Journal of Parallel and Distributed Computing 147 (2021) 140–151

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Fast shared-memory streaming multilevel graph partitioning

Nazanin Jafari [a], Oguz Selvitopi [b,*], Cevdet Aykanat [c]

[a] College of Information and Computer Science, UMass Amherst, Amherst, MA 01002, United States of America
[b] Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, United States of America
[c] Department of Computer Engineering, Bilkent University, Ankara, 06800, Turkey

ARTICLE INFO

Article history:
Received 15 August 2019
Received in revised form 19 June 2020
Accepted 1 September 2020
Available online 12 September 2020

Keywords:
Streaming algorithms
Graph partitioning
Multilevel graph partitioning
Parallel graph partitioning

ABSTRACT

A fast parallel graph partitioner can benefit many applications by reducing data transfers. The online methods for partitioning graphs have to be fast and they often rely on simple one-pass streaming algorithms, while the offline methods for partitioning graphs contain more involved algorithms and the most successful methods in this category belong to the multilevel approaches. In this work, we assess the feasibility of using streaming graph partitioning algorithms within the multilevel framework. Our end goal is to come up with a fast parallel offline multilevel partitioner that can produce competitive cutsize quality. We rely on a simple but fast and flexible streaming algorithm throughout the entire multilevel framework. This streaming algorithm serves multiple purposes in the partitioning process: a clustering algorithm in the coarsening, an effective algorithm for the initial partitioning, and a fast refinement algorithm in the uncoarsening. Its simple nature also lends itself easily for parallelization. The experiments on various graphs show that our approach is on the average up to 5.1x faster than the multi-threaded MeTiS, which comes at the expense of only 2x worse cutsize.

Published by Elsevier Inc.

1. Introduction

Graphs are ubiquitous in many diverse fields of science and engineering. They form the basis of many important algorithms and data structures in computer science. Graphs in computational biology help scientists in understanding functions of proteins or assembling long genome sequences from shorter strings called reads. In VLSI circuit design, graphs are successfully used to minimize the propagation delay or the wire crossings in a layout. Graphs are powerful mediums in addressing important questions from almost any field.

Today, large graphs can have billions or even trillions of edges and such a size more than often necessitates the execution of algorithms on graphs in parallel. There are many distributed graph computation frameworks, GraphLab [22], Giraph [2], Pregel [23], to name a few, that facilitate parallel execution of algorithms based on graphs. In parallel processing of the graph, the vertices and edges are distributed among the processors and the processors operate on their portions of the graph and occasionally exchange data. Partitioning of the graph usually has a crucial effect on the parallel performance of the executed algorithm, and if not done carefully it can lead to poor performance. Most distributed graph frameworks rely on a simple hash function to distribute graphs. Although this can lead to somewhat acceptable computational load balance, it is very likely to cause high communication (i.e., high edge-cut or vertex-cut) as this translates into a random partitioning of the graph.

In distributed graph computation frameworks, the distribution of the vertices and edges of the graph must be fast. Although trivial hash functions are certainly acceptable, the lightweight streaming algorithms can offer better quality partitions while still staying within the acceptable preprocessing time limits. There are several works [4,13,24,27,30,35–37] that use streaming algorithms for distributing a graph. These algorithms are usually categorized in the *online* algorithms category for partitioning a graph. The *offline* graph partitioning is more expensive and usually contains more involved methods. Examples include, but certainly not constrained to, spectral methods [3,11,12,17,28], multilevel methods [6,16,18,19], and rather recently, metaheuristics [5,31,34]. Multilevel methods are usually the choice of preference as they are able to produce high-quality partitions very fast and the most widely adopted tools [19,26] rely on this method. Offline methods are usually deemed to be too expensive for distributed graph computation frameworks and they are usually preferred in scientific computing or VLSI circuit design, where the overhead of expensive partitioning can be justified.

This work tries to answer the following question: How feasible is it to utilize a streaming graph algorithm within the context of multilevel graph partitioning framework? Most multilevel graph partitioners make use of several different heuristics in different sections of the multilevel framework. Although doing so certainly

* Corresponding author.
E-mail address: roselvitopi@lbl.gov (O. Selvitopi).

https://doi.org/10.1016/j.jpdc.2020.09.004
0743-7315/Published by Elsevier Inc.

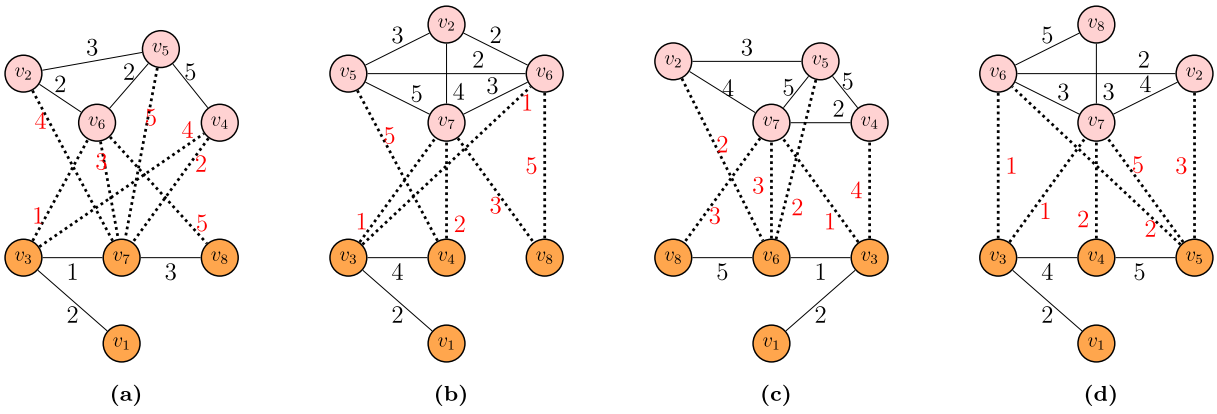


Fig. 1. Four different approaches in obtaining a perfectly balanced two-way partition. The vertices belonging to different partitions are illustrated with different colors. The cut edges are shown with dashed lines and the numbers above edges indicate their weights. (a) Random partitioning (cutsize = 24). (b) The streaming algorithm linear deterministic greedy [35] (cutsize = 17). (c) A single pass of Kernighan–Lin iterative improvement heuristic [20], with the initial starting partition being the random partition in (a) (cutsize = 15). (d) An optimal partition (cutsize = 14).

improves the quality of the solutions, they often sacrifice from the execution time and parallel efficiency. The solutions obtained by a fast streaming algorithm that is amenable for parallelization and utilized within the multilevel framework can be more attractive when its advantages are considered. The hastily made decisions by the streaming algorithm can be corrected by using the same streaming algorithm for refinement purposes. Fig. 1 shows four different methods to partition a graph with eight vertices and fourteen edges into two parts with each part having the same number of vertices. These methods are: (i) random partitioning (Fig. 1a), (ii) a streaming algorithm (Fig. 1b), (iii) an iterative improvement heuristic (Fig. 1c), and (iv) an optimal partition in terms of cutsize (Fig. 1d). The figure shows that a smaller cutsize can be obtained using more complex and expensive algorithms.

We choose to utilize the streaming algorithm linear deterministic greedy (LDG) that is proposed by [35]. Among the many algorithms tested out in [35], the LDG algorithm attained the best performance. The LDG algorithm serves multiple purposes in multilevel framework in our approach. It is used as a clustering algorithm in the coarsening phase and it inherently produces an initial partition of the original graph after the coarsening completes. The LDG algorithm can be considered as an agglomerative clustering algorithm, in which two or more vertices are used to form a new coarser vertex. It has also the nice feature of obtaining coarse vertices uniform in size, which is a feature that is commonly sought for the algorithms used in the coarsening. In the uncoarsening phase, it serves as a refinement algorithm which is used for repartitioning purposes. The flexibility and fast nature of this heuristic renders it attractive to be utilized anywhere in the multilevel framework. The contributions of this work are listed as follows:

1. We assess the feasibility of using a streaming algorithm within the multilevel algorithm.
2. We describe in detail how a flexible streaming algorithm is used for the purposes of coarsening, initial partitioning, and uncoarsening in the multilevel partitioning framework. We outline how to best exploit the algorithm to suit the needs of these different stages.
3. We parallelize the LDG algorithm and the multilevel framework for shared memory architectures. We essentially come up with a fully-fledged shared-memory parallel multilevel graph partitioner that relies on a simple and flexible streaming algorithm to do its bidding.
4. We investigate the feasibility of utilizing multilevel framework within an online context. We describe how our approach can be utilized for online graph partitioning under the assumption that the vertices in the stream arrive in batches.
5. We validate our approach on a comprehensive dataset by comparing it to the widely-adopted partitioner MeTiS [19] and the flat LDG algorithm [35].

The rest of this paper is organized as follows: The notation is introduced in Section 2. Background and the studies related to this work are discussed in Section 3. Our approach is explained in Section 4 and the parallelization of our approach for shared memory systems is discussed in Section 5. We describe how to utilize the proposed methodology for online graph partitioning in Section 6. The proposed methodology is validated in Section 7 with experiments and the conclusions are given in Section 8.

