pdGRASS: A Fast Parallel Density-Aware Algorithm for Graph Spectral Sparsification

Tiancheng Zhao
Georgia Institute of Technology
tzhao350@gatech.edu

Zekun Yin
Shandong University
zekun.yin@sdu.edu.cn

Huihai An
Shandong University
202335297@mail.sdu.edu.cn

Xiaoyu Yang
China University of Petroleum-Beijing
yangxiaoyu@student.cup.edu.cn

Zhou Jin
Zhejiang University
z.jin@zju.edu.cn

Jiasi Shen
Hong Kong University of Science and Technology
sjs@cse.ust.hk

Helen Xu
Georgia Institute of Technology
hxu615@gatech.edu

***Abstract*—Graph Spectral Sparsification (GSS) identifies an ultra-sparse subgraph, or *sparsifier*, whose Laplacian matrix closely approximates the spectral properties of the original graph, enabling substantial reductions in computational complexity for computationally intensive problems in scientific computing. The state-of-the-art method for efficient GSS is *feGRASS*, consisting of two steps: 1) spanning tree generation and 2) off-tree edge recovery. However, feGRASS suffers from two main issues: 1) difficulties in parallelizing the recovery step for strict data dependencies, and 2) performance degradation on skewed inputs, often requiring multiple passes to recover sufficient edges.**

To address these challenges, we propose *parallel density-aware Graph Spectral Sparsification* (pdGRASS), a parallel algorithm that organizes edges into disjoint *subtasks* without data dependencies between them, enabling efficient parallelization and sufficient edge recovery in a single pass. We empirically evaluate feGRASS and pdGRASS based on 1) off-tree edge-recovery *runtime* and 2) sparsifier quality, measured by the *iteration count* required for convergence in a preconditioned conjugate gradient (PCG) application. The evaluation demonstrates that, depending on the number of edges recovered, pdGRASS achieves average speedups ranging from $3.9\times$ to $8.8\times$. The resulting sparsifiers also show between $1.2\times$ higher and $1.8\times$ lower PCG iteration counts, with further improvements as more edges are recovered. Additionally, pdGRASS mitigates the worst-case runtimes of feGRASS with over $1000\times$ speedup. These results highlight pdGRASS's significant improvements in scalability and performance for the graph spectral sparsification problem.

***Index Terms*—graph sparsification, parallel subtask division**

I. Introduction

Graph spectral sparsification (GSS) aims to construct a ***sparsifier***, which is a nearly-linear-sized subgraph that can approximate the spectral information (i.e., eigenvalues and eigenvectors) of the original graph. Researchers have investigated this problem in both theory [1]–[4] and practice [5]–[10]. Using the sparsifier rather than the original graph facilitates near-linear-time algorithms for a wide variety of computationally-intensive matrix and graph applications such as scientific computing [4], max-flow problems in graphs [4], [11]–[13], data mining [14], social network analysis [15], solving large systems of equations [16], machine learning [17], and computer-aided design (CAD) for very large-scale integration (VLSI) [5], [18]. Due to the heavy computational complexity of these applications, graph sparsification is necessary to solve large-scale problems in a reasonable timeframe. Unfortunately, most existing implementations of algorithms for GSS are sequential and stop short of parallelization.

Edge recovery in sparsification. The state-of-the-art method for GSS is feGRASS [7] and pGRASS [19] (a parallel implementation of feGRASS but not open-sourced), which performs two key steps: 1) spanning-tree generation and 2) recovering "dissimilar" spectrally-critical off-tree edges that were not chosen in the first phase but are necessary for improving the approximation quality. The main challenge in designing parallel algorithms for GSS is the *off-tree edge recovery* phase, as spanning-tree generation has well-known optimized solutions.

Challenges to parallelization from data dependencies. An off-tree edge can be recovered only if it is not similar to all previous recovered edges. In other words, each iteration is inherently dependent on the results of the previous iterations, imposing a sequential constraint on the overall process. The state-of-the-art *parallel* implementation for GSS is pGRASS [19], [20], which parallelizes over edges in batches but incurs *excess work*. Edges in the same batch can be similar edges. Specifically, if there are p threads, pGRASS performs similarity check on the next p edges simultaneously while there can be redundant computation, which is unavoidable for the correctness of the parallel algorithm.

Since there is no available implementation for pGRASS, we compare our algorithm directly with feGRASS.

