} \cup \{v_k^c\}$
for $k = 1 \rightarrow K^\ell$ in parallel do
    for each $u \in \mathcal{P}_k$ do
        for each $v \in Adj(u)$ do
            $p \leftarrow part[v]$
            if $k \neq p$ then
                $Adj(v_k^c) \leftarrow Adj(v_k^c) \cup \{v_p^c\}$
                $\omega(v_k^c, v_p^c) \leftarrow \omega(v_k^c, v_p^c) + \omega(u, v)$
```

detecting *edgecuts* updating set of adjacent vertices of coarse vertices and edge weight between coarse vertex v_p^c and v_k^c is done in parallel; each thread updates adjacency list and edge weights for coarse vertices corresponding to the same parts in parallel. It should be pointed out that, in our design, the edge (v_k^c, v_p^c) is stored twice in the adjacency lists of vertex v_k^c and v_p^c, therefore they are both updated separately leading to no need for atomic operations.

It is important to mention that, tracing vertices inside *one* part is done by only one thread and other threads do not access vertices inside that particular part. This property of our parallel implementation prevents race conditions since each thread can access only vertices inside its own predetermined parts consequently can only update adjacency set and edge weights locally for their own parts. Moreover, as the number of parts in each level decreases, workload of threads also reduces such that in last levels, only a few parts are left for coarsening. In this case, obviously parallelism does not have much effect on the performance of coarsening phase. But since in initial levels of coarsening, the number of parts are huge enough, multi-threaded coarsening can effectively scale to increasing number of threads.

4.3 Multi-threaded uncoarsening

At level L where graph is partitioned in K ways and its corresponding coarse graph is created from partition at level L, uncoarsening phase can start to take effect. In parallel implementation of uncoarsening phase, vertices of the finer graph at each level $L-1, L-2, \ldots, 0$ are divided among threads. The part information of vertices in level $L-1$ can be accessed using vector $part^{L-1}$ and we also have part information of vertices in coarser graph L which is stored in vector $part^{L}$, we can access part information of vertex v at level $L-1$ using $part^{L-1}[v]$ which returns part k that corresponds to the coarse vertex v_k^c of the coarse level L, also $part^{L}[v_k^c]$ accounts for the part information of coarse vertex v_k^c. Hence we can find the most recent part of vertex v by utilizing these two part vectors and we can store part information of fine vertex v in another part vector $part^{\xi}$ by simply mapping most recent part information of fine vertex v at level L into this vector. Hence by having these two vectors we can simply decompose coarse vertices in last level and project the part back to the previous level. Part weights of coarsest level partitioning is exactly mapped to the next level finer parts. This is also implemented in parallel with dividing parts among threads. This process is shown in Algorithm 3.

Algorithm 3 Multi Threaded uncoarsening

```
Input: graph $\mathcal{G}^{L-i} = (\mathcal{V}^{L-i}, \mathcal{E}^{L-i})$,number of parts $K$, $part^{L-i}$: part information
    vector for each vertex $\in \mathcal{V}^{L-i}$, $part^{L}$: part information vector for each vertex
    $\in \mathcal{V}^{L}$ $\omega(V^{L})$: partition weight vector for each part $\in K$ at level $L$, $i =$
    $\{1, 2, ..., L\}$, uncoarsening steps.
Output: $part^{\xi}$,$\omega(V^{\xi})$
for each $v \in \mathcal{V}^{L-i}$in parallel do
    $k \leftarrow part^{L-i}[v]$
    $part^{\xi}[v] \leftarrow part^{L}[k]$
for $k = 1 \rightarrow k^{L}$ in parallel do
    $\omega(V_k^{\xi}) \leftarrow \omega(V_k^{L})$
```

Part vectors and partition weight is globally shared among threads however, as each thread locally update its share of vertices and parts for $part^{\xi}$ and $\omega(V^{\xi})$ we incur no concurrent writes. This straightforward parallel algorithm in sufficiently large graphs can produce highly scalable results. Nevertheless, it is important to mention that, clearly in initial steps of uncoarsening where number of vertices in finer graph is still small, work space for threads can be negligible, as the number of vertices in latter steps increase, parallelism is visibly more effective.

After projecting back coarse vertices of parts into finer vertices, we apply refinement by repartitioning the graph using LDG algorithm. Since all neighbors of vertices are assigned to parts, random assignment is eliminated from the algorithm since there is no vertex with zero affinity score. All the threads compute affinity scores for all the neighbors of vertex u in graph $\mathcal{G}$. Absence of random assignment in this section increases the work space of threads and consequently provide a very good scalibility for repartitioning.

Chapter 5

Experimental results

In this chapter we discuss effectiveness of our paradigm by providing variety of results on different graph datasets. In our experiments we consider exploring several topics. After explaining our datasets, we discuss our parameter β and its effect on performance and quality given different values. Then we argue the essence of multilevel paradigm on streaming graph partitioning algorithms. Next, we compare our scheme to two approaches. The first compared framework is LDG algorithm in several iterations of repartitioning. Second approach is well-known multi-threaded multilevel scheme, *mt-metis*. We compare our results to these methods in terms of quality of partitions and performance.

We implemented our framework in C and compiled in gcc with the -O3 flag. In our parallel implementation we exploited OpenMP multi-threading library. All of the results reported are the average of five runs. Our imbalance ratio is set to $\epsilon = 0.10$. We have tested our paradigm and compared it on 24 graphs. It is important to note that in all the experiments that we have reported, after each level of partitioning single pass of repartitioning is applied. At each level more repartitioning passes can be done leading to more refinement. In evaluating quality of partitions two metrics are examined. First, the number of edges being cut between parts and second, imbalance ratio for each part, which is defined below:

$$EC = \frac{cutsize(\Pi)}{|\mathcal{E}|} * 100 \tag{5.1}$$

$$LI = \frac{\mathcal{W}_{max}}{\mathcal{W}_{avg}} \tag{5.2}$$

EC calculates fraction of edges cut in a graph and LI computes how much the maximum loaded part diverse from average part weight where $\mathcal{W}_{max}$ denotes the weight of maximally loaded part and $\mathcal{W}_{avg}$ denotes average part weight under perfect balance. i.e.,

$$\mathcal{W}_{avg} = \frac{\sum\limits_{v \in \mathcal{V}} \omega(v)}{K} \tag{5.3}$$

In the following section we explore details of datasets that we have used in this study.

5.1 Datasets

We perform experiment on a number of real graph datasets. Type of these real world graphs are from different domains. Social graphs, Web graphs, finite element meshes (FEMs), collaboration graphs and similarty is among our chosen real-world graph categories. Most of these graphs are collected from SNAP[35] archive. Table 5.1 shows the basic information about each graph such as number of vertices, number of edges and type. We have also tested our method in synthetic graphs of Watts-Strogatz [31]. For creating these synthetic graphs we followed the parameters mentioned in [9] and used NetworkX package in python with a rewiring parameter of 0.1 for any number of nodes and edges.