2. Definitions

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a tuple of a vertex set $\mathcal{V}$ of size n and an edge set $\mathcal{E}$ of size m. Each edge $(v_i, v_j) \in \mathcal{E}$ connects two distinct vertices. Each vertex v_i and each edge (v_i, v_j) are associated with weights, which are respectively denoted by $\omega(v_i)$ and $\omega(v_i, v_j)$. The neighbors of v_i are denoted by $Adj(v_i) = \{v_j : (v_i, v_j) \in \mathcal{E}\}$. *Adj* function easily extends to a set of vertices.

$\Pi(\mathcal{G}) = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is said to be a K-way partition of $\mathcal{G}$ if each part is nonempty, the parts are pairwise disjoint and mutually exhaustive. In $\Pi(\mathcal{G})$, an edge (v_i, v_j) is *cut* if the pair of vertices connected by that edge is in different parts, and *uncut* otherwise. The cutsize of $\Pi(\mathcal{G})$ is denoted with $cutsize(\Pi(\mathcal{G}))$ and is equal to the sum of the weights of the cut edges, i.e.,

$$cutsize(\Pi(\mathcal{G})) = \sum_{\substack{(v_i,v_j)\in\mathcal{E} \\ v_i\in\mathcal{V}_k, v_j\in\mathcal{V}_{\ell\neq k}}} \omega(v_i, v_j). \tag{1}$$

The weight of each part is equal to the sum of the weights of the vertices in that part, i.e.,

$$\omega(\mathcal{V}_k) = \sum_{v_i\in\mathcal{V}_k} \omega(v_i). \tag{2}$$

$\Pi(\mathcal{G})$ is said to be balanced if it satisfies the balance constraint for each part $\mathcal{V}_k$:

$$\omega(\mathcal{V}_k) \leq (1+\epsilon)\mathcal{W}_{avg}, \tag{3}$$

where ϵ is the given maximum imbalance ratio and $\mathcal{W}_{avg}$ is the average part weight. Under these definitions, the graph partitioning problem is defined as finding $\Pi(\mathcal{G})$ that minimizes $cutsize(\Pi(\mathcal{G}))$ and satisfies the balance constraint. In the rest of the paper, the indices i and j are used for vertices, while the index k is used for vertex parts.

3. Background and related work

We review two different approaches to graph partitioning problem in this section: (i) the successful multilevel and offline graph partitioning and (ii) the online streaming graph partitioning. We also discuss parallelism in studies that realize these approaches.

3.1. Multilevel graph partitioning

Multilevel partitioning is a successful paradigm widely adopted in several graph/hypergraph partitioners such as Metis [19], Patoh [7], Scotch [26], Jostle [39], Chaco [16], and KaHIP [31]. Parallelizing multilevel algorithms in the context of graph partitioning has been the focus of several studies. Akhremtsev et al. [1] propose a shared memory multilevel graph partitioner by parallelizing the label propagation algorithm [29] in the coarsening phase and introducing a parallel version of k-way multi-try local search [31]. ParMetis [32], as a distributed memory graph partitioner, and *mt-metis* [21], as the shared memory version of Metis are among the commonly used parallel graph partitioners. PT-Scotch [8] is a parallel version of Scotch [26]. These partitioners often suffer from poor scalability because the algorithms used in the initial partitioning and uncoarsening phases of the multilevel framework do not always lend themselves to efficient parallelism.

General structure of a multilevel framework consists of coarsening a given graph into successively coarser graphs, then applying a graph partitioning algorithm on the coarsest graph, and finally projecting it back to the original graph by performing a refinement algorithm on successively finer graphs. Since finer graphs have more degrees of freedom, refinement can effectively increase the quality of the partitions in the uncoarsening [21].

In the *coarsening phase*, the original graph $\mathcal{G} = \mathcal{G}^0 = (\mathcal{V}^0, \mathcal{E}^0)$ is transformed into a sequence of coarser graphs $\mathcal{G}^1, \mathcal{G}^2, \ldots, \mathcal{G}^L$. Different types of algorithms have been tried out for this phase, among which edge matching algorithms [10,19,26,38] and label propagation algorithm [29] are common choices. The coarsest graph $\mathcal{G}^L$ is then partitioned in the *initial partitioning* phase. Various algorithms are used for this phase as well: spectral bisection [28], Kernighan–Lin algorithm [20], graph growing partitioning algorithm [9,14], greedy graph growing partitioning algorithm [19] are among the most common partitioning algorithms. At each level ℓ of the *uncoarsening phase*, a refinement algorithm is applied with the aim of improving the cutsize obtained in the previous uncoarsening level. Multilevel framework successfully stood the test of time and is usually the choice for obtaining good quality solutions. However, it is expensive. In addition, an offline partitioning strategy may not be viable in the context of harsher conditions such as limited memory or incremental updates to the graph which adds/removes vertices or edges to/from the graph. These reasons necessitate the lightweight algorithms for graph partitioning, which we focus next.

3.2. Streaming graph partitioning

In contrast to the offline partitioning methods, the online methods make use of lightweight algorithms that generate "sufficiently good" partitions by keeping only a fraction of all graph information in the memory at any time. These algorithms assign vertices or edges to the parts as they arrive in the stream.

Stanton and Kliot [35] propose 10 heuristics, among which the Linear Deterministic Greedy (LDG) algorithm produces the best results. This heuristic greedily assigns vertices to the parts in which they have the highest number of neighbors while penalizing heavy parts with a linearly weighted function. Its linear penalty function is able to produce well-balanced partitions without neglecting the graph structure. Fennel [37], another streaming graph partitioning algorithm, is a modularity maximization framework that approximately balances the part weights while greedily assigning vertices to the parts. Tsourakakis et al. [36] 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. [27] proposed replicated streaming edge-based graph partitioning mainly focusing on power-law graphs. HoVerCut [30] is a multi-threaded streaming edge-based partitioning platform. A more recent work in this area, ADWISE [24], proposes a smart approach in taking into account a window of edges for partitioning in a stream instead of handling one vertex at a time in random order.

The quality of the partitions obtained by streaming graph partitioning algorithms is usually not comparable to the quality of the partitions obtained by offline partitioners. To partially alleviate this, Nishimura and Ugander [25] propose a multipass solution for two single-pass streaming algorithms, Fennel and LDG. In their work, a graph partitioned with the streaming algorithms can be repartitioned several times. Even though repartitioning can be effective in increasing the quality of the partitions in only few passes, the quality does not further improve in the later passes.

Besides these works, Firth and Missier [13] propose a workload-aware streaming graph partitioning algorithm. They use frequent patterns seen in the graph as workloads and partition the graph considering these motifs with the LDG algorithm. Grasp [4] proposes distributed-memory streaming graph partitioning. For this purpose, it uses MPI to parallelize the framework proposed in Fennel.

As our proposed method is based on the LDG algorithm, here we briefly review it. We utilize the LDG heuristic in both *coarsening* and *uncoarsening* phases of the multilevel partitioning framework. The LDG heuristic computes the affinity of a vertex v in the stream to the parts considering its neighbors and assigns it to the part with the maximum affinity score. Heavy parts are penalized by scaling the raw affinity with a linear factor. In assigning the next vertex v in the stream, the LDG algorithm makes its choice based on the following equation:

$$\arg\max_{k \in K} \{|\mathcal{V}_k \cap Adj(v)|(1 - \frac{\omega(\mathcal{V}_k)}{C})\}, \tag{4}$$

where C is the capacity constraint and is equal to total vertex weight divided by the number of parts K, i.e., $C = \frac{\sum_i \omega(v_i)}{K}$. In (4), $|\mathcal{V}_k \cap Adj(v)|$ denotes the raw affinity score of v to part $\mathcal{V}_k$ and $(1 - \frac{\omega(\mathcal{V}_k)}{C})$ denotes the linear penalty factor in assigning that vertex to the same part. This quantity does not take into account the cases where the edges of the graph have weights. This is especially important in our approach as the coarser graphs formed in the multilevel framework result in weighted edges. In assigning a vertex v in the stream, (4) is extended to handle edge weights as:

$$\arg\max_{k \in K} \{\Big(\sum_{\substack{u \in Adj(v) \\ u \in \mathcal{V}_k}} \omega(v,u)\Big)(1 - \frac{\omega(\mathcal{V}_k)}{C})\}. \tag{5}$$

4. Multilevel streaming graph partitioning

In the following sections, we describe how the LDG algorithm is utilized in different phases of the multilevel framework.

4.1. Coarsening with LDG algorithm

In coarsening, the given original graph $\mathcal{G}^0 = (\mathcal{V}^0, \mathcal{E}^0)$ is processed through a number of successive levels to obtain a smaller graph. Each of these levels consists of a partitioning stage that is followed by a coarsening stage. We use a fixed bin size β for the LDG algorithm throughout all coarsening levels. This allows us to determine the number of coarse vertices at each level and have a rough idea how many fine vertices a coarse vertex will contain in any level before running the coarsening phase. We differ from the conventional multilevel coarsening algorithms in the sense that it is usually not possible to know how many vertices there will be in the coarsening levels. For a given β and the desired number of parts, K, we compute the total number of coarsening levels L as:

$$L = \left\lfloor \log_\beta \frac{|\mathcal{V}^0|}{K} \right\rfloor. \tag{6}$$

Here and hereafter, a superscript denotes the level index.