Worst-case inputs. The off-tree edge recovery phase only stops when the algorithm recovers a fixed number of edges of $\alpha|V|$, where α is a predefined ratio that specifies the proportion of edges to be recovered (default value is 2%) and $|V|$ is the number of vertices. For a small α such as 0.01, one pass through the off-tree edges is sufficient to recover enough edges, but when α is set to a moderately larger value, such as 0.05, the feGRASS strategy may require many passes through the off-tree edges to recover enough edges to meet the required threshold. In an extreme case, on the *com-Youtube* graph in

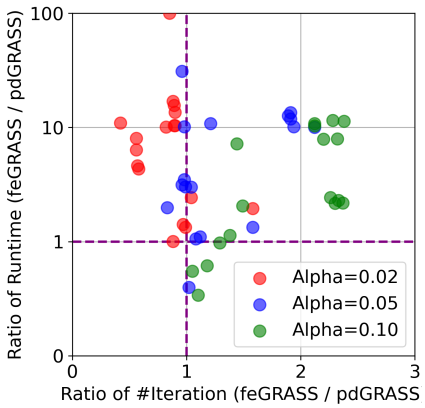

Fig. 1: Relative off-tree-edge recovery time and PCG iteration count (feGRASS/ pdGRASS) for 18 test graphs. A number above 1 in either metric means that pdGRASS improves along that metric compared to feGRASS.

part V, we found that feGRASS requires over 6000 passes to recover $0.02|V|$ edges. It reveals that algorithm based on vertex cover degrades severely on certain special graph structures.

Density-aware graph spectral sparsification. To address these issues, we propose ***parallel density-aware Graph Spectral Sparsification*** (pdGRASS), a fast and accurate parallel algorithm for GSS. Like feGRASS and pGRASS, pdGRASS follows a two-step framework but introduces two key modifications: 1) changing the edge-similarity condition to a *"strict"* condition to recover more edges in one pass for various graph structures, which gives rise to 2) a careful parallelization strategy based on division of the entire problem into *independent "subtasks"*. The proposed subtask-level parallelization significantly reduces the data dependencies between edges in feGRASS and pGRASS through subtask partitioning. Furthermore, we find that the strict condition enables pdGRASS to complete recovery in a single pass by retaining more edges.

Mixed parallel strategy. We empirically evaluate several parallelization strategies for pdGRASS and propose a ***mixed*** approach that exploits parallelism both *within* and *across* subtasks. This is necessary because real-world graphs, characterized by a few high-degree and many low-degree vertices, lead to skewed subtask sizes. Parallelizing only along one dimension would leave performance on the table and unnecessarily serialize the computation. Meanwhile, a mixed parallel strategy can adapt the ratio of the two parallel strategies based on the number of available cores, achieving better speedup.

Results summary. We evaluate the state-of-the-art serial algorithm feGRASS[1] [7] and the proposed pdGRASS based on: 1) their off-tree edge recovery *runtime*, and 2) the *iteration count* required for convergence in a downstream preconditioned conjugate gradient (PCG) [21] solver. Figure 1 summarizes the results in terms of runtime and PCG iteration count on a suite of 18 input graphs. As α increases and more edges are required to be recovered, the data points collectively shift toward the upper right direction, reflecting consistent improvements of pdGRASS over feGRASS in both run-time efficiency and sparsifier quality as shown in Figure 1. Finally, as detailed in Section V, pdGRASS achieves strong parallel scaling on both uniform and skewed inputs, enabled by its mixed parallel strategy across and within subtasks. On 32 threads, pdGRASS attains an average parallel speedup of $11.3\times$ for $\alpha = 0.02$, and further improves to $13.7\times$ and $14.8\times$ for 0.05 and 0.10.

[1]We compare with feGRASS because no open-source implementation of pGRASS is available.

II. Preliminaries

This section presents background and current algorithms for graph spectral sparsification problem. We introduce key concepts such as the graph Laplacian matrix and then review with the efficient serial algorithm feGRASS [7] and pGRASS [19].

A. Graph Spectral Sparsification

The Graph Spectral Sparsification (GSS) problem takes a weighted undirected graph $G = (V, E, w)$ as input and generates an ultra-sparse subgraph (also called a ***sparsifier***) P as output which is "spectrally similar" to the original graph G.

We first define the ***Laplacian matrix*** of a graph G, denoted by $L_G \in \mathbb{R}^{|V|\times|V|}$.

$$L_G(i,j) = \begin{cases} -w_{i,j}, & (i,j) \in E \\ \sum\limits_{(i,k)\in E} w_{i,k}, & i = j \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

A graph P is ***σ-spectrally similar*** to a graph G if for any $u \in \mathbb{R}^{|V|}$,

$$\frac{1}{\sigma} u^T L_P u \leq u^T L_G u \leq \sigma u^T L_P u \tag{2}$$

where L_G and L_P are the Laplacian matrices of graphs G and P, respectively [22]. If graphs G and P are σ-spectrally similar, then $\kappa(L_G, L_P) \leq \sigma^2$, where $\kappa(L_G, L_P)$ denotes the relative condition number[2] of their Laplacian matrices.

B. feGRASS Algorithm [7]

feGRASS is an efficient serial algorithm for GSS, which consists of two main steps: 1) spanning-tree generation based on the *Effective Weight* of edges and 2) off-tree edge recovery based on the *Resistance Distance* of paths in the spanning tree.

Definition 1 (Effective weight)**.** *The **effective weight** [7] W_{eff} of an edge $e = (i, j)$ is defined as follows:*

$$W_{\text{eff}}(e) = w_{u,v} \times \frac{\log(\max\{\deg(u), \deg(v)\})}{\text{dist}_G(root, u) + \text{dist}_G(root, v)} \tag{3}$$

where G denotes the original graph and $root$ is the vertex with the maximum degree in G and $\text{dist}_G(\cdot)$ refers unweighted distance, which can be computed by Breadth-First Search(BFS).