Graph name	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	**category**
ljournal-2008	5363260	49514271	social-network
soc-LiveJournal1	4847571	42851237	social-network
hollywood-2009	1139905	56375711	social-network
Web-notredome	325729	1090108	web
Web-google	916428	4322051	web
Eu-2005	862664	16138468	web
copapersDBLP	540486	15245729	citation
coAuthorsDBLP	299067	977676	citation
dblp-2010	326186	807700	citation
cit-Patents	3774768	16518947	citation
coPapersCiteseer	434102	16036720	citation
Patents	3774768	14970766	citation
hcircuit	105676	203734	circuit
circuit5M	5558326	26983926	circuit
Fullchip	2987012	11817567	circuit
amazon-2008	735323	3523472	similarity
Bump_2911	2911419	62409240	FEM
HV15R	2017169	162357569	FEM
ML_Laplace	377002	13656485	FEM
Flan_1565	1564794	57920625	FEM
Dubcova1	16129	118440	FEM
WS	1000000	10000000	synthetic
WS	5000000	55000000	synthetic
WS	10000000	110000000	synthetic

Table 5.1: table of datasets consisting of graphs from social network, web, citation, circuit, similarity, FEM and syntethic categories. In syntethic graph 3 instances of Watts-Strogatz [31] graph in different number of vertices, $\lvert\mathcal{V}\rvert$ and edges $\lvert\mathcal{E}\rvert$ is given

5.2 Parameter β and its effect on partitioning paradigm

Before representing our results, it is important to briefly discuss our chosen parameter β and its effect on the quality and performance of our algorithm. As we discussed in Chapter 4, β, which determines the average number of vertices in each part, can lead to partition with different qualities in the presence of several number of threads hence for a more stable results between each thread, we need to choose proper number for β in parallel platform. On the other hand, without considering the effect of this parameter on parallelism, β also have effect on overall quality of partitioning and runtimes. The reason is that, this parameter determines number of levels of coarsening, very low value for this parameter can produce high number of levels leading to more improvements in qualities because of additional steps for partitioning and repartitioning. Obviously, more levels lead to increase in the performance cost as number of coarsening and partitioning phases increase. Therefore, choosing very high value for parameter β reduce number of levels but at the same time it reduces the quality of partitions. This tradeoff between quality and runtime is demonstrated in Figure 5.1 for three values of $\beta = 10$, $\beta = 20$ and $\beta = 40$.

Values less than 10 can give very volatile results in multi-threaded implementation and values above 40 can totally ignore multilevel scheme by reducing number of levels to as low as 1, which can perform the same as LDG in one pass. Thus we have chosen this range of values for our experiments. As shown in figure 5.1, runtime of each graph reduce by increasing value of β however, at the same time percentage of edgecut increases, leading to poor quality. Therefore we set a value for this parameter such that it can provide a solution which can reduce the tradeoff between quality and performance. Even though using empirical tests we can reach desired value for β for each graph, in our current experiments for the sake of simplicity we set this parameter to a fix value of 20.

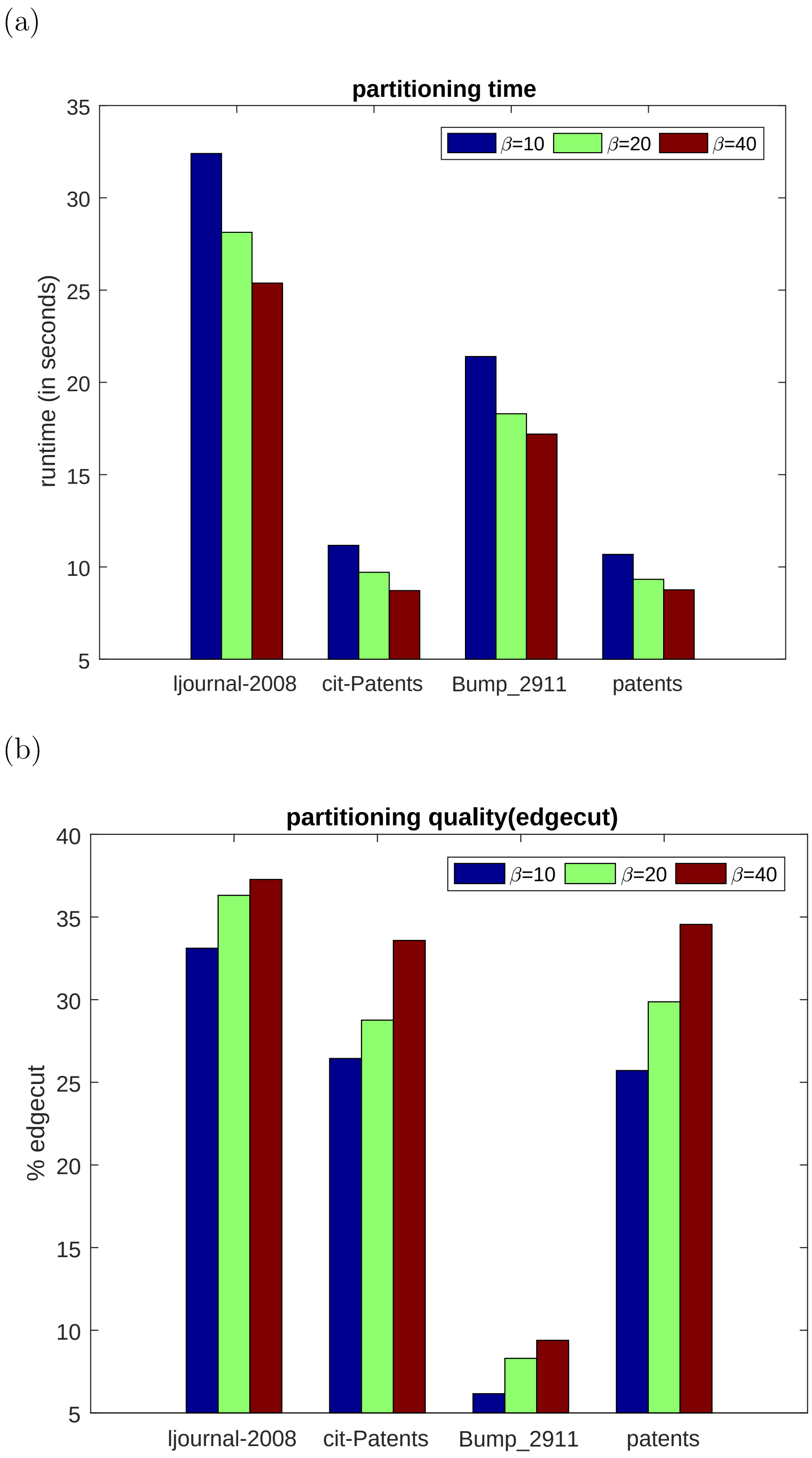

Figure 5.1: Effect of different values of β on (a) runtimes and (b) edgecuts on 4 graph datasets

5.3 Discussion: viability of multilevel paradigm

In this section we aim on showing viability of a multilevel method within the context of streaming algorithm. Even though repartitioning graphs using LDG algorithm in several number of iterations can boost quality of partitions in first few passes, due to its structure, it stuck into a local minimum and produce nearly same edgecuts in the following repartitioning phases. Considering this fact, we aim at showing that our multilevel paradigm can improve quality of parts beyond the improvements in LDG algorithm when given the same running time as our method.