At each level $0 \leq \ell < L$, the graph $\mathcal{G}^\ell = \{\mathcal{V}^\ell, \mathcal{E}^\ell\}$ is first partitioned into K^ℓ bins to get $\Pi^\ell = \{\mathcal{V}_1^\ell, \mathcal{V}_2^\ell, \ldots, \mathcal{V}_{K^\ell}^\ell\}$. Here, each bin represents a part in Π^ℓ and the number of bins/parts at level ℓ is:

$$K^\ell = |\mathcal{V}^\ell|/\beta. \tag{7}$$

Then, Π^ℓ is used to obtain the coarser graph $\mathcal{G}^{\ell+1} = \{\mathcal{V}^{\ell+1}, \mathcal{E}^{\ell+1}\}$ in the next level. Note that $K^{\ell+1} < K^\ell$. Each part in the finer graph in level ℓ becomes a new vertex in the coarser graph in level $\ell+1$. Hence, there are K^ℓ vertices in the coarse graph at level $\ell + 1$:

$$\mathcal{V}^{\ell+1} = \{v_i^{\ell+1} : \mathcal{V}_i^\ell \in \Pi^\ell\}, \text{ where } |\mathcal{V}^{\ell+1}| = K^\ell. \tag{8}$$

The edges between the constituent fine vertices of any two parts in Π^ℓ are coalesced into a single edge between the pair of coarse vertices representing these two parts in the coarse graph. Hence, in $\mathcal{G}^{\ell+1}$, there exists an edge between a pair of coarse vertices if and only if there exists at least one edge between the fine vertices of the respective parts in Π^ℓ, i.e.,

$$\mathcal{E}^{\ell+1} = \{(v_i^{\ell+1}, v_j^{\ell+1}) : Adj(\mathcal{V}_i^\ell) \cap \mathcal{V}_j^\ell \neq \emptyset \wedge i \neq j\}. \tag{9}$$

The neighbors of a coarse vertex are given by:

$$Adj(v_i^{\ell+1}) = \{v_{j\neq i}^{\ell+1} : Adj(\mathcal{V}_i^\ell) \cap \mathcal{V}_j^\ell \neq \emptyset\}. \tag{10}$$

The weight of a coarse vertex is set to the sum of the weights of the finer vertices in the part it is formed from:

$$\omega(v_i^{\ell+1}) = \sum_{v \in \mathcal{V}_i^\ell} \omega(v). \tag{11}$$

Finally, the weight of an edge in $\mathcal{G}^{\ell+1}$ is set to the sum of the weights of the edges between the respective parts in $\mathcal{G}^\ell$ from which the vertices connected by this edge are formed:

$$\omega(v_i^{\ell+1}, v_j^{\ell+1}) = \sum_{v \in \mathcal{V}_i^\ell, u \in \mathcal{V}_j^\ell} \omega(v, u). \tag{12}$$

We note that even in the case where the original graph does not have weighted edges, the coarsening may still lead to weighted edges as multiple cut edges between two parts in a level will be represented by a single edge after coarsening.

We use a relaxed balance constraint in the coarsening, starting with a large imbalance of $1 + \epsilon + \epsilon'$, where $\epsilon \leq \epsilon'$, and reducing it $(\epsilon' - \epsilon)/L$ at each level to get the desired balance with $1 + \epsilon$ at the end. Balancing is less critical in coarsening than it is in uncoarsening, hence we allow a loose constraint in coarsening and use the strict one in uncoarsening. Reducing the strict burden of the balance constraint aids us in better decisions in partitioning stages of the coarsening phase.

4.2. Initial partitioning

The initial partitioning phase simply partitions the coarsest graph $\mathcal{G}^L$ that contains $|\mathcal{V}^L|$ vertices into K parts using the LDG algorithm to get $\Pi^L = \{\mathcal{V}_1^L, \mathcal{V}_2^L, \ldots, \mathcal{V}_K^L\}$. Note that K refers to the desired number of partitions to be obtained on the original graph.

4.3. Uncoarsening with LDG algorithm

The uncoarsening phase starts with the K-way partitioned $\mathcal{G}^L$. Each level ℓ of the uncoarsening consists of refining Π^ℓ on $\mathcal{G}^\ell$ using the LDG algorithm followed by projecting it back to $\Pi^{\ell-1}$ on the finer graph $\mathcal{G}^{\ell-1}$. Note that the number of parts in each Π^ℓ is K. The LDG algorithm is used with each vertex already assigned to some part, but according to the affinity score computed, the vertices may change their parts. It may be difficult to move around vertices in the early levels of the uncoarsening due to very coarse vertices, however in the later stages as the vertices get finer and finer, the LDG algorithm is expected to have more degrees of freedom in improving the cutsize. Moreover, now that all neighbors of a vertex are assigned to some part, the raw affinity scores computed by the LDG algorithm should be more accurate. Projecting back a graph to its finer graph is straightforward as this information is already stored in the coarsening phase and all that needs to be done is simply set the parts of the finer vertices. In projection, a vertex $v_i^\ell \in \mathcal{V}_k^\ell$ is decomposed into its constituent vertices and expected to spawn approximately β new fine vertices, all of which are in $\mathcal{V}_k^{\ell-1}$.

Choosing a large value for β (bin/part size) leads to a small number of coarsening and uncoarsening levels in the multilevel framework. Choosing a small value, on the other hand, may adversely affect the quality of the solutions produced by the LDG algorithm since it leads to many random decisions. This is because the neighbors of a vertex that is about to be assigned to a bin are likely to be scattered across many bins, which in turn causes the affinity scores of different bins to be close to each other.

5. Shared memory parallelization

In this section, we explain in detail how the LDG algorithm is parallelized within the context of multilevel partitioning framework on shared memory systems. We use OpenMP for parallelization.

5.1. Multi-threaded LDG algorithm

Simple structure of the LDG algorithm makes it suitable for parallelization. We divide the vertices of the graph equally among the threads and make each thread responsible for assigning its portion of vertices into parts. Despite being simple, the parallelization of the LDG algorithm requires care in order not to disturb its natural behavior.

In our implementation, partition information is shared among the threads. This information consists of a weight value $\omega(\mathcal{V}_k)$ for each part $\mathcal{V}_k$, and a *part* array storing the part indices of each vertex. For a vertex $v \in \mathcal{V}_k$ we can define $part[v]$ as:

$$part[v] = \begin{cases} k, & \text{if } v \in \mathcal{V}_k, \text{ for } 1 \leq k \leq K \\ \text{NIL}, & \text{otherwise.} \end{cases} \tag{13}$$

The entries in the *part* array are accessed by a single writer and multiple readers. We do not use any synchronization mechanism for this array as the effects of doing so are expected to be minuscule. On the other hand, the part weight information is accessed by multiple readers and multiple writers. The correctness of part weights is important in terms of load balance. Therefore, we manage accesses to this array.

Algorithm 1 Multi-threaded LDG algorithm

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of parts K, capacity C
Output: Vertex partition ($part$), part weights ($\omega(\mathcal{V}_k)$), vertex weights ($\omega(v)$)

```
1: for each v ∈ V in parallel do
2:   part[v] = NIL
3: for k = 1 → K in parallel do
4:   af[k] ← 0 {stores affinity scores}
5: for each u ∈ V in parallel do
6:   partset = ∅
7:   for each v ∈ Adj(u) do
8:     if part[v] = NIL then
9:       continue
10:    k ← part[v]
11:    if af[k] = 0 then
12:      partset ← partset ∪ {k} {neighbors of u}
13:    af[k] ← af[k] + ω(u, v)
14:  k_max ← 0
15:  α_max ← 0
16:  for each k ∈ partset do
17:    α ← af[k] · (1 − ω(V_k)/C)
18:    if α > α_max then
19:      k_max ← k
20:      α_max ← α
21:    af[k] ← 0
22:  while k_max = 0 do
23:    k_ξ ← rand(1, K)
24:    if ω(V_{k_ξ}) + ω(u) ≤ C then
25:      k_max ← k_ξ
26:  part[u] ← k_max
27:  ω(V_{k_max}) ← ω(V_{k_max}) + ω(u) {atomic update}
```

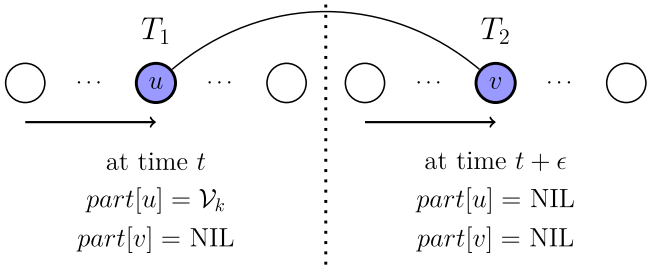


Fig. 2. Although not frequently expected, multi-threaded LDG algorithm may make inferior decisions compared to the sequential version.