[2]The condition number of two matrices A and B is defined as $\kappa(A, B) = \lambda_{\max}(A, B)/\lambda_{\min}(A, B)$ where $\lambda_{\max}$ and $\lambda_{\min}$ denote the largest and smallest nonzero generalized eigenvalues, respectively.

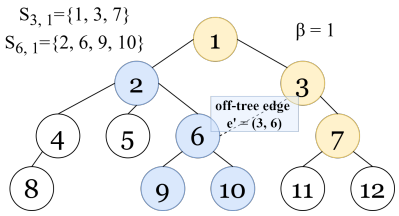

Fig. 2: An illustration of the similarity check for a recovered off-tree edge $e = (3, 6)$ with BFS step size $\beta = 1$. The blue vertices represent S_6 and yellow vertices represent S_3.

Definition 2 (Resistance distance)**.** *Given a spanning tree T, the **resistance distance** R_T (paraphrased[3] from [7]) of an edge $e = (u, v)$ is defined as*

$$R_T(u, v) = \text{dist}_{re}(u, \text{LCA}_T(u, v)) + \text{dist}_{re}(v, \text{LCA}_T(u, v)), \tag{4}$$

where dist_{re} is distance of u, v based on resistant weight W_{re} and $W_{re}(e) = 1/w(e)$.

After spanning-tree generation, each edge in the original graph belongs to either the ***tree edge*** set and ***off-tree edge*** set. The next step is off-tree edge recovery based on the concept of "resistance distance" from the theory of electrical resistance networks [3], [24] which determines spectrally criticality.

Intuitively, two off-tree edges are "similar" if they have "similar positions on the spanning tree". Formally, similarity is determined by vertex neighborhoods of the endpoints of an off-tree edge, which can be computed with BFS process.

Definition 3 (BFS step size and vertex neighborhoods)**.** *Let $e = (u, v)$ be a recovered off-tree edge and let T be the spanning tree from the first step. The corresponding **BFS step size** β is defined as a small constant c (default value is 8) in the feGRASS algorithm and let $S_{u,\beta}$ (resp., $S_{v,\beta}$) be the **β-hop neighborhood** of vertex u (resp., vertex v). That is, $S_{u,\beta}$ is a set containing all vertices with distance at most β from u (i.e., all vertices visited in β BFS iterations from u):*

$$S_{u,\beta} = \{v : dist(u, v) \leq \beta, \beta = c\} \tag{5}$$

Definition 4 (Similiar edge (Loose Definition))**.** *Suppose we have a spanning tree T, an off-tree edge $e = (u, v)$, and BFS step size β and get two β-hop vertex neighborhood sets $S_{u,\beta}, S_{v,\beta}$.*

*An off-tree edge $e' = (u', v')$ is **similar** to off-tree edge e if*

$$(u' \in S_{u,\beta} \vee v' \in S_{v,\beta}) \vee (u' \in S_{v,\beta} \vee v' \in S_{u,\beta}). \tag{6}$$

The process of recovering similar edges can be interpreted as a **Vertex Cover** problem and we can derive an equivalent formulation of the condition 6 stated above as follows.

$$(u' \in S_{u,\beta} \cup S_{v,\beta}) \vee (v' \in S_{u,\beta} \cup \in S_{v,\beta}). \tag{7}$$

[3]In the original pGRASS [19], [20] paper, the algorithm partitions the spanning tree so it can run Tarjan's offline least common ancestor (LCA) algorithm [23]. In this paper, we dynamically compute the LCA, so we do not need to partition the spanning tree.

The size of the resulting subgraph in the feGRASS algorithm is constrained by the input parameter α, determining the number of recovered edges as $\alpha|V|$. The generated subgraph contains $|V| - 1 + \alpha|V|$ edges in total. If feGRASS fails to recover $\alpha|V|$ edges after one pass, the algorithm will repeat the off-tree-edge recovery process on the remaining off-tree edges until it recovers $\alpha|V|$ edges.

C. Parallelizing the feGRASS Algorithm

Subsequent work introduced pGRASS [19], a parallel algorithm based on feGRASS, which focuses on parallelizing the off-tree edge recovery step because the other steps have standard parallel solutions. First, the algorithm computes the effective resistance of every edge in the original graph using the spanning tree, which can be parallelized using divide-and-conquer to divide the spanning tree into subtrees [19] and applying Tarjan's offline LCA algorithm [23].

At a high level, pGRASS parallelizes the seemingly-sequential edge-similarity checks by dividing the off-tree edges upfront into sequential blocks such that parallel threads process edges within each block in parallel. Note that the algorithm may perform *excess work* because it may process edges that are similar to earlier edges in the block in parallel. After all edges in a block have been processed, pGRASS performs a serial pass through all edges in the block to check whether they should truly be recovered, or if they should have been skipped due to parallel edges within the same block.