In Figure 5.2 we report edgecut values for each iteration of LDG algorithm in 40 iterations for 4 graphs. As can be seen, in at most first 10 iterations LDG algorithm is able to improve quality of partitions while in the rest of the iterations edgecut values remain almost the same. We have also specified the edgecut value attained by each graph of our method, mt-SML. The iteration number where our algorithm has same runtime as LDG algorithm is also determined in this figure. Interestingly our method on average accounts for 10 iterations of LDG algorithm however, edgecut produced by our method is considerably lower than that of LDG algorithm in that specific iteration number. For instance, in graph `HV15R` our algorithm has run time equal to 8 iterations of LDG algorithm, however, we can achieve better quality than LDG algorithm in 8 iterations.

In a broad comparison it is interpretative that multilevel paradigm on streaming graph partitioning algorithms is a viable solution for improvements in quality of partitions and can usually generate results better than a flat repartitioning approach. In the next evaluations we set the iteration number of LDG algorithm to 10 iterations as it accounts for approximately same runtime as our method and further improvements in edgecuts in more number of iterations is not feasible.

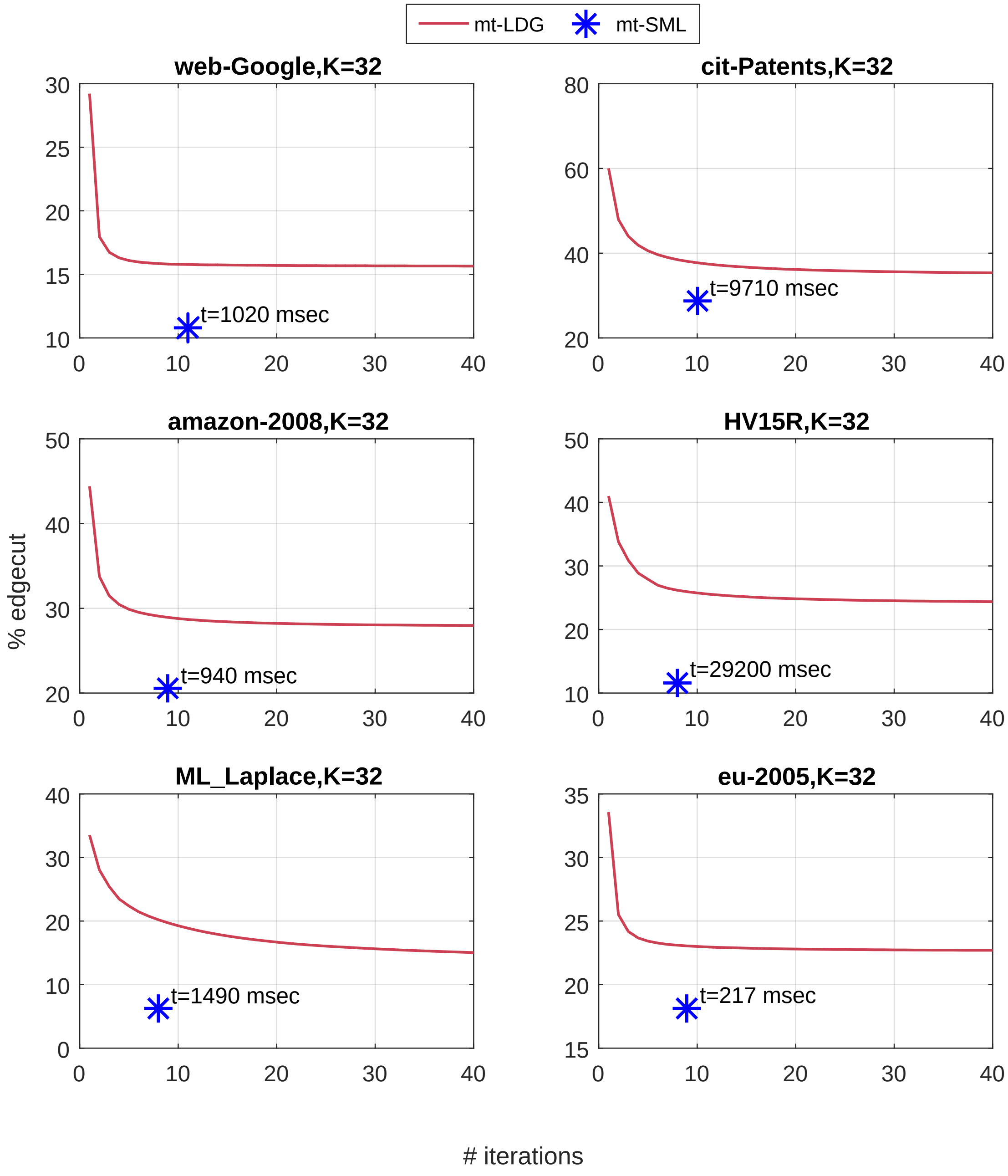

Figure 5.2: Variation of the partitioning quality of mt-LDG with increasing number of repartitioning passes. * shows the edgecut value of the proposed mt-SML

5.4 Quality comparison

In this section we compare our method, mt-SML against *mt-metis* and LDG algorithm in terms of edgecut and imbalance ratio. As LDG algorithm cannot achieve further improvements in terms of quality through performing large number of repartitioning, we choose this algorithm in 10 iteration as a lower bound in our comparisons while *mt-metis* which is a successful multi-threaded multilevel scheme is chosen as our upper bound as it is among fast state-of-the-art practical graph partitioners and can provide good quality parts. Our goal is to generate partitions close to the quality of *mt-metis* while using a lightweight algorithm, LDG.

Figure 5.3 compares percentage of edgecuts, EC of 6 datasets for LDG algorithm in 10 iterations, *mt-metis* and our framework, mt-SML as a function of K. Our method has a steady performance in between these upper and lower bounds. In syntethic graph `WS (10m)` mt-SML performs nearly the same as *mt-metis* for all values of K. In three irregular graphs, `amazon-2008`, `web-google` and `coPapersCiteseer` our performance in each K values are nearly in between upper and lower bounds while in regular graphs `HV15R` and `ML_Laplace`, mt-SML performs closely to the *mt-metis*.