Algorithm 2 Multi-threaded coarsening

Input: $\mathcal{G}^\ell = (\mathcal{V}^\ell, \mathcal{E}^\ell)$, number of parts K^ℓ, $part$ (Π^ℓ), part weights ($\omega(\mathcal{V}_k^\ell)$)
Output: $\mathcal{G}^{\ell+1} = (\mathcal{V}^{\ell+1}, \mathcal{E}^{\ell+1})$

```
1: V^{ℓ+1} = ∅
2: for each u ∈ V^ℓ in parallel do
3:   k ← part[u]
4:   P_k ← P_k ∪ {u} {atomic update}
5: for k = 1 → K^ℓ in parallel do
6:   ω(v_k^{ℓ+1}) ← ω(V_k^ℓ)
7:   V^{ℓ+1} = V^{ℓ+1} ∪ {v_k^{ℓ+1}}
8:   for each u ∈ P_k do
9:     for each v ∈ Adj(u) do
10:      p ← part[v]
11:      if p ≠ k then
12:        Adj(v_k^{ℓ+1}) ← Adj(v_k^{ℓ+1}) ∪ {v_p^{ℓ+1}}
13:        ω(v_k^{ℓ+1}, v_p^{ℓ+1}) ← ω(v_k^{ℓ+1}, v_p^{ℓ+1}) + ω(u, v)
```

The multi-threaded LDG algorithm is given in Algorithm 1. A vertex-centric parallelism is adopted (line 5). In the algorithm, the neighbors of a vertex u are processed by first computing u's affinity to the parts in which it has at least one neighbor (lines 6–12). These parts and their affinities are respectively stored in *partset* and *af* arrays. Then, among these parts the one with a feasible capacity and the highest affinity is selected (lines 15–20). If u is the first vertex that is being assigned to a part among its neighbors or none of the parts with at least one neighbor of u has a feasible capacity, we select a random feasible part (lines 21–24). The update of part weights in line 27 is performed atomically.

Since we do not serialize accesses to the *part* array, the multi-threaded LDG algorithm may occasionally make inferior decisions in assigning vertices to the parts compared to the sequential algorithm. The reason is that since they are in a streaming order, vertices are divided among threads and adjacent vertices might be processed by different threads simultaneously. Consider two adjacent vertices u and v (i.e., $(u, v) \in \mathcal{E}$) that are processed near in time by two threads T_1 and T_2, respectively. Assume T_1 assigns u to $\mathcal{V}_k$ while T_2 is still computing the affinity scores for v. T_2 may miss the part assignment decision of its neighbor u and consequently make a rush decision in assigning v (see Fig. 2). Although this does not occur in the sequential case, we expect the effects of this phenomenon not to be severe as the multilevel framework will probably correct it in further executions of the LDG algorithm. For this reason, we do not manage concurrent accesses to the *part* array, which has very positive practical implications in terms of parallel efficiency.

5.2. Multi-threaded coarsening

Multi-threaded coarsening algorithm for constructing the coarser graph $\mathcal{G}^{\ell+1}$ at level $\ell+1$ from the partition Π^ℓ on $\mathcal{G}^\ell = (\mathcal{V}^\ell, \mathcal{E}^\ell)$ is given in Algorithm 2. The algorithm first gathers the vertices in individual parts (lines 2–4) and then processes them to form the adjacency list structure of the coarser graph (lines 5–13). In order to create the coarse graph, a set corresponding to the vertices assigned to each part is created with the *part* array such that vertex u which is located in part k is added to part set $\mathcal{P}_k$. The accesses to these part sets are synchronized, as multiple threads may write to the same set concurrently. In the algorithm, the vertices u and v denote the vertices in level ℓ.

In parallel implementation of the coarsening, each thread is held responsible for constructing coarse vertices of the new level from approximately K^ℓ/N number of parts of the previous level, where N is the number of threads. Each part in level ℓ consists of approximately β vertices. Vertices assigned to each part are processed by a single thread. This leads to a somewhat coarse-grained parallelism, but enables no concurrent accesses to any of the data structures. In addition, the parts in level ℓ are expected to have roughly equal number of vertices, which in turn will very likely lead to a good balance among the threads. A finer level of parallelism may lead to better load balance, but its synchronization overhead due to concurrent updates to the adjacency lists of the coarse vertices is likely to hinder parallel efficiency.

Since the graphs in our work are undirected and stored in adjacency list format, each edge is stored twice. Consider two neighbor vertices u and v in level ℓ, which are respectively assigned to parts $\mathcal{V}_k^\ell$ and $\mathcal{V}_p^\ell$, and to be processed by different threads in coarsening. In updating the adjacency lists of the vertices $v_k^{\ell+1}$ and $v_p^{\ell+1}$ corresponding to these parts, the edge (u, v) will be processed once each of one of the two threads, and each will update the adjacency list that belongs to it, ensuring no concurrent writes. Hence, the edge weights of the coarse graph can be computed seamlessly.

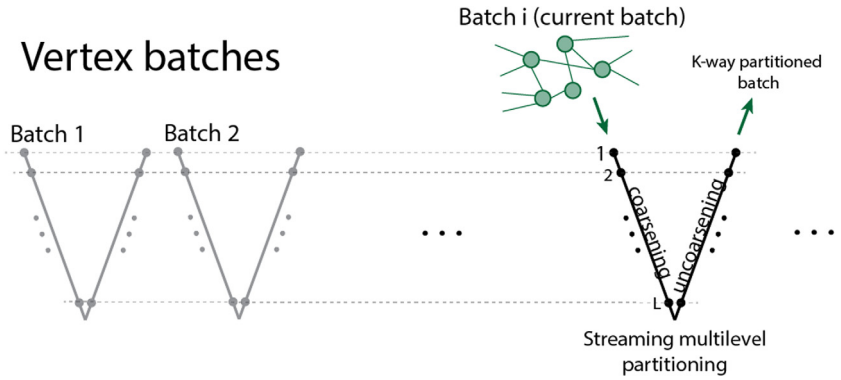


Fig. 3. Streaming multilevel framework adapted to an online context. For each batch, the full multilevel algorithm is executed. The multilevel scheme run for each batch has the same number of levels.

5.3. Multi-threaded uncoarsening

In parallel implementation of the uncoarsening, vertices of the finer graph are divided among the threads and each thread is held responsible for mapping vertices of level ℓ into the parts that their corresponding coarser vertices in level $\ell + 1$ have been assigned to.

After projecting back the coarse vertices in parts into finer vertices, we refine the graph using the LDG algorithm. The parallel LDG algorithm run in uncoarsening is the same with the algorithm given in Algorithm 1 except the fact that there are no unprocessed vertices in uncoarsening and each and every vertex has already been assigned to a part. The refinement in uncoarsening has the positive effect of correcting poor decisions carried out by the LDG algorithm in earlier iterations.

6. Online streaming multilevel partitioning

The methodology described so far uses a streaming algorithm within the multilevel framework in an offline context. In other words, it assumes that the entire graph is available for partitioning at once. Although the theme of our work is offline partitioning, how the multilevel framework would perform in an online context is also worth of investigating. Obviously, the multilevel framework would not make sense if we were to assign a single vertex at a time. However, if we assume that the vertices arrive in batches, then utilizing the LDG algorithm within the multilevel framework can potentially exploit the intra-neighborhood information in the batch better than the flat LDG algorithm can.

Assume n vertices of $\mathcal{G}$ arrive in batches of sizes $n_b \ll n$. For each batch, we can run the full multilevel algorithm described in Section 4 to determine the assignments of the vertices in the batch. The vertices in a batch are coarsened only among themselves through bin assignment. On the other hand, while computing affinities of these vertices to the bins, they also take into account the vertices that are already assigned in the earlier batches. In this setting, the capacities of the bins are computed with respect to batch size instead of the number of vertices in the entire graph. Hence, the capacities of bins are increased accordingly whenever a new batch will be processed and then the multilevel framework is executed. Such a scheme necessitates maintaining the vertex assignments at each level of the multilevel scheme. Although this overhead may seem prohibitive, the number of levels in the multilevel framework is very small since it is determined by the batch size (n_b) and not by the number of vertices in the entire graph (n). This online streaming multilevel partitioning scheme is illustrated in Fig. 3.

This batch scheme is likely to benefit less from shared-memory parallelization described in Section 5 compared to the offline partitioning described earlier. This is due to the fewer vertices handled in each batch and fewer levels in the multilevel framework. In addition, there is no inter-batch parallelism due to the fact that the batches must be processed sequentially.

Table 1
Graphs used in the experiments.

Graph name	Vertices	Edges	Category
ljournal-2008	5 363 260	49 514 271	Social
soc-LiveJournal1	4 847 571	42 851 237	Social
hollywood-2009	1 139 905	56 375 711	Social
Web-notredome	325 729	1 090 108	Web
Web-google	916 428	4 322 051	Web
Eu-2005	862 664	16 138 468	Web
copapersDBLP	540 486	15 245 729	Citation
coAuthorsDBLP	299 067	977 676	Citation
dblp-2010	326 186	807 700	Citation
cit-Patents	3 774 768	16 518 947	Citation
coPapersCiteseer	434 102	16 036 720	Citation
Patents	3 774 768	14 970 766	Citation
hcircuit	105 676	203 734	Circuit
circuit5M	5 558 326	26 983 926	Circuit
Fullchip	2 987 012	11 817 567	Circuit
amazon-2008	735 323	3 523 472	Similarity
Bump_2911	2 911 419	62 409 240	FEM
HV15R	2 017 169	162 357 569	FEM
ML_Laplace	377 002	13 656 485	FEM
Flan_1565	1 564 794	57 920 625	FEM
Dubcova1	16 129	118 440	FEM
WS-1M	1 000 000	10 000 000	Synthetic
WS-5M	5 000 000	55 000 000	Synthetic
WS-10M	10 000 000	110 000 000	Synthetic

7. Experimental results

7.1. Graphs

Our approach is empirically tested on 27 graphs consisting of real world and synthetic datasets. Real world graphs are collected from SNAP [33] archive. These graphs are chosen from the following domains: social networks, web graphs, citation networks, circuit simulation, item similarity and finite element meshes (FEM). Our dataset also contains three synthetic Watts–Strogatz [40] graphs. These synthetic graphs are created using NetworkX [15] package with a rewiring parameter of 0.1. Table 1 summarizes the properties of the tested graphs. All graphs are made undirected and self loops are removed. The vertices and edges in all graphs have unit weights.