D. Analysis method

In this paper, we analyze parallel algorithms in the ***work-span model*** [25, Chapter 27]. The ***work*** is the total time to execute the entire algorithm in serial. The ***span*** is the longest serial chain of dependencies in the computation. In the work-span model with binary forking, a parallel for loop with k iterations with $O(w_i)$ work and $O(S_i)$ span in the i-th iteration has $O(\sum_{i=0}^{k} w_i)$ work and $O(\log(k) + \max_{i=0}^{k} S_i)$ span.

III. Parallel Density-Aware Graph Spectral Sparsification

This section proposes ***parallel density-aware Graph Spectral Sparsification*** (pdGRASS), an efficient algorithm for GSS that resolves the drawbacks of feGRASS: 1) loose approximation and 2) redundant work during parallelization. The pdGRASS algorithm modifies the off-tree edge recovery phase to *restrict the similarity condition* to recover more off-tree edges within one pass, and *parallelize disjoint subproblems* that avoid redundant work due to data dependencies as described in Section II.

A. Density-Aware Approximation Condition

The density-aware approximation restricts the similarity condition in Definition 4 to require that *both* endpoints of potentially similar edges be contained in the separate β-hop neighborhoods of the original edge endpoints rather than

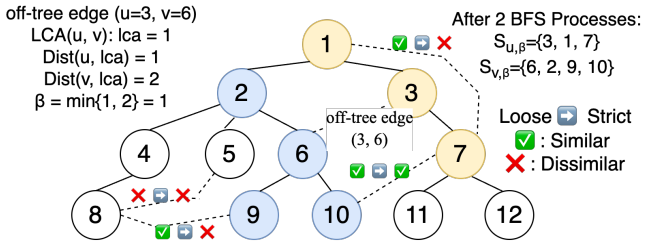

Fig. 3: An illustration of the similarity check under the loose and strict similarity conditions. In this example, the edge being checked against is $e = (3, 6)$, $\mathrm{LCA}_T(3, 6) = 1$, and $\beta = 1$. The yellow (resp., blue) vertices denote $S_{u=3,\beta=1}$ (resp., $S_{v=6,\beta=1}$). The arrow on the off-tree edges denotes whether the edge is loosely similar to e (left of the arrow) and then whether the edge is strictly similar to e (right of the arrow).

either endpoint. We further introduce a new definition for the parameter β as follows.

$$\beta^*_{u,v} = \min\{\mathrm{dist}(u, \mathrm{LCA}(u, v)), \mathrm{dist}(v, \mathrm{LCA}(u, v)), c\} \quad (8)$$

More formally, the density-aware condition replaces the *OR* condition with *AND* condition in the endpoint membership.

Definition 5 (Similiar edge (Strict))**.** *Suppose we have a spanning tree T, an off-tree edge $e = (u, v)$, and BFS step size $\beta^*_{u,v}$ and the β^*-hop vertex sets $S_{u,\beta^*}, S_{v,\beta^*}$. An off-tree edge $e' = (u', v')$ is* ***strictly similar*** *to off-tree edge e if*

$$(u' \in S_{u,\beta^*} \wedge v' \in S_{v,\beta^*}) \vee (u' \in S_{v,\beta^*} \wedge v' \in S_{u,\beta^*}). \quad (9)$$

To clearly delineate between the two conditions going forward, we will use the term ***loosely similar*** to refer to the similarity definition in feGRASS and pGRASS (Definition 4). Figure 3 presents a worked example of edge similarity under the previous loose condition and the proposed strict condition.

Under strict similarity conditions, only two edges with highly similar positions are considered similar, which enables pdGRASS to recover a sufficient number of edges in a single recovery pass. In contrast, feGRASS, which is based on vertex cover, tends to meet the similarity condition more easily, and thus can only recover a small number of edges in each pass. In extreme cases, feGRASS may recover only one edge per round. More specifically, if the constant β is greater than or equal to the diameter of the spanning tree, then recovering any single edge can cover the entire graph.

Properties of strictly-similar edges. Next, we show how the strict edge-similarity condition in Definition 5 give rise to disjoint subproblems without redundant work. The full version of the paper will contain the proofs.

Lemma 6 (Strictly similar edges share their LCA)**.** *If off-tree edge $e' = (u', v')$ is strictly similar to off-tree edge $e = (u, v)$, on a spanning tree T, then $\mathrm{LCA}(u, v) = \mathrm{LCA}(u', v')$.*

Lemma 7 (Contraposition)**.** *Given two off-tree edges $e = (u, v)$, $e' = (u', v')$ and a spanning tree T, if $\mathrm{LCA}(u, v) \neq \mathrm{LCA}(u', v')$, then e' cannot be strictly similar to e.*

Lemma 8 (Strict similarity is non-commutative)**.** *Given a spanning tree T, let $e = (u, v)$ and $e' = (u', v')$ be two strictly similar off-tree edges determined by first recovering e, performing BFS from u and v to get β-hop neighborhoods, and applying the strict similarity condition. If edge e' was recovered first, edge e may not have been marked strictly similar to e'.*

Lemma 6 proves that two strictly similar off-tree edges must share a LCA on spanning tree, which forms the basis for dividing all off-tree edges into groups based on their LCA. Then Lemma 7 (contraposition of Lemma 6) forms the basis for dividing the off-tree edges into *disjoint subtasks* in pdGRASS based on the LCA of their endpoints. If the LCAs of the endpoints of two edges do not match, they cannot possibly be strictly similar, so we can skip their similarity check. Therefore, pdGRASS groups off-tree edges based on the LCA of their endpoints. Only edges within a group need to be checked, because edges cannot be strictly similar across groups.