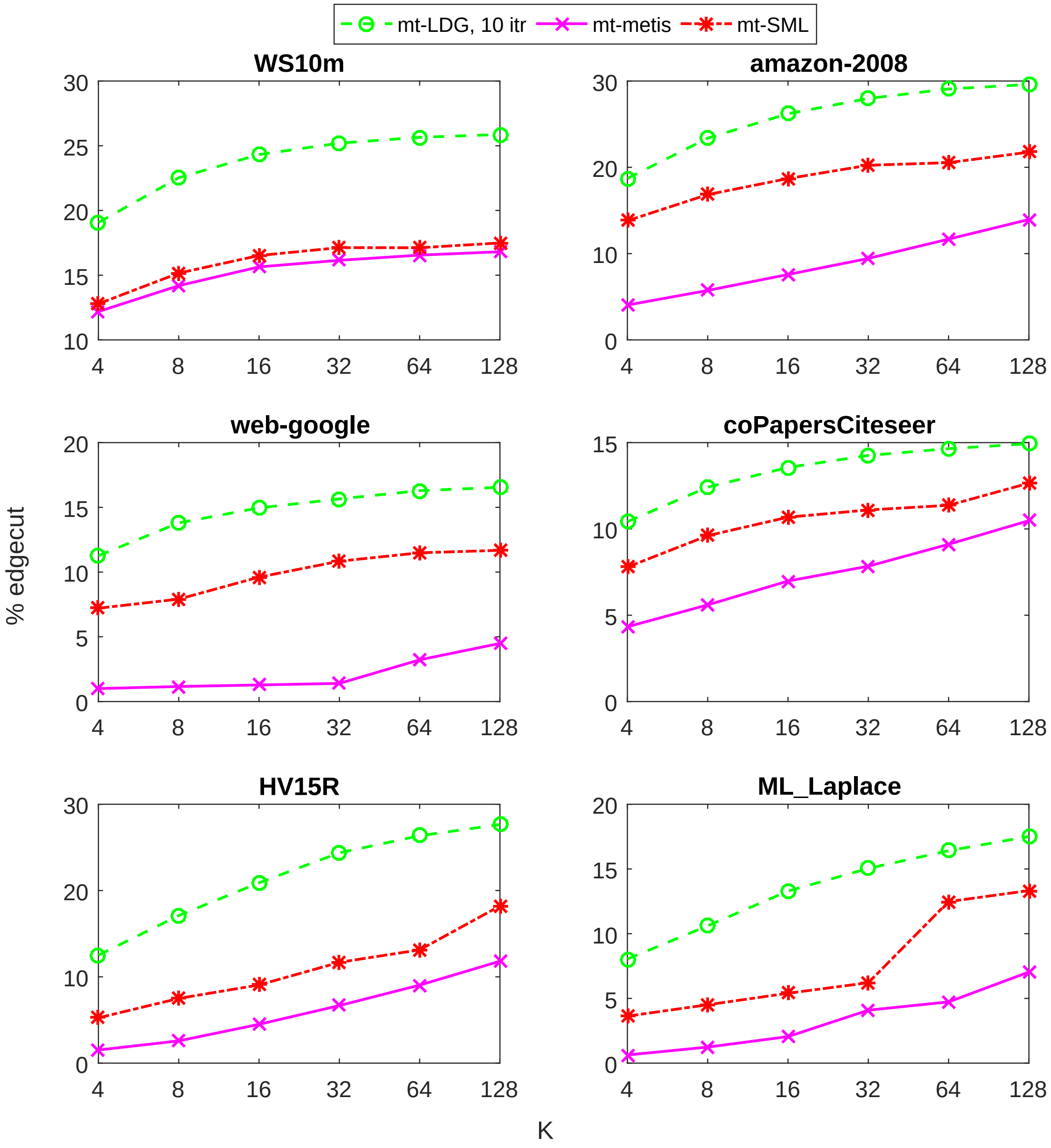

Figure 5.3: Edgecut quality comparison for three methods on different graphs as a function of K

Performance comparison of our method in terms of imbalance ratio, LI is also given in Table 5.2 for each graph in $K = 64$. In all experiments we set capacity constraint by setting $\epsilon = 0.1$ for all methods. Due to the penalty function in LDG algorithm, having an exact balance is expectable thus for each graph dataset it produces steadily balanced partitions. Our method, mt-SML, on the other hand, due to its multilevel structure and the fact that we further relax capacity constraint ϵ in coarsening phase, produces slightly imbalanced partitions however, compared to *mt-metis* our partitions achieve better balance. In the following sections we will provide a broad comparison on imbalance ratio and edgecut qualities on all graph datasets.

graph name	**mt-SML**	**mt-LDG**	***mt-metis***
amazon-2008	1.05	1.01	1.10
web-Google	1.07	1.00	1.10
hcircuit	1.00	1.00	1.10
cit-Patents	1.06	1.00	1.05
coPapersCiteseer	1.02	1.03	1.10
HV15R	1.08	1.01	1.08
WS (10m)	1.00	1.00	1.08
ML_Laplace	1.00	1.00	1.10
patents	1.00	1.01	1.07

Table 5.2: imbalance ratio *LI* for each graph in K=64.

5.5 Scalibility and runtime analysis

Our parallel design is built on several phases and each of these phases are parallelized. Before demonstrating our runtime and performance analysis, we briefly argue the scalibility of each phase of our method in the following section. Then we compare our runtime results with LDG and *mt-metis*.