7.2. Experimental framework

We implemented our framework in C and compiled all the codes with gcc 4.9.2 in O3 optimization level. In our parallel

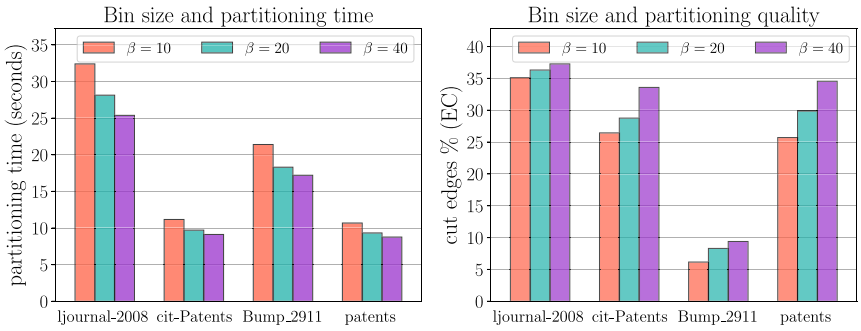


Fig. 4. Partitioning time and edge cut for different bin sizes (β).

implementation, we used OpenMP multi-threading library. Experiments are conducted on a system with four Intel Xeon E7-8860v4 CPUs. Each CPU has 18 cores, clocked at 2.2 GHz, and 32 kB L1 cache and 256 kB L2 cache. The system has 72 cores and 256 GB of memory. All reported results are the average of five runs. The imbalance ratio is set to $\epsilon = 0.10$. It is important to note that in all the experiments that we have reported, at each level of coarsening phase, a single pass of repartitioning is applied after the partitioning stage. This is an effort towards improving the quality of the partitioning.

In evaluating quality of partitions, two metrics are examined. The first metric is the fraction of cut edges (EC), which we simply refer to as edge cut, and the second metric is the load imbalance ratio (LI). These metrics are respectively defined as follows:

$$EC = \frac{cutsize(\Pi)}{\sum_{(v_i,v_j)\in\mathcal{E}} \omega(v_i, v_j)} * 100, \tag{14}$$

$$LI = \frac{\mathcal{W}_{max}}{\mathcal{W}_{avg}}. \tag{15}$$

Here, $\mathcal{W}_{max}$ denotes the weight of maximally loaded part and $\mathcal{W}_{avg}$ denotes average part weight.

To evaluate the proposed framework, we compared our results with the multi-threaded graph partitioner *mt-metis* and the flat LDG algorithm. The flat LDG algorithm is run many times, so that we can assess if the multilevel framework benefits from a simple streaming algorithm. In the following discussions, we refer to our framework as SML (Streaming MultiLevel), the *mt-metis* as MTS, and the flat LDG algorithm as FLL. Before comparing our approach with the described schemes, we first examine the effect of bin size parameter β.

7.3. Bin sizes

We briefly discuss the chosen value for the bin size parameter β and its effect on the quality and performance of our algorithm, SML. Recall that β determines the average number of vertices to be assigned to each part in each level. A small value for β results in more levels of partitioning, which is likely to lead to more improvements in partition quality because the number of times the streaming algorithm is run increases with decreasing β. On the other hand, utilizing a very small β value may also have an adverse effect on the quality of partitions as the affinity scores will not lead to good clusterings in bins containing very few vertices. A large number of levels causes a high partitioning time due to increase in the number of coarsening and uncoarsening levels. This tradeoff between the quality and partitioning time can be seen in Fig. 4 for three values of $\beta = 10$, $\beta = 20$ and $\beta = 40$.

β values smaller than 10 can give very volatile results especially in a multi-threaded environment and values larger than 40 can totally ignore multilevel scheme by reducing number of levels to as low as 1, which would eliminate the coarsening and uncoarsening phases and partition the graph as the LDG algorithm. Although different β values might fit better for different graphs, in our experiments we found out that $\beta = 20$ produces a satisfying tradeoff between the metrics of interest. Hence, we use $\beta = 20$ for SML in the rest of this section.

7.4. Experimental evaluation on all graphs

In this section we compare the performance of our scheme SML against FLL and MTS in terms of LI (Load Imbalance), EC (edge cut), and runtime (in seconds). All the schemes are experimented in shared-memory platform using OpenMP and the number of threads for this experiment is set to 8. In Table 2, we provide a comprehensive evaluation of these three schemes for 32-way partitioning (i.e., $K = 32$).

All the experiments for FLL are reported as the average of 10 passes. FLL is usually stuck in local optima after 10 passes and it does not improve the edge-cut quality for almost all graphs in our dataset after that value. We report both the actual and the normalized values in the table. All the actual values reported in Table 2 are normalized with respect to the results of FLL and the geometric mean for each category and the overall mean for all the graphs are also provided.

We categorized the graphs based on their types in our dataset in Table 2 with their respective geometric means in evaluated metrics. When we compare all the schemes in terms of edge-cut quality, MTS expectedly attains the best results, followed by SML, and the worst in this metric being FLL. When we compare SML and FLL, in FEM category SML has the best category-wise edge-cut improvement over FLL with an average value of 51%. It also leads to significant improvements in categories Web, Citation, Circuit, Similarity, and Synthetic. Only in the Social category SML and FLL produce similar quality partitions. These results show that we can successfully exploit the multilevel framework with a flexible streaming algorithm to significantly improve the partition quality and still do that within the bounds of an acceptable partitioning time.

The reason for SML and FLL to have comparable edge cuts in Social category is that these types of graphs are known to be power law graphs with a high average local clustering coefficient compared to the graphs in other categories, leading to irregular structure around the individual nodes. This impacts the effectiveness of the multilevel partitioning framework [25]. In fact MTS, being a successful offline multilevel graph partitioner, leads to only 17% improvement over FLL. It also has the least category-wise edge-cut improvement over FLL for the Social category, which justifies the claim for multilevel approaches.

When we compare the schemes in terms of partitioning time, SML is on the average 10% faster than FLL in all categories. This might seem odd at first, but the partitioning time of FLL

Table 2
Performance comparison of the proposed streaming multilevel method SML against FLL [35] and MTS [21] for all the graphs in terms of *LI* (load imbalance), *EC* (cut edge ratio %), and runtime in seconds. Normalized values of SML and MTS are calculated with respect to the actual values of FLL based on the respective metrics.