Lemma 8 shows that the edges in the subtasks must be processed in sorted order because strict similarity is non-commutative. In other words, any parallelization within a subtask must process edges in order. According to its definition in Equation 8, β is further constrained by the minimum between a shorter distance and a predefined constant c, so the final value of β is no greater than the one used in the theoretical proof. Hence, the introduction of the constant c does not affect the validity of the previously established properties.

Due to space limitations, we sketch some of the proofs in this section, but will include all proofs in the full version.

IV. Parallelizing pdGRASS

This section will 1) show how the properties in Section III give rise to disjoint subtasks and 2) discuss several parallelization strategies of the subtasks depending on their size. Then we analyze the work and span of the proposed pdGRASS algorithm using the analysis model from Section II.

A. Parallelization Strategies

First, we propose several subtask parallelization strategies based on the data dependencies shown in Lemmas 7 and 8 and distribution of the size of the subtasks.

Outer parallelization. The most straightforward parallelization method is ***outer parallelization***, or parallelization across the subtasks. By Lemma 7, all of these subtasks can be processed independently because there are no strictly similar edges across subtasks. The similarity checks are embarassingly parallel across subtasks, so there is no need for additional complexity.

Inner parallelization. If some subtasks are much larger than others, which is likely due to the skewed nature of graphs, inter-task parallelization alone may not be sufficient to achieve good parallel speedup. Therefore, we can also perform ***inner parallelization*** within tasks using the same blocked method as pGRASS described in Section II. In the initial pGRASS parallelization method, the list of off-tree edges is divided into blocks upfront, and edges in each block are processed

in parallel, but the blocks themselves are serialized. Some of the edges in a block may have been marked during a previous block, so the amount of edges that need to be processed in subsequent blocks depends on previous blocks.

During each edge recovery, the algorithm first checks whether the current edge has already been marked. If so, it enters the continue branch and skips further processing. When a set of edges is processed in parallel, threads that enter the continue branch may lead to execution bubbles. We propose an optimization method, ***Judge-before-Parallel***, that extracts the if-condition checks outside the parallel region, thereby eliminating conditional branching and ensuring full thread utilization without idle threads.

Heuristic-based mixed parallel strategy. In pdGRASS, we always apply outer-task parallelism across the independent subtasks, but use heuristics to determine whether to apply inner-task parallelism in a given subtask based on whether the subtasks is sufficiently large (i.e., it has many edges or covers over 10% of total edges). We found that in practice, applying inner-task parallelism in subtasks with at least 1E5 edges resulted in good parallel speedup. Outer parallelism may encounter issues with uneven thread load. When a very large subtask is assigned to a particular thread, that thread may still be executing even after all other subtasks have finished, which results in a decline in overall performance. Therefore, we need to apply an inner parallel strategy for such large tasks.

B. Algorithm Description and Analysis

Finally, we describe pdGRASS and analyze its work and span in the binary-forking model described in Section II. The full version will contain the proofs.

At a high level, pdGRASS applies the strict edge-similarity condition and divides the edges into disjoint ***subtasks*** according to the LCA of their endpoints. From Lemma 6, if two edges are not strictly similar, they have different LCAs. Edges must be processed sequentially within a subtask, according to Lemma 8, but pdGRASS can apply inner parallel strategy of dividing the edges into blocks, as described in Section II.

There are four main steps in pdGRASS: 1) computing the resistance distance for each off-tree edge, 2) sorting the off-tree edges by their resistance distance, 3) creating subtasks based on shared LCAs and sorting them based on their size, and 4) recovering edges via the strict edge-similarity condition within mixed parallel strategy on subtasks. Table I summarizes the work and span bounds for each of these steps.

Step	Work	Span
1	$O(\|E\| \lg \|V\|))$	$O(\lg^2 \|V\|)$
2	$O(\|E\| \lg \|E\|)$	$O(\lg^2 \|E\|)$
3	$O(\|E\| \lg \|E\|)$	$O(\lg^2 \|E\|)$
4	$O(\sum_{i=0}^{k} \|S_i\|^2)$	$O(\|S_o\|^2/p + \|S_{\text{serial}}\|^2)$
Total	$O(\|E\| \lg \|E\| + \sum_{i=0}^{k} \|S_i\|^2)$	$O(\lg^2 \|E\| + \|S_o\|^2/p + \|S_{\text{serial}}\|^2)$

p: number of parallel threads. k: number of subtasks.
S_i: set of edges in i-th subtask. S_{serial}: the largest sequential subtask.

TABLE I: Work-Span analysis of each step in pdGRASS.