5.5.1 Scalibility of each phase

In our implementation, in each phase of multilevel partitioning (*partitioning, coarsening, uncoarsening and repartitioning*) threads are created at the beginning of the parallel section and killed at the end. Hence no synchronization is needed among each phase. We calculate runtime for each phase independently. Despite the fact that our algorithms for each phase are highly amenable for parallelism, some phases are less scalable than others. Figure 5.4 shows runtimes

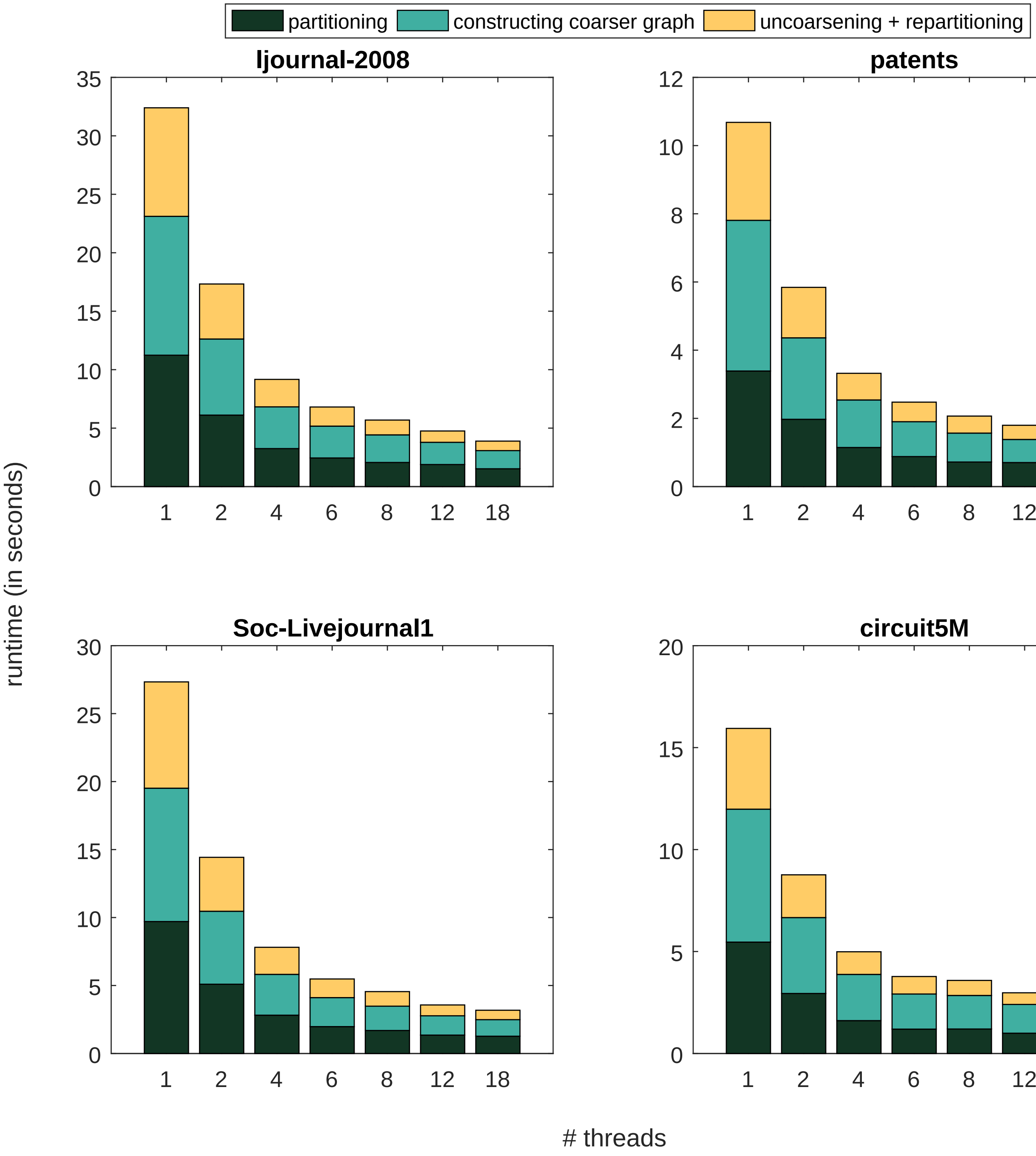

Figure 5.4: Dissection of runtime of mt-SML for 32- way partitioning with $\beta = 10$

for each phase of our partitioning in several number of threads for *social-network* graphs `ljournal-2008` and `soc-Livejournal1` a *citation* graph `patents` and a *circuit* graph `circuit5M`. These irregular graphs are large enough and can provide logically enough workspace for each thread. It is important to note that due to very small runtime results for uncoarsening phase, we have reported runtime results for repartitioning and uncoarsening as a unit.

Figure 5.4 shows that partitioning phase has the *least* scalability while *repartitioning* provide best scalable results in three phases. In partitioning graph in each level from scratch, random assignment can be bottleneck for the parallel performance as, number of parts in initial levels can be very big leading to more time consumption in random assignments for each thread. Since, random assignment is eliminated in repartitioning, this obstacle is removed consequently we have better scalibility in this phase. In coarsening phase, even though assigning vertices into each part from part vector is an atomic operation, scalibility is not limited to this sequential assignment and we can see that coarsening can provide acceptably scalable results in 18 threads.

5.5.2 Runtime and speedup comparison

We have compared our multi-threaded implementation, mt-SML, with a publicly availabale multi-threaded graph partitioning tool, mt-metis, and multi-threaded implementation of LDG algorithm on multiple passes. We have partitioned 13 graphs on 32 parts and we set $\beta = 20$. As we discussed in previous sections, LDG algorithm improve quality of partitions nearly in 10 iterations and after that, it has a nearly steady performance in terms of quality of partitions. Hence in our runtime analysis we compute runtimes for 10 passes of LDG algorithm and compare it to our method. Table 5.3 shows runtime results for these three methods.

Graph	2 Threads			4 Threads			8 Threads			18 Threads		
	LDG	*metis*	SML	LDG	*metis*	SML	LDG	*metis*	SML	LDG	*metis*	SML
amazon-2008	0.60	1.75	**0.52**	0.39	1.12	**0.32**	0.30	0.70	**0.23**	0.26	0.60	**0.18**
web-Google	0.56	1.52	**0.62**	0.36	0.92	**0.34**	0.24	0.62	**0.25**	0.20	0.49	**0.19**
eu-2005	1.34	2.45	**1.43**	0.78	1.59	**0.79**	0.47	1.02	**0.47**	0.34	0.82	**0.32**
cit-Patents	5.61	19.5	**5.26**	3.20	12.36	**2.91**	2.14	7.26	**1.8**	1.65	5.62	**1.27**
coPapersCiteseer	1.12	1.51	**0.93**	0.63	1.1	**0.5**	0.36	0.72	**0.3**	0.20	0.6	**0.18**
patents	5.27	18.97	**4.96**	3.02	11.65	**2.79**	2.07	6.65	**1.76**	1.64	5.08	**1.26**
ML_Laplace	0.96	5.76	**0.96**	0.54	4.00	**0.54**	0.47	2.59	**0.34**	0.35	2.0	**0.21**
HV15R	16.73	14.95	**16.10**	8.4	9.2	**8.20**	4.35	6.32	**4.31**	2.14	4.44	**2.22**
ljournal-2008	14.35	61.28	**14.66**	7.34	36.56	**7.75**	5.16	21.25	**4.33**	2.49	17.83	**2.9**
circuit5M	6.2	120.99	**7.35**	3.6	72.7	**4.9**	2.16	47.94	**2.62**	1.59	42.47	**1.95**
WS(1m)	1.28	4.15	**1.15**	0.79	2.33	**0.68**	0.6	1.39	**0.44**	0.51	1.19	**0.34**
WS(5m)	15.07	30.90	**12.52**	7.75	17.12	**7.17**	4.34	9.97	**3.72**	2.86	8.37	**2.42**
WS(10m)	40.53	71.23	**32.84**	20.85	39.62	**18.29**	11.46	23.42	**10.50**	6.64	20.05	**6.49**

Table 5.3: Runtimes of multi-threaded methods, mt-LDG, mt-SML and *mt-metis* in seconds for different number of threads.