Category	Graph	Actual values									Normalized values (w.r.t. FLL)					
		EC			*LI*			Runtime			*EC*		*LI*		Runtime	
		FLL	MTS	SML	FLL	MTS	SML	FLL	MTS	SML	MTS	SML	MTS	SML	MTS	SML
Social	soc-LiveJournal1	34.94	27.13	35.69	1.04	1.11	1.07	4.07	21.25	4.33	0.76	0.97	1.07	1.02	5.22	1.06
	hollywood-2009	32.47	32.05	32.51	1.05	1.10	1.04	7.31	23.86	3.70	0.78	1.02	1.05	0.99	3.26	0.51
	ljournal-2008	35.78	27.08	34.76	1.05	1.18	1.08	1.23	14.69	1.60	0.99	1.00	1.12	1.02	11.94	1.30
	mean	**34.37**	**28.66**	**34.29**	**1.05**	**1.13**	**1.06**	**3.32**	**19.53**	**2.95**	**0.83**	**1.00**	**1.08**	**1.01**	**5.88**	**0.89**
Web	web-NotreDame	11.72	2.94	11.01	1.02	1.10	1.05	0.07	0.21	0.09	0.25	0.94	1.08	1.03	3.04	1.28
	web-Google	15.80	1.37	9.66	1.00	1.05	1.01	0.24	0.62	0.25	0.09	0.61	1.05	1.01	2.54	1.01
	eu-2005	16.66	5.69	14.57	1.06	1.56	1.07	0.47	1.02	0.47	0.34	0.87	1.47	1.00	2.16	1.00
	mean	**14.56**	**2.84**	**11.57**	**1.03**	**1.22**	**1.04**	**0.20**	**0.51**	**0.22**	**0.20**	**0.79**	**1.18**	**1.01**	**2.56**	**1.09**
Citation	cit-Patents	37.47	14.85	28.65	1.00	1.08	1.03	0.40	0.99	0.34	0.67	0.88	1.08	1.03	2.51	0.85
	coPapersDBLP	22.04	14.73	19.29	1.01	1.05	1.04	0.09	0.26	0.08	0.57	0.88	1.04	1.03	2.95	0.94
	coAuthorsDBLP	28.14	16.18	24.82	1.00	1.04	1.03	0.07	0.22	0.07	0.48	0.81	1.04	1.03	3.11	1.02
	dblp-2010	22.93	11.12	18.52	1.00	1.07	1.01	2.14	7.26	1.80	0.40	0.76	1.07	1.01	3.39	0.84
	coPapersCiteseer	14.24	8.20	11.22	1.02	1.08	1.01	0.36	0.72	0.29	0.58	0.79	1.06	0.99	2.02	0.82
	patents	38.95	14.39	29.12	1.00	1.06	1.00	2.07	6.65	1.76	0.37	0.75	1.06	1.00	3.21	0.85
	mean	**25.81**	**12.92**	**20.89**	**1.01**	**1.06**	**1.02**	**0.39**	**1.12**	**0.35**	**0.50**	**0.81**	**1.06**	**1.01**	**2.82**	**0.88**
Circuit	hcircuit	23.03	1.34	17.75	1.00	1.13	1.00	0.02	0.05	0.02	0.06	0.77	1.13	1.00	2.29	0.86
	circuit5M	34.35	28.13	34.60	1.02	1.07	1.07	2.16	47.94	2.71	0.82	1.01	1.05	1.05	22.16	1.25
	Fullchip	52.25	36.41	48.16	1.03	1.58	1.04	1.43	2.04	1.12	0.70	0.92	1.54	1.02	1.43	0.78
	mean	**34.57**	**11.11**	**30.93**	**1.02**	**1.24**	**1.04**	**0.40**	**1.67**	**0.38**	**0.32**	**0.89**	**1.22**	**1.02**	**4.17**	**0.94**
Similarity	amazon-2008	28.80	9.43	19.01	1.01	1.10	1.02	0.30	0.70	0.23	0.33	0.66	1.09	1.01	2.30	0.76
FEM	Bump_2911	30.68	4.13	11.73	1.00	1.05	1.01	3.08	2.58	2.77	0.13	0.38	1.05	1.00	0.84	0.90
	HV15R	26.44	6.65	11.48	1.00	1.08	1.03	4.35	6.33	4.31	0.25	0.43	1.07	1.03	1.45	0.99
	ML_Laplace	21.16	4.13	13.44	1.00	1.05	1.02	0.47	2.58	0.34	0.20	0.64	1.05	1.02	5.45	0.71
	Flan_1565	21.99	2.81	9.36	1.00	1.04	1.03	1.49	2.08	1.52	0.13	0.43	1.04	1.02	1.39	1.02
	Dubcova1	23.29	8.04	14.26	1.00	1.02	1.01	0.01	0.02	0.00	0.35	0.61	1.02	1.00	4.17	0.85
	mean	**24.48**	**4.80**	**11.93**	**1.00**	**1.05**	**1.02**	**0.56**	**1.16**	**0.50**	**0.20**	**0.49**	**1.04**	**1.01**	**2.07**	**0.89**
Synthetic	WS-1M	25.50	16.12	18.71	1.00	1.10	1.04	0.55	1.40	0.44	0.63	0.73	1.10	1.04	2.56	0.81
	WS-5M	25.37	16.14	19.58	1.00	1.10	1.01	4.34	9.97	3.72	0.64	0.77	1.10	1.01	2.30	0.86
	WS-10M	25.36	16.10	19.54	1.00	1.10	1.04	11.46	23.42	10.50	0.63	0.77	1.10	1.04	2.04	0.92
	mean	**25.41**	**16.12**	**19.27**	**1.00**	**1.10**	**1.03**	**3.01**	**6.88**	**2.58**	**0.63**	**0.76**	**1.10**	**1.03**	**2.29**	**0.86**
	Overall mean	**25.61**	**9.57**	**19.02**	**1.01**	**1.11**	**1.03**	**0.65**	**1.89**	**0.59**	**0.37**	**0.74**	**1.10**	**1.02**	**2.91**	**0.91**

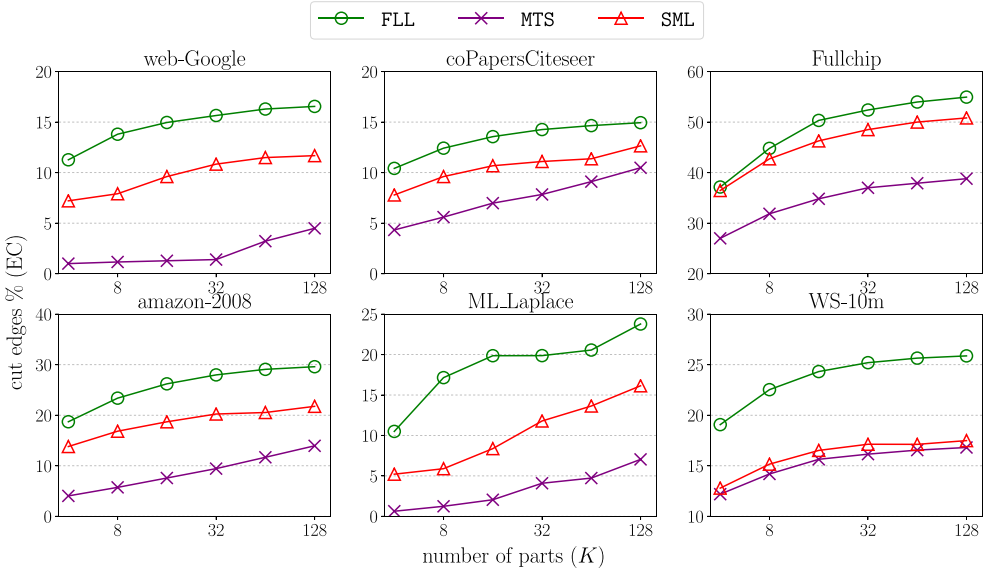
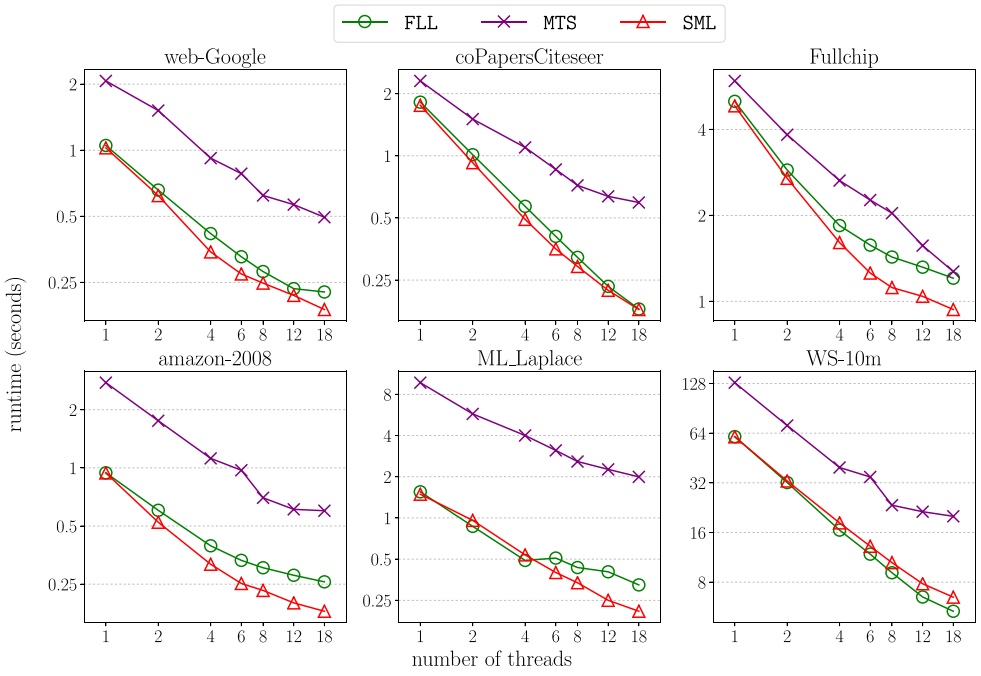
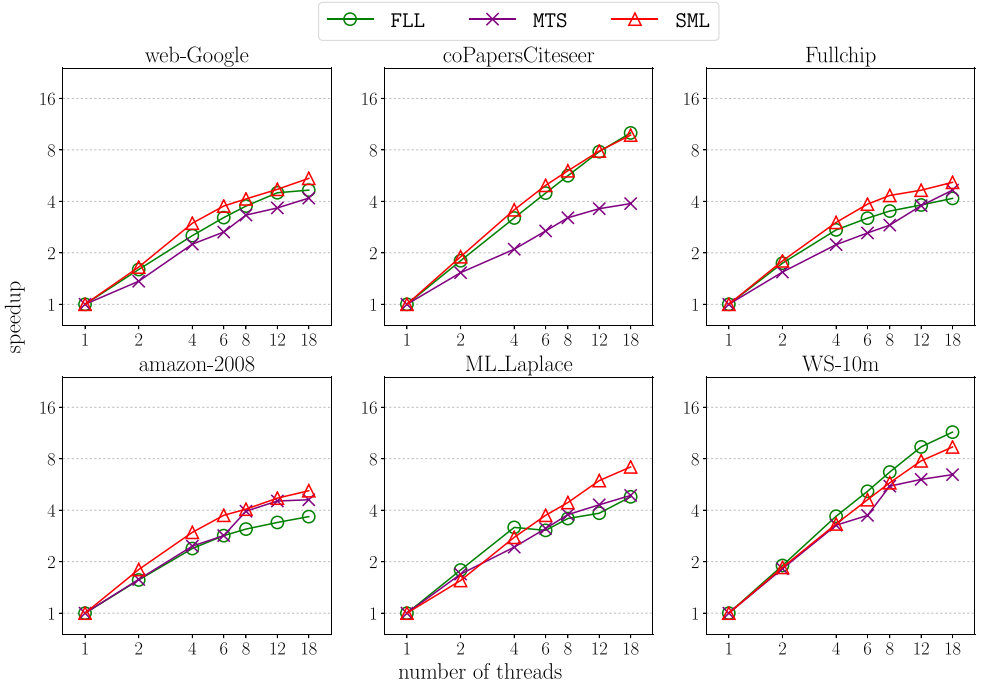