V. Evaluation

This section empirically evaluates pdGRASS compared to feGRASS in terms of their 1) recovery runtime and 2) sparsifier quality, in terms of the iteration count required for convergence when using the sparsifier in a downstream application of preconditioned conjugate gradient (PCG) [21]. Then we evaluate the strong scaling of pdGRASS on representative cases of skewed and uniform subtask distributions to evaluate the impact of different parallelization strategies. Experiments show that the mixed parallel strategy consistently achieves effective parallel scaling across diverse data distributions.

Setup. We performed experiments on an Rocky Linux (8.9, Green Obsidian) server with AMD EPYC 7T83 64-Core Processor. Each core has 32 KB L1d/L1i and 512 KB L2 caches, with a 32 MB L3 cache shared per socket. We implemented pdGRASS in `C++17` with `OpenMP 4.5` [26] and compiled by `GCC 8.5.0`. We compare with the open-source[4] implementation of feGRASS. To do an apples-to-apples comparison of the recovery step, pdGRASS uses the same spanning tree as feGRASS. We report the minimum *runtime* over 5 trials of the edge recovery step.

The parameter α defines the ratio of edges to be recovered. We evaluate feGRASS and pdGRASS with α set to 0.02, 0.05, and 0.10 to assess performance across different recovery ratios.

We measure sparsifier *quality* by the iteration count required for convergence in a PCG solver from MATLAB R2023b to evaluate GSS performance. Specifically, given a subgraph P of the original graph G, the PCG solver uses L_P as the preconditioner to solve $||L_G x - b|| \leq 10^{-3}||b||$ iteratively. A lower iteration count indicates a higher-quality sparsifier.

Datasets. We evaluate the algorithms on a suite of 18 graphs from the SuiteSparse Matrix Collection [27], including datasets from SNAP [28] and DIMACS10 [29]. We select symmetric matrices representing undirected graphs with a single connected component. For graphs without edge weights, we assign random positive weights uniformly sampled between 1 and 10. Table II summarizes the graph sizes.

pdGRASS parameters. Given p threads, we set the block size (as defined in Section II) to p, enabling p edges to be processed in parallel. This design follows the *Judge-Before-Parallel* optimization to maximize thread occupancy. pdGRASS first handles large tasks using inner parallelism in a one-by-one manner and then applies outer parallelism to the remaining tasks. The cutoff between inner and outer is defined as 1E5, or 10% of the total off-tree edges, effectively isolating larger subtasks for improved performance. As for α, we choose $0.02, 0.05, 0.10$ and report the results in Table II. The default in previous work [7] is $\alpha = 0.02$, but we show that pdGRASS with increasing α can efficiently compute higher-quality subgraphs than with smaller α.

[4]https://github.com/5Mrzhao/CSP/blob/main/Sfegrass.cpp

Graph	$\|V\|$	$\|E\|$	$\alpha = 0.02$						$\alpha = 0.05$						$\alpha = 0.10$					
			T_{fe}	Pass	$iter_{fe}$	T_{pd-32}	$iter_{pd}$	$\frac{iter_{fe}}{iter_{pd}}$	T_{fe}	Pass	$iter_{fe}$	T_{pd-32}	$iter_{pd}$	$\frac{iter_{fe}}{iter_{pd}}$	$T_{fe}(ms)$	pass	$iter_{fe}$	T_{pd-32}	$iter_{pd}$	$\frac{iter_{fe}}{iter_{pd}}$
01-*mi2010*	3.30E5	7.89E5	82	1	**83**	**3**	93	0.9	116	3	66	**6**	**35**	1.9	180	6	57	**12**	**24**	2.4
02-*mo2010*	3.44E5	8.28E5	88	1	**84**	**4**	93	0.9	130	3	65	**6**	**34**	1.9	202	6	57	**13**	**25**	2.3
03-*oh2010*	3.65E5	8.84E5	92	1	**81**	**3**	92	0.9	134	3	65	**8**	**34**	1.9	210	6	53	**16**	**25**	2.1
04-*pa2010*	4.22E5	1.03E6	81	1	**83**	**7**	93	0.9	117	3	66	**11**	**34**	1.9	182	6	58	**20**	**25**	2.3
05-*il2010*	4.52E5	1.08E6	115	1	**81**	**11**	99	0.8	170	3	68	**13**	**56**	1.2	270	5	55	**24**	**26**	2.1
06-*tx2010*	9.14E5	2.23E6	276	1	**87**	**24**	97	0.9	471	3	72	**47**	**34**	2.1	836	6	39	**96**	**27**	1.4
07-*com-DBLP*	3.17E5	1.05E6	252	2	**134**	**118**	135	1.0	475	6	131	**385**	**121**	1.1	**897**	14	124	1139	**105**	1.2
08-*com-Amazon*	3.35E5	9.26E5	130	2	**72**	128	82	0.9	**208**	4	61	347	**60**	1.0	**334**	7	54	635	**49**	1.1
09-*com-Youtube*∗	1.13E6	2.99E6	-	6931	199	**1353**	**190**	1.0	-	38308	224	**8544**	**127**	1.8	-	103081	161	**23062**	**93**	1.7
10-*coAuthorsCiteseer*	2.27E5	8.14E5	192	2	175	**53**	**111**	1.6	341	5	160	**160**	**101**	1.6	582	10	128	**387**	**93**	1.4
11-*citationsCiteseer*	2.68E5	1.16E6	514	4	131	**115**	**126**	1.0	1245	15	**94**	**338**	113	0.9	3021	52	104	**791**	**70**	1.5
12-*coAuthorsDBLP*	2.99E5	9.78E5	248	3	**86**	**105**	89	1.0	477	6	83	**342**	**74**	1.1	883	14	80	889	**62**	1.3
13-*coPapersDBLP*	5.40E5	1.52E7	10770	2	**200**	**105**	234	0.9	22813	6	**206**	**420**	215	1.0	42865	13	202	**3703**	**192**	1.1
14-*NACA0015*	1.04E6	3.11E6	352	1	**98**	**26**	234	0.4	519	2	**83**	**49**	85	1.0	834	4	66	**92**	**30**	2.2
15-*M6*	3.50E6	1.05E7	1500	1	**105**	**164**	106	1.0	2192	2	**89**	**372**	91	1.0	3520	4	66	**786**	**31**	2.1
16-*333SP*	3.71E6	1.11E7	1550	1	**107**	**222**	187	0.6	2218	2	88	**419**	**85**	1.0	3489	4	69	**865**	**30**	2.3
17-*AS365*	3.80E6	1.14E7	1629	1	**105**	**242**	182	0.6	2390	2	**87**	**437**	91	1.0	3863	4	70	**906**	**30**	2.3
18-*NLR*	4.16E6	1.25E7	1818	1	**108**	**221**	193	0.6	2674	2	**89**	**502**	90	1.0	4256	4	71	**1052**	**30**	2.4