In nearly all graph partitioning instances that we have tested, mt-SML runs faster and produce better speedup compared to *mt-metis*. Only in FEM graph, `HV15R`, we have higher runtime than *mt-metis* when implemented with two threads, however our algorithm produces approximately **two** times faster results in 18 threads which accounts for better speedup in our algorithm. In graph `ML_Laplace`, mt-SML saw its largest lead in performance compared to *mt-metis* in 18 threads. Our scheme performs nearly ten times faster than *mt-metis* in same number of threads. In comparing runtime results of our method to LDG algorithm we can see approximately same performances where in some graphs LDG achieve slightly faster results and in others we achieve better results. As the number of iterations where LDG algorithm can have nearly same runtime as our method differes in each dataset, this diversity can be justified. For instance, in graph dataset `coPapersCiteseer` our method have nearly the same runtime as LDG algorithm in 8 iterations therefore 10 iterations of LDG algorithm can take more time. Figure 5.5 demonstrates the runtime results with respect to number of threads for three methods on different graphs as a log scale. In this figure we can clearly observe that mt-SML has the most scalable and fast results.

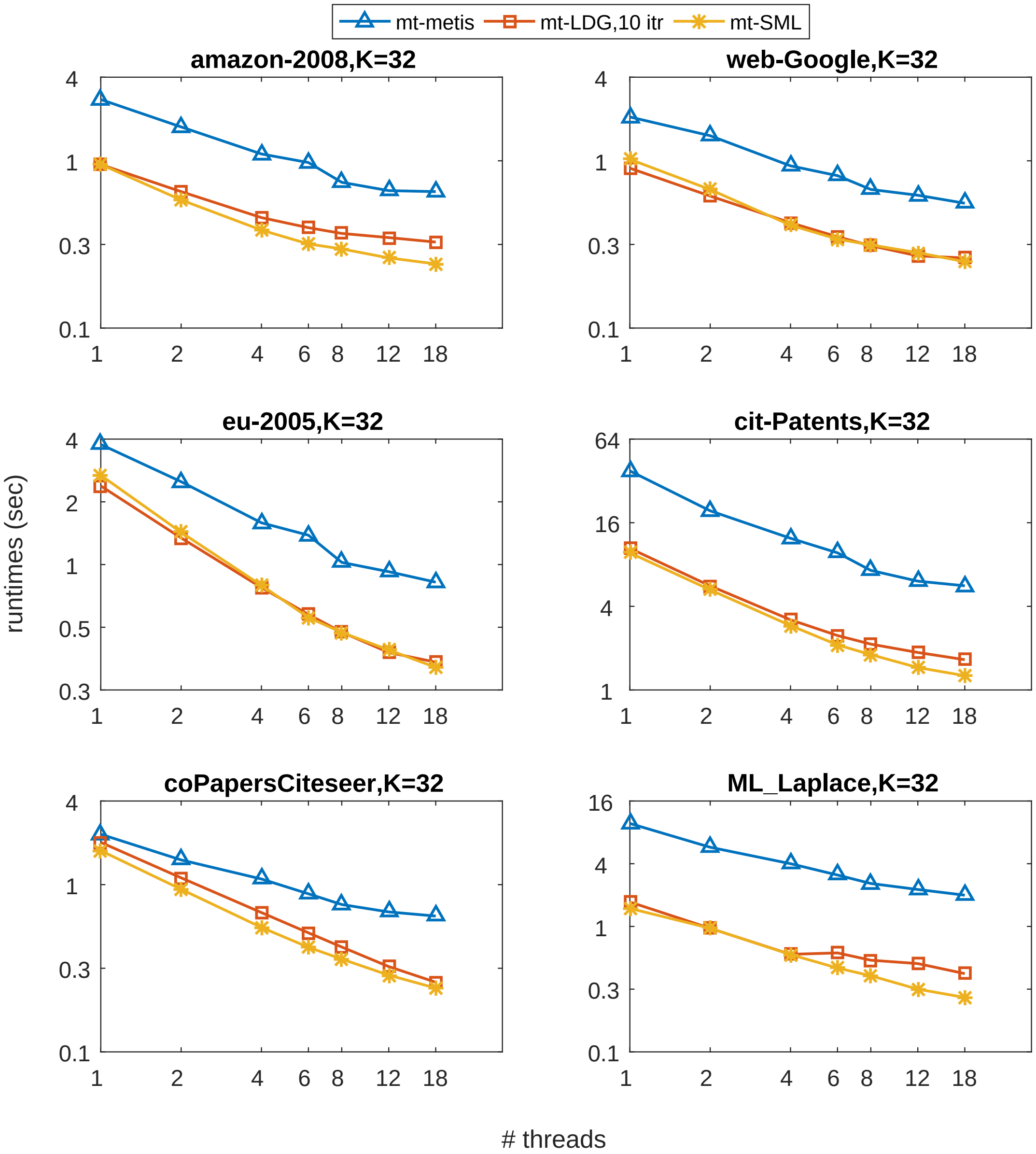

Figure 5.5: Variation of the running time of mt-LDG, *mt-metis* and mt-SML with increasing number of threads for $\beta = 20$.

Figure 5.6 demonstrate comparison of three methods in terms of speedup for 4 graphs. As can be seen, mt-SML overall runs faster and achieve better speedup than *mt-metis*, in all 4 datasets. In graph `HV15R` our method perform close to the ideal speedup line. In graph `WS(10m)` speedup value for mt-SML, *mt-metis* and LDG algorithm in 7 iterations for 18 threads is 9.37, 6.46 and 11.44 respectively. Even though *mt-metis* has speedup close to mt-SML, our algorithm can achieve 3.5 times faster results in 18 threads.