Fig. 5. Edge-cut quality comparison of FLL, SML, and MTS for varying number of parts (K) on different graphs from different categories.

corresponds to 10 passes on the original input graph. On the other hand, SML utilizes the streaming algorithm probably more than 10 times, but most of them are on the graphs that are smaller than the original graph. MTS is much more slower on the average compared to FLL and SML. In FEM graphs, MTS is relatively faster compared to the graphs in other categories. Yet in this category its running time is still 107% and 118% higher than those of FLL and SML, respectively. On grand average, MTS is 2.91 slower than FLL, while SML and FLL have comparable running times. In general, using a fast lightweight greedy algorithm (i.e., LDG) within a multilevel approach we are able to produce relatively good quality partitions by improving edge cut 26% compared to FLL while having nearly the same load imbalance ratio as FLL, with only a 2% of slack. Despite the fact that MTS in general provides better quality partitions (with a 63% improvement over FLL on the average and a 36% over SML), its runtime is much



(a)



(b)

Fig. 6. Running time and speedup plots of FLL, MTS and SML for 32-way partitioning with increasing number of threads on 6 different matrices.

higher than FLL and SML: 191% slower than FLL and nearly 220% slower than SML. This is, however, hardly surprising as MTS utilizes more sophisticated algorithms in each multilevel phase and as a result it has better improvements at the cost of slower runtime.

In Fig. 5, we present the edge-cut percentage of FLL, MTS, and SML for $K = \{4, 8, 16, 32, 64, 128\}$ for 6 graphs, each from a different category. In all these 6 graphs, SML exhibits a steady performance that is in between FLL and MTS. In the synthetic graph WS-10m, SML performs nearly the same as MTS for all values of K. In three irregular graphs, amazon-2008, web-google and coPapersCiteseer the performance of SML is roughly in halfway between FLL and MTS, while in the regular graph Fullchip, SML performs close to FLL.

7.5. Scalability

In this section we compare the running time and scalability of SML, MTS, and FLL with varying number of threads. Table 3 presents the average runtimes (in seconds) of the compared

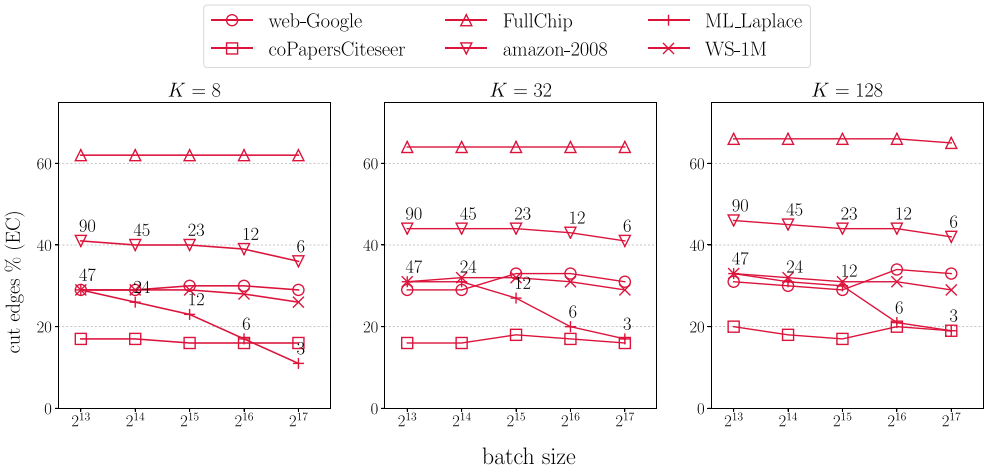


Fig. 7. Edge-cut quality comparison of the multilevel framework utilized within online context for varying batch size on different graphs from different categories. The annotated values for matrices `amazon-2008` and `ML_Laplace` denote the number of batches.

Table 3
Running times (in seconds) of FLL, MTS, and SML for different number of threads for 32-way partitioning. The values are the averages of all graphs.

#threads	FLL	MTS	SML
2	2.22	6.36	1.62
4	1.26	3.99	0.93
6	0.92	3.36	0.70
8	0.75	2.55	0.59
12	0.61	2.31	0.48
18	0.53	2.11	0.41

schemes in all datasets for $K = 32$. FLL is again fixed to 10 passes. Fig. 6 displays the results of strong scaling experiments on 6 graph instances in two different forms. Fig. 6a shows the variation of parallel running time with increasing number of threads, whereas Fig. 6b displays the speedup.

As seen from Table 3 and Fig. 6, MTS exhibits inferior scalability compared to FLL and SML. On the other hand, while SML on the average is faster than FLL, it is slightly less scalable compared to FLL. The reason is that SML is a multilevel approach with coarsening and uncoarsening phases, each of which consists of a number of successive stages and the coarser the graph gets, the less the parallelism it exhibits. The running time and scalability of SML are comparable to those of FLL while the running time and scalability of MTS are worse than these two schemes.

7.6. Assessment of online multilevel framework

We evaluate the performance of multilevel framework when used in an online context as described in Section 6. We assess the edge-cut percentage obtained by partitioning 6 graphs, each from different category, for $K = \{8, 32, 128\}$, and for 5 different batch sizes of $\{2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}\}$. We plot edge-cut percentage against batch size to assess how different batch sizes affect the quality of the obtained partitions. The three plots for three different K values are presented in Fig. 7. The annotation points in the plots for matrices `amazon-2008` and `ML_Laplace` denote the number of batches for the respective batch size.

As seen in Fig. 7, the edge cut decreases or stays the same with increased batch size. This is mainly because the larger the batch size, the higher the likelihood that the batch will contain vertices that are in the neighborhood of each other, and hence the multilevel framework may exploit those relations in order to get a smaller edge cut. Another reason is that the multilevel framework will benefit from larger batch sizes more because otherwise the multilevel scheme will approximate to the flat algorithm when the batch size is small due to the reduced number of levels. As the number of batches decreases, the online partitioning becomes no different than the offline partitioning. This is best seen in matrices `amazon-2008` and `ML_Laplace` in Fig. 7. When the batch size is 2^{17}, the number of batches for `ML_Laplace` is three and its edge cut gets close to that of offline partitioning. It can be said that as the number of batches increases, the quality of the partitions obtained by the online multilevel partitioning decreases or stays the same. This is because the locality in the graph gets more fragmented across different batches when we have more batches. As seen in Fig. 7, the edge cut often gradually gets worse as the number of batches increases, and then it consolidates at a certain point. Note that the vertices in our experiments are randomly ordered. Using a breadth-first or depth-first order is likely to make the threshold where the edge cut consolidates higher.

8. Conclusion

We proposed a fast parallel streaming multilevel graph partitioning method. Instead of using several different expensive algorithms for different stages of the multilevel framework, we utilized a single lightweight, easy-to-parallelize and flexible streaming algorithm throughout the partitioning. We parallelized this algorithm within the multilevel framework and tested it extensively against the state-of-the-art offline partitioner *mt-metis* as well as against flat streaming heuristics to see whether the multilevel framework can really exploit such a simple algorithm. Our results indicate that our approach can attain good quality partitions for certain classes of graphs much faster than *mt-metis*. We also demonstrated how it scales with varying number of threads.

As future work, we consider testing other fast streaming graph partitioning heuristics. This is a promising direction as some heuristics can suit better for certain stages of the multilevel framework. In addition, different heuristics can exhibit better performance for certain types of graphs. The second future direction is the testing of different vertex visit orders. It is shown in other works that various orders can significantly affect the edge-cut quality.

CRediT authorship contribution statement

Nazanin Jafari: Methodology, Software, Validation, Investigation, Resources, Data curation, Writing - original draft, Visualization. **Oguz Selvitopi:** Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing. **Cevdet Aykanat:** Conceptualization, Methodology, Writing - review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by the Director, Office of Science, U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Y. Akhremtsev, P. Sanders, C. Schulz, High-quality shared-memory graph partitioning, in: M. Aldinucci, L. Padovani, M. Torquati (Eds.), Euro-Par 2018: Parallel Processing, Springer International Publishing, Cham, 2018, pp. 659–671.

[2] Apache giraph, Apache software foundation, 2019, URL http://giraph.apache.org.

[3] S.T. Barnard, H.D. Simon, Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, Concurrency, Pract. Exp. 6 (2) (1994) 101–117, http://dx.doi.org/10.1002/cpe.4330060203, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4330060203 URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330060203.

[4] C. Battaglino, P. Pienta, R. Vuduc, GraSP: distributed streaming graph partitioning, in: HPGM: High Performance Graph Mining. "1st High Performance Graph Mining Workshop", 2015, http://dx.doi.org/10.5821/hpgm15.3.