∗Time out for feGRASS on graph *com-Youtube*, for $\alpha = 0.02, 0.05$, feGRASS runs over 10 minutes; for $\alpha = 0.10$, feGRASS runs over 1 hour.

TABLE II: Evaluation of runtime and subgraph quality. T_{fe} (ms) and T_{pd-32} (ms) represent the execution times of feGRASS and pdGRASS (with 32 threads), respectively. $Pass$ represents the number of passes required by feGRASS to restore a sufficient number of edges. The $Pass$ for pdGRASS is omitted because it always completes in one single pass. $iter_{fe}$ and $iter_{pd}$ represents the number of iterations for the PCG solver to converge when using the generated sparsifier as the preconditioner.

Runtime. On 32 threads, pdGRASS shows a significant runtime advantage over feGRASS. Excluding the *com-youtube* dataset due to a timeout, pdGRASS achieves average speedups of $8.76\times$, $6.32\times$, and $3.94\times$ over feGRASS under 32-thread execution, for $\alpha = 0.02$, 0.05, and 0.10, respectively. These results align with our algorithmic analysis: as α increases and more edges are recovered, the problem size grows, causing the runtime of pdGRASS with single thread to increase faster than that of feGRASS due to its inherently quadratic complexity, whereas feGRASS operates in linear time.

As discussed in Section I, pdGRASS addresses the worst-case runtimes encountered by feGRASS on challenging inputs. Although feGRASS has lower theoretical complexity, its vertex-cover-based loose similarity condition leads to many edges being marked as similar, often requiring multiple passes to meet recovery targets. *com-Youtube*, a highly skewed graph where a few high-degree vertices connect to many others, exemplifies this issue. Once a high-degree vertex is covered, nearly all incident edges are marked as similar, severely limiting the number of edges recoverable per pass. Even with $\alpha = 0.02$, feGRASS requires over 6000 passes to recover enough edges. While each pass runs in linear time, the cumulative overhead results in significant performance degradation. Excluding *com-youtube*, feGRASS still requires an average of 1.6, 4.1, and 9.7 passes to recover 2%, 5%, and 10% of edges, respectively. In contrast, pdGRASS 's stricter similarity condition allows substantially more edges to be recovered per pass.

Quality. As shown in Table II, pdGRASS yields lower PCG iteration counts than feGRASS on most datasets as α increases, indicating improved subgraph quality. The iteration ratio between feGRASS and pdGRASS rises from $0.9\times$ at $\alpha = 0.02$ to $1.3\times$ at $\alpha = 0.05$, and $1.8\times$ at $\alpha = 0.10$, suggesting that pdGRASS benefits more from increased edge recovery. At $\alpha = 0.10$, pdGRASS consistently achieves convergence with roughly half the PCG iterations required by feGRASS. Although $\alpha = 0.02$ is commonly used in feGRASS as a heuristic to balance quality and runtime, pdGRASS 's higher efficiency enables the use of larger α values to produce higher-quality subgraphs without significant overhead.

The strict similarity condition in pdGRASS allows more spectrally critical edges to be recovered, leading to higher-quality subgraphs. In contrast, feGRASS employs a looser condition, which often causes some spectrally important edges to be marked as similar and excluded from recovery once their endpoints are covered.