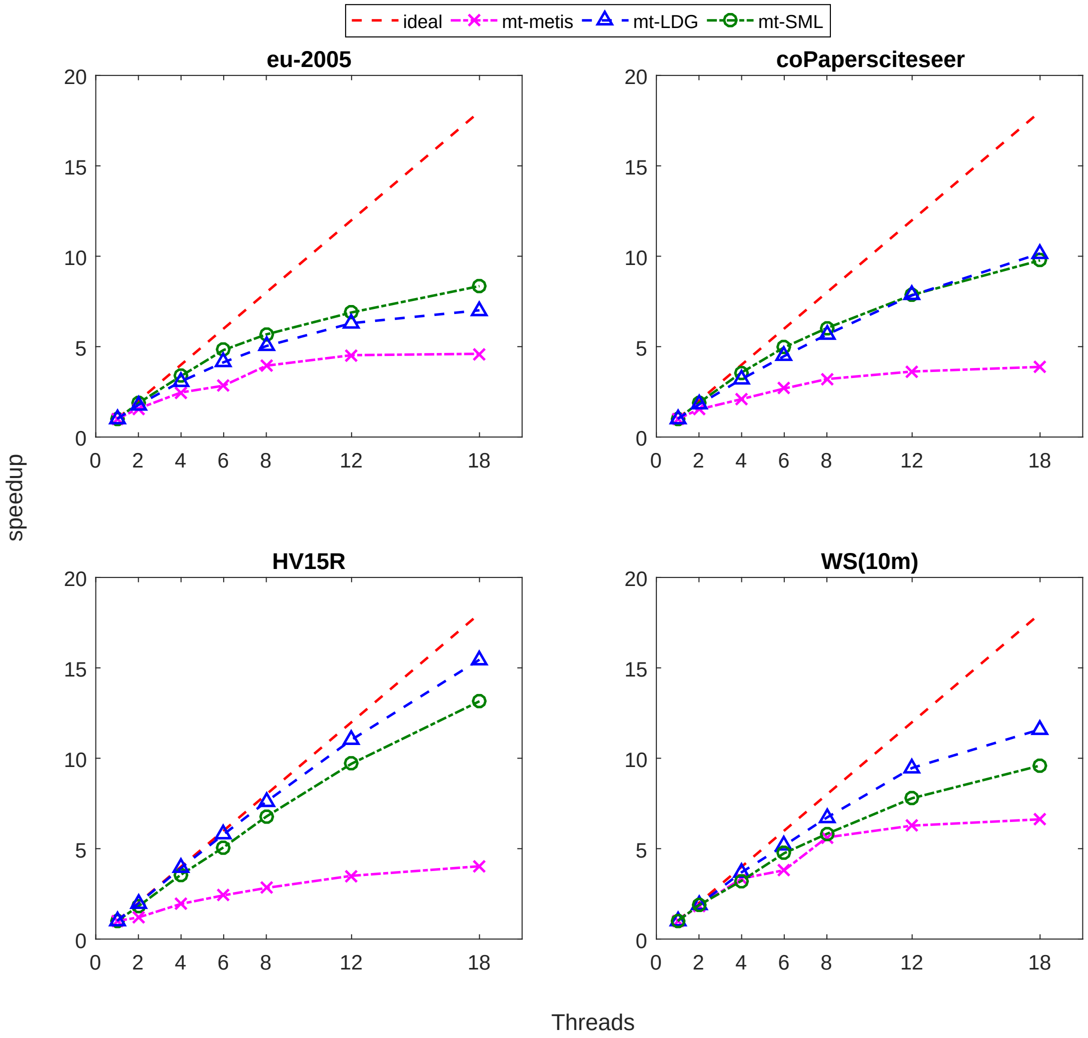

Figure 5.6: Comparison of mt-SML, *mt-metis* and mt-LDG in case of speedup on 4 different graph datasets, the ideal speedup is also shown as dashed red line

5.6 Experimental evaluation on all graphs

In this section we evaluate our method on all graphs for three different values of $\beta = \{10, 20, 40\}$. In table 5.4 we provide scaled geometric mean of all graphs with respect to LDG algorithm in 10 iterations for the methods that we have discussed in this thesis both in terms of quality and runtime for $K = \{4, 32, 64\}$ and all thread numbers $\{1, 2, 4, 6, 8, 12, 18\}$. The Table 5.4 displays the performance results of *mt-metis* and mt-SML normalized with respect to those of mt-LDG. Normalized geomtric means are displayed within different graph category. Overall, *mt-metis* has worse imbalance ratio and runtimes than LDG while its edgecut quality is better. This is expectable as *mt-metis* utilizes more sophisticated algorithms in its refinement phase and as a result it has better improvements with the cost of increased runtime and imbalance ratio. Our method, mt-SML, on the other hand provides different results depending on the β values for each metric. In edgecut comparison even though overall mt-SML attain better qualities in all values of β compared to LDG, degree to which our algorithm can provide better partitions has reverse relationship with β i.e. with increasing β edgecut quality reduces. Also the tradeoff between edgecut quality and runtimes is visibly clear in this table. As we increase β edgecut quality reduce compared to LDG while runtime results improve, such that we achieve even faster runs than LDG algorithm in 10 iterations. Our imbalance ratio does not diverge from results obtained for LDG algorithm.

We also provide averages for each graph types in this table which we specify them with the category they belong. In FEM graphs mt-SML has better *EC* quality compared to LDG algorithm. In syntethic graphs our edgecut quality is mostly close to quality of the *mt-metis*. In social networks however, impact of multilevel paradigm on streaming algorithms is less visible as our edgecut results are nearly the same as LDG. As social networks are usually power law graphs with few high degree vertices and many low degree vertices, this result is interpretative due to the strict penalty function in the nature of LDG algorithm which usually prevents assigning most of the neighbors of high degree vertices to be in same parts. Nevertheless, *mt-metis* also has the least improvement compared to LDG in this type of networks.

type	**metrics**	**mt-LDG**	***mt-metis***	**mt-SML**		
				β=10	β=20	β=40
general	*EC*	1.00	0.32	0.71	0.75	0.77
	LI	1.00	1.08	1.02	1.01	1.01
	runtime	1.00	2.56	1.12	0.95	0.87
social	*EC*	1.00	0.76	0.92	1.00	1.03
	LI	1.00	1.09	1.02	1.01	1.02
	runtime	1.00	4.65	1.26	1.01	0.97
web	*EC*	1.00	0.18	0.73	0.73	0.74
	LI	1.00	1.09	1.02	1.01	1.00
	runtime	1.00	2.53	1.26	1.09	0.98
citation	*EC*	1.00	0.46	0.73	0.81	0.81
	LI	1.00	1.05	1.02	1.01	1.00
	runtime	1.00	2.94	1.08	0.93	0.85
circuit	*EC*	1.00	0.20	0.82	0.89	0.97
	LI	1.00	1.15	1.02	1.01	1.01
	runtime	1.00	3.78	1.12	0.97	0.91
similarity	*EC*	1.00	0.30	0.67	0.67	0.72
	LI	1.00	1.08	1.04	1.02	1.00
	runtime	1.00	2.68	0.97	0.84	0.76
mesh	*EC*	1.00	0.15	0.49	0.53	0.58
	LI	1.00	1.04	1.02	1.01	1.01
	runtime	1.00	1.48	1.06	0.89	0.81
syntethic	*EC*	1.00	0.66	0.82	0.77	0.72
	LI	1.00	1.12	1.07	1.04	1.05
	runtime	1.00	1.81	1.09	0.90	0.78

Table 5.4: Extensive comparison based on runtime, imbalance ratio and edgecut values over all graphs for three schemes scaled relative to mt-LDG in 10 iterations

Chapter 6

Conclusion and future work

In this study we proposed a parallel multilevel streaming graph partitioning approach. The main contribution of our framework is to utilize lightweight streaming graph partitioning heuristic in a multilevel framework. Streaming graph partitioning solutions are proposed for one pass graph partitioning in a stream order and repartitioning using these algorithms have little effect on improvements in quality of partitions. Our proposed method embed streaming heuristic in a multilevel approach aiming at boosting quality of partitions. Moreover, we proposed multi-threaded implementation of our graph partitioning framework targeting at increasing performance both on partition quality and runtimes.