[5] U. Benlic, J. Hao, A multilevel memetic approach for improving graph k-partitions, IEEE Trans. Evol. Comput. 15 (5) (2011) 624–642, http://dx.doi.org/10.1109/TEVC.2011.2136346.

[6] T.N. Bui, C. Jones, A heuristic for reducing fill-in in sparse matrix factorization, in: PPSC, 1993, pp. 445–452.

[7] U. Catalyurek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 10 (1999) 673–693, http://dx.doi.org/10.1109/71.780863, URL http://portal.acm.org/citation.cfm?id=311796.311798.

[8] C. Chevalier, F. Pellegrini, PT-scotch: A tool for efficient parallel graph ordering, Parallel Comput. 34 (6) (2008) 318–331, http://dx.doi.org/10.1016/j.parco.2007.12.001, URL http://www.sciencedirect.com/science/article/pii/S0167819107001342 Parallel Matrix Algorithms and Applications.

[9] P. Ciarlet, F. Lamour, On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint, Numer. Algorithms 12 (1) (1996) 193–214, http://dx.doi.org/10.1007/BF02141748.

[10] R. Diekmann, R. Preis, F. Schlimbach, C. Walshaw, Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM, Parallel Comput. 26 (12) (2000) 1555–1581, http://dx.doi.org/10.1016/S0167-8191(00)00043-0, URL http://www.sciencedirect.com/science/article/pii/S0167819100000430 Graph Partitioning and Parallel Computing.

[11] W.E. Donath, A.J. Hoffman, Algorithms for partitioning of graphs and computer logic based on eigenvectors of connected matrices, in: IBM Technical Disclosure Bulletin, Vol. 15.3, 1972, pp. 938–944.

[12] M. Fiedler, A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory, Czechoslovak Math. J. 25 (4) (1975) 619–633, URL http://eudml.org/doc/12900.

[13] H. Firth, P. Missier, Workload-aware streaming graph partitioning, in: EDBT/ICDT Workshops, 2016.

[14] T. Goehring, Y. Saad, Heuristic Algorithms for Automatic Graph Partitioning, Tech. Rep., University of Minnesota, 1994.

[15] A. Hagberg, P. Swart, D. S. Chult, Exploring network structure, dynamics, and function using networkx, 0000.

[16] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM), in: Supercomputing '95, ACM, New York, NY, USA, 1995, URL http://doi.acm.org/10.1145/224170.224228 doi:http://doi.acm.org/10.1145/224170.224228.

[17] B. Hendrickson, R. Leland, S. Plimpton, An efficient parallel algorithm for matrix-vector multiplication, Int. J. High Speed Comput. 7 (1995) 73–88.

[18] G. Karypis, V. Kumar, METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0, Tech. Rep., 1995.

[19] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. (ISSN: 1064-8275) 20 (1) (1998) 359–392, http://dx.doi.org/10.1137/S1064827595287997, URL http://dx.doi.org/10.1137/S1064827595287997.

[20] B. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell Syst. Tech. J. 49 (1970) 291–307.

[21] D. Lasalle, G. Karypis, Multi-threaded graph partitioning, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 225–236, http://dx.doi.org/10.1109/IPDPS.2013.50.

[22] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed graphlab: A framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (8) (2012) 716–727, http://dx.doi.org/10.14778/2212351.2212354.

[23] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, in: SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146, http://dx.doi.org/10.1145/1807167.1807184, URL http://doi.acm.org/10.1145/1807167.1807184.

[24] C. Mayer, R. Mayer, M.A. Tariq, H. Geppert, L. Laich, L. Rieger, K. Rothermel, ADWISE: Adaptive window-based streaming edge partitioning for high-speed graph processing, in: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 685–695, http://dx.doi.org/10.1109/ICDCS.2018.00072.

[25] J. Nishimura, J. Ugander, Restreaming graph partitioning: Simple versatile algorithms for advanced balancing, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, in: KDD '13, ACM, New York, NY, USA, 2013, pp. 1106–1114, http://dx.doi.org/10.1145/2487575.2487696, URL http://doi.acm.org/10.1145/2487575.2487696.

[26] F.c. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: H. Liddell, A. Colbrook, B. Hertzberger, P. Sloot (Eds.), High-Performance Computing and Networking, in: Lecture Notes in Computer Science, vol. 1067, Springer Berlin Heidelberg, 1996, pp. 493–498, http://dx.doi.org/10.1007/3-540-61142-8_588.

[27] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, G. Iacoboni, HDRF: Stream-based partitioning for power-law graphs, in: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, in: CIKM '15, ACM, New York, NY, USA, 2015, pp. 243–252, http://dx.doi.org/10.1145/2806416.2806424, URL http://doi.acm.org/10.1145/2806416.2806424.

[28] A. Pothen, H. Simon, K. Liou, Partitioning sparse matrices with eigenvectors of graphs, SIAM J. Matrix Anal. Appl. 11 (3) (1990) 430–452, http://dx.doi.org/10.1137/0611030, arXiv:https://doi.org/10.1137/0611030.

[29] U.N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Phys. Rev. E 76 (2007) 036106, http://dx.doi.org/10.1103/PhysRevE.76.036106, URL https://link.aps.org/doi/10.1103/PhysRevE.76.036106.

[30] H.P. Sajjad, A.H. Payberah, F. Rahimian, V. Vlassov, S. Haridi, Boosting vertex-cut partitioning for streaming graphs, in: 2016 IEEE International Congress on Big Data (BigData Congress), 2016, pp. 1–8, http://dx.doi.org/10.1109/BigDataCongress.2016.10.

[31] P. Sanders, C. Schulz, Engineering multilevel graph partitioning algorithms, in: C. Demetrescu, M.M. Halldórsson (Eds.), Algorithms – ESA 2011, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 469–480.

[32] K. Schloegel, G. Karypis, V. Kumar, Parallel static and dynamic multi-constraint graph partitioning, Concurr. Comput.: Pract. Exper. 14 (3) (2002) 219–240, http://dx.doi.org/10.1002/cpe.605, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.605 URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.605.

[33] SNAP: Stanford network analysis project, 2019, URL http://snap.stanford.edu/snap/.

[34] A. Soper, C. Walshaw, M. Cross, A combined evolutionary search and multilevel optimisation approach to graph-partitioning, J. Global Optim. 29 (2) (2004) 225–241, http://dx.doi.org/10.1023/B:JOGO.0000042115.44455.f3.

[35] I. Stanton, G. Kliot, Streaming graph partitioning for large distributed graphs, in: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, in: KDD '12, ACM, New York, NY, USA, 2012, pp. 1222–1230, http://dx.doi.org/10.1145/2339530.2339722, URL http://doi.acm.org/10.1145/2339530.2339722.

[36] C. Tsourakakis, Streaming graph partitioning in the planted partition model, in: Proceedings of the 2015 ACM on Conference on Online Social Networks, in: COSN '15, ACM, New York, NY, USA, 2015, pp. 27–35, http://dx.doi.org/10.1145/2817946.2817950, URL http://doi.acm.org/10.1145/2817946.2817950.

[37] C. Tsourakakis, C. Gkantsidis, B. Radunovic, M. Vojnovic, FENNEL: Streaming graph partitioning for massive scale graphs, in: Proceedings of the 7th ACM International Conference on Web Search and Data Mining, in: WSDM '14, ACM, New York, NY, USA, 2014, pp. 333–342, http://dx.doi.org/10.1145/2556195.2556213, URL http://doi.acm.org/10.1145/2556195.2556213.

[38] C. Walshaw, M. Cross, Mesh partitioning: A multilevel balancing and refinement algorithm, SIAM J. Sci. Comput. 22 (1) (2000) 63–80, http://dx.doi.org/10.1137/S1064827598337373, arXiv:https://doi.org/10.1137/S1064827598337373.

[39] C. Walshaw, M. Cross, in: F. Magoules (Ed.), Mesh Partitioning Techniques and Domain Decomposition Methods, Saxe-Coburg Publications, Stirling, Scotland, UK, 2007, pp. 27–58, Chapter. JOSTLE: parallel multilevel graph-partitioning software an overview.

[40] D.J. Watts, S.H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442, http://dx.doi.org/10.1038/30918.

Nazanin Jafari received her master's degree from Bilkent University, at the Department of Computer Engineering. She is currently working towards her Ph.D. degree in the College of Information and Computer Science, UMass Amherst, MA, USA. Her research interests include high-performance computing, information retrieval and big data analysis.

Oguz Selvitopi received his B.S. degree from Marmara University, Computer Science and Engineering Department, Istanbul, Turkey in 2008, and M.S. and Ph.D. degrees from Bilkent University, Computer Engineering Department, Ankara, Turkey, in 2010 and 2016, respectively. His research interests are parallel computing, scientific computing, parallel and distributed systems, and bioinformatics. He is currently a post-doctoral fellow in Computational Research Division at Lawrence Berkeley National Laboratory.

Cevdet Aykanat received the B.S. and M.S. degrees from Middle East Technical University, Ankara, Turkey, both in Electrical Engineering, and the Ph.D. degree from Ohio State University, Columbus, in Electrical and Computer Engineering. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a Professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects. He is the recipient of the 1995 Young Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He has served as an associate editor of IEEE Transactions of Parallel and Distributed Systems between 2008 and 2012.