Strong Scalability. We conducted a comparison between feGRASS and pdGRASS by evaluating their performance ratios on 1, 8, and 32 threads. As the number of threads increases to 32, pdGRASS consistently surpasses feGRASS in performance across all datasets, with an average speedup of $8.8\times$. Notably, pdGRASS delivers over $20\times$ speedup on 5 out of 18 tested datasets. Furthermore, pdGRASS scales efficiently, achieving an average speedup of $5.8\times$ on 8 threads and $11.3\times$ on 32 threads. On skewed inputs, the inner parallel region dominates execution time due to extreme load imbalance, while the outer region contributes minimally and quickly saturates in speedup. A mixed parallel strategy is essential in this case to balance the workload and improve scalability. On more uniform inputs, even task distribution enables efficient parallelism, resulting in near-ideal strong scaling across threads. More detailed tables will be included in the full version.

VI. Conclusion

We present pdGRASS, an efficient parallel Graph Spectral Sparsification (GSS) algorithm generating density-aware subgraphs. pdGRASS improves the state-of-the-art feGRASS by 1) restricting edge-similarity conditions to recover more edges per pass, and 2) parallelizing over the resulting independent subtasks. Additional technical details and extended results are provided in the full version of this paper.[5]

[5]The full version is available at https://arxiv.org/abs/2508.20403.

REFERENCES

[1] M. B. Cohen, R. Kyng, G. L. Miller, J. W. Pachocki, R. Peng, A. B. Rao, and S. C. Xu, "Solving sdd linear systems in nearly mlog1/2n time," in *STOC*, 2014, p. 343–352.

[2] I. Koutis, G. L. Miller, and R. Peng, "Approaching optimality for solving sdd linear systems," in *FOCS*, 2010, pp. 235–244.

[3] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1913–1926, 2011.

[4] D. A. Spielman and S.-H. Teng, "Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems," *SIAM Journal on Matrix Analysis and Applications*, vol. 35, no. 3, pp. 835–885, 2014.

[5] Z. Feng, "Spectral graph sparsification in nearly-linear time leveraging efficient spectral perturbation analysis," in *DAC*, 2016.

[6] ——, "Grass: Graph spectral sparsification leveraging scalable spectral perturbation analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4944–4957, 2020.

[7] Z. Liu, W. Yu, and Z. Feng, "fegrass: Fast and effective graph spectral sparsification for scalable power grid analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 681–694, 2021.

[8] Y. Zhang, Z. Zhao, and Z. Feng, "Sf-grass: solver-free graph spectral sparsification," in *ICCAD*, 2020.

[9] Y. Zhao, X. Yang, Y. Bai, L. Zeng, D. Niu, W. Liu, and Z. Jin, "Csp: Comprehensively-sparsified preconditioner for efficient nonlinear circuit simulation," in *ICCAD*, 2024.

[10] Y. Bai, X. Yang, Y. Lu, D. Niu, C. Zhuo, Z. Jin, and W. Liu, "Efficient spectral-aware power supply noise analysis for low-power design verification," in *DATE*, 2024.

[11] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng, "Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs," in *STOC*, 2011, pp. 273–282.

[12] I. Koutis, G. L. Miller, and R. Peng, "Approaching optimality for solving sdd linear systems," *SIAM Journal on Computing*, vol. 43, no. 1, pp. 337–354, 2014.

[13] D. A. Spielman and S.-H. Teng, "Spectral sparsification of graphs," *SIAM Journal on Computing*, vol. 40, no. 4, pp. 981–1025, 2011.

[14] R. Peng, H. Sun, and L. Zanetti, "Partitioning well-clustered graphs: Spectral clustering works!" vol. 46, no. 2. USA: Society for Industrial and Applied Mathematics, Jan 2017, p. 710–743.

[15] S.-H. Teng, *Scalable Algorithms for Data and Network Analysis*, 2016.

[16] Z. Zhao, Y. Wang, and Z. Feng, "Samg: Sparsified graph-theoretic algebraic multigrid for solving large symmetric diagonally dominant (sdd) matrices," in *ICCAD*, 2017, pp. 601–606.

[17] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *NIPS*, 2016.

[18] Z. Zhao and Z. Feng, "A spectral graph sparsification approach to scalable vectorless power grid integrity verification," in *DAC*, 2017.

[19] Z. Liu and W. Yu, "pgrass-solver: a parallel iterative solver for scalable power grid analysis based on graph spectral sparsification," in *ICCAD*, 2021.

[20] ——, "pgrass-solver: A graph spectral sparsification-based parallel iterative solver for large-scale power grid analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 9, pp. 3031–3044, 2023.

[21] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[22] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, "Spectral sparsification of graphs: theory and algorithms," *Communications of the ACM*, vol. 56, no. 8, pp. 87–94, 2013.

[23] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," in *STOC*, 1983, p. 246–251.

[24] D. Babić, D. J. Klein, I. Lukovits, S. Nikolić, and N. Trinajstić, "Resistance-distance matrix: a computational algorithm and its application," *International Journal of Quantum Chemistry*, vol. 90, no. 1, pp. 166–176, 2002.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[26] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," May 2008. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[27] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec 2011.

[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[29] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner, *Benchmarking for Graph Clustering and Partitioning. In Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.