We experimentally tested our method, mt-SML, on more than 20 graphs. First we show that a greedy streaming graph partitioning algorithm cannot improve partition qualities after a few passes of repartitioning, then we show that our multilevel framework achieves much better qualities than streaming graph partitioning algorithm as given the same amount of running time. Moreover, we compare performance of our method against the state-of-the-art multi-threaded multilevel graph partitioning tool. Our results clearly indicate that our framework can achieve faster results which are also more scalable to the number of threads used for parallel implementation.

As a future work, we first propose testing several streaming graph partitioning heuristics besides the one used in our work. This can be a good empirical examination on the effects of these heuristics on different graph structures. The second approach can be modifying stream order for a more structure aware order. After partitioning graph using streaming graph partitioning algorithm, we can store graph information such as degree of each vertex to modify the order of vertices in the next iteration. In this case we may have a chance to assign vertices in parts with most number of neighbors before getting penalized by full parts.

Bibliography

[1] Garey, Michael R., David S. Johnson, and Larry Stockmeyer. *"Some simplified NP-complete graph problems."* Theoretical computer science 1.3 (1976): 237-267.

[2] Bui, Thang Nguyen, and Curt Jones. *"Finding good approximate vertex and edge partitions is NP-hard."* Information Processing Letters 42.3 (1992): 153-159.

[3] Catalyurek, Umit V., and Cevdet Aykanat. "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication." IEEE Transactions on parallel and distributed systems 10.7 (1999): 673-693.

[4] Karypis, George, and Vipin Kumar. *"A fast and high quality multilevel scheme for partitioning irregular graphs."* SIAM Journal on scientific Computing 20.1 (1998): 359-392.

[5] Pellegrini, François, and Jean Roman. "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs." International Conference on High-Performance Computing and Networking. Springer, Berlin, Heidelberg, 1996.

[6] LaSalle, Dominique, and George Karypis. *"Multi-threaded graph partitioning."* Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013.

[7] Akhremtsev, Yaroslav, Peter Sanders, and Christian Schulz. "High-Quality Shared-Memory Graph Partitioning." arXiv preprint arXiv:1710.08231 (2017).

[8] Tsourakakis, Charalampos, et al. *"Fennel: Streaming graph partitioning for massive scale graphs."* Proceedings of the 7th ACM international conference on Web search and data mining. ACM, 2014.

[9] Stanton, Isabelle, and Gabriel Kliot. *"Streaming graph partitioning for large distributed graphs."* Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2012.

[10] Nishimura, Joel, and Johan Ugander. *"Restreaming graph partitioning: simple versatile algorithms for advanced balancing."* Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2013.

[11] Firth, Hugo, and Paolo Missier. "Workload-aware Streaming Graph Partitioning." EDBT/ICDT Workshops. 2016.

[12] Battaglino, Casey, Pienta Pienta, and Richard Vuduc. "Grasp: distributed streaming graph partitioning." 1st High Performance Graph Mining workshop, Sydney, 10 August 2015. 2015.

[13] Tsourakakis, Charalampos. "Streaming graph partitioning in the planted partition model." Proceedings of the 2015 ACM on Conference on Online Social Networks. ACM, 2015.

[14] Petroni, Fabio, et al. "Hdrf: Stream-based partitioning for power-law graphs." Proceedings of the 24th ACM International on Conference on Information and Knowledge Management. ACM, 2015.

[15] Stanton, Isabelle. "Streaming balanced graph partitioning algorithms for random graphs." Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2014.

[16] Abou-Rjeili, Amine, and George Karypis. "Multilevel algorithms for partitioning power-law graphs." Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. IEEE, 2006.

[17] Huang, Jiewen, and Daniel J. Abadi. ”Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs.” Proceedings of the VLDB Endowment 9.7 (2016): 540-551.

[18] Stanton, Isabelle. ”Streaming balanced graph partitioning algorithms for random graphs.” Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2014.

[19] Huang, Jiewen, and Daniel J. Abadi. ”Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs.” Proceedings of the VLDB Endowment 9.7 (2016): 540-551.

[20] Schloegel, Kirk, George Karypis, and Vipin Kumar. ”Parallel static and dynamic multi-constraint graph partitioning.” Concurrency and Computation: Practice and Experience 14.3 (2002): 219-240.

[21] Walshaw, Chris, and Mark Cross. ”Mesh partitioning: a multilevel balancing and refinement algorithm.” SIAM Journal on Scientific Computing 22.1 (2000): 63-80.

[22] Diekmann, Ralf, et al. ”Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM.” Parallel Computing 26.12 (2000): 1555-1581.

[23] Pellegrini, François, and Jean Roman. ”Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs.” International Conference on High-Performance Computing and Networking. Springer, Berlin, Heidelberg, 1996.

[24] Raghavan, Usha Nandini, Réka Albert, and Soundar Kumara. ”Near linear time algorithm to detect community structures in large-scale networks.” Physical review E 76.3 (2007): 036106.

[25] Kernighan, Brian W., and Shen Lin. ”An efficient heuristic procedure for partitioning graphs.” The Bell system technical journal 49.2 (1970): 291-307.

[26] Walshaw, Chris, and Mark Cross. ”JOSTLE: parallel multilevel graph-partitioning software–an overview.” Mesh partitioning techniques and domain decomposition techniques (2007): 27-58.

[27] Sanders, Peter, and Christian Schulz. ”Engineering multilevel graph partitioning algorithms.” European Symposium on Algorithms. Springer, Berlin, Heidelberg, 2011.

[28] Karypis, George, and Vipin Kumar. ”Multilevel algorithms for multi-constraint graph partitioning.” Supercomputing, 1998. SC98. IEEE/ACM Conference on. IEEE, 1998.

[29] Aykanat, Cevdet, B. Barla Cambazoglu, and Bora Uçar. ”Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices.” Journal of Parallel and Distributed Computing 68.5 (2008): 609-625.

[30] Schloegel, Kirk, George Karypis, and Vipin Kumar. ”Parallel multilevel algorithms for multi-constraint graph partitioning.” European Conference on Parallel Processing. Springer, Berlin, Heidelberg, 2000.

[31] Watts, Duncan J., and Steven H. Strogatz. ”Collective dynamics of ‘small-world’networks.” nature 393.6684 (1998): 440.

[32] Kwak, Haewoon, et al. ”What is Twitter, a social network or a news media?.” Proceedings of the 19th international conference on World wide web. AcM, 2010.

[33] Hendrickson, Bruce, and Robert Leland. ”A multi-level algorithm for partitioning graphs.” (1995): 28.

[34] Chevalier, Cédric, and François Pellegrini. ”PT-Scotch: A tool for efficient parallel graph ordering.” Parallel computing 34.6-8 (2008): 318-331.

[35] http://snap.stanford.edu/snap/