The ParClusterers Benchmark Suite (PCBS): A Fine-Grained Analysis of Scalable Graph Clustering [Experiment, Analysis & Benchmark]

Shangdi Yu
MIT
shangdiy@mit.edu

Jessica Shi
MIT
jessicashi@gmail.com

Jamison Meindl
MIT
jmeindl@mit.edu

David Eisenstat
Google
eisen@google.com

Xiaoen Ju
Google
xiaoen@google.com

Sasan Tavakkol
Google
tavakkol@google.com

Laxman Dhulipala
UMD
laxman@umd.edu

Jakub Łącki
Google
jlacki@google.com

Vahab Mirrokni
Google
mirrokni@google.com

Julian Shun
MIT
jshun@mit.edu

ABSTRACT

We introduce the ParClusterers Benchmark Suite (PCBS)—a collection of highly scalable parallel graph clustering algorithms and benchmarking tools that streamline comparing different graph clustering algorithms and implementations. The benchmark includes clustering algorithms that target a wide range of modern clustering use cases, including community detection, classification, and dense subgraph mining. The benchmark toolkit makes it easy to run and evaluate multiple instances of different clustering algorithms, which can be useful for fine-tuning the performance of clustering on a given task, and for comparing different clustering algorithms based on different metrics of interest, including clustering quality and running time.

Using the benchmark suite, we evaluate a broad collection of real-world graph clustering datasets. Somewhat surprisingly, we find that correlation clustering consistently performs the best in our experiments across both unweighted and weighted graph clustering. The PCBS provides a standardized way to evaluate and judge the quality-performance tradeoffs of the active research area of scalable graph clustering algorithms. We believe it will help enable fair, accurate, and nuanced evaluation of graph clustering algorithms in the future.

PVLDB Reference Format:
Shangdi Yu, Jessica Shi, Jamison Meindl, David Eisenstat, Xiaoen Ju, Sasan Tavakkol, Laxman Dhulipala, Jakub Łącki, Vahab Mirrokni, and Julian Shun. The ParClusterers Benchmark Suite (PCBS): A Fine-Grained Analysis of Scalable Graph Clustering [Experiment, Analysis & Benchmark]. PVLDB, 17(1): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ParAlg/ParClusterers.

Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Clustering is a critical tool in almost any scientific field that involves classifying and organizing data today. Examples of fields leveraging clustering range from computational biology and phylogenetics to complex network analysis, machine learning, and astrophysics [19, 58, 71, 74]. Clustering has proven particularly useful in fields transformed by AI and machine learning because of its utility in understanding and leveraging high-dimensional vector representations (embeddings) of data [12, 28, 29, 33, 63].

In this paper, we are interested in understanding the behavior of parallel clustering algorithms for shared-memory multi-core machines that are scalable in the size of the dataset and the number of threads. Our focus is on *graph clustering*, which is a versatile clustering approach that can be used with different input types.

On one hand, graph clustering is a natural approach whenever the input is modeled as a graph (e.g., friendships, interactions, etc.). On the other hand, graph clustering can also be applied in the other popular scenario, when the input is a collection of points in a metric space. In this case, one can obtain a graph representation of the input by computing a *weighted similarity graph*, where continuous or complete phenomena can be cast into sparse similarity graphs, e.g., by keeping only edges between nearby points or only the most significant entries of a similarity matrix.

Despite substantial prior work studying the quality (e.g., precision and recall) and scalability of individual graph clustering methods [28, 29, 31, 75, 80, 84, 86], no prior works have systematically compared different graph clustering methods to understand how different methods compare against each other under different metrics. For example, celebrated and widely-utilized graph clustering algorithms, such as modularity clustering are well understood to be highly effective in community detection tasks on unweighted natural graphs, but little is known about their performance for clustering on classification tasks.

In this paper, we address this gap by performing a systematic and thorough comparison of a diverse set of graph clustering methods. Our evaluation includes methods tailored to both weighted and unweighted graphs and incorporates a diverse set of natural graphs and graphs derived from pointsets. We also stratify our evaluation based on three distinct unsupervised clustering tasks that are commonly found in the literature and in practice—(1) community detection, (2) unsupervised classification, and (3) dense subgraph mining. Due to insisting on scalability, we focus our evaluation

on the most scalable parallel graph clustering methods currently available in the literature.

To make our evaluation easily reusable and extensible by future researchers, we designed a graph clustering benchmark called the **ParClusterers Benchmark Suite (PCBS)**. PCBS enables users to accurately measure the scalability and accuracy of different shared-memory parallel graph clustering algorithms. In addition to providing a simple and easy to use benchmarking platform, we have also incorporated eleven parallel graph clustering methods into PCBS. The algorithms include algorithms from our recent prior work, as well as several new implementations. In addition to classic graph clustering methods such as modularity-based clustering [75], structural clustering [85], and label propagation methods [72], we include recently developed hierarchical agglomerative graph clustering methods [29] and connectivity-based methods such as k-core and low-diameter decomposition [27]. Finally, unlike much of the existing work on graph clustering, which usually focus on specific graph clustering metrics (e.g., modularity or conductance) that a proposed method is usually designed to optimize, PCBS supports evaluation on a very broad set of metrics, which helps us understand *what* different clustering algorithms are able to optimize for on real-world datasets, and help inform users of the best clustering algorithm for a given metric. Besides PCBS's clustering implementations, PCBS also supports running many clustering implementations in other graph clustering frameworks and systems such as NetworKit [80], Neo4j [5], and TigerGraph [7].

The datasets we study include both widely-used graph datasets from the SNAP repository, as well as several new graph clustering datasets that we have generated from spatial and embedding datasets using a simple nearest-neighbor-based graph building process, and which we will open-source as part of this work. Our datasets cover a wide range of scales and clustering tasks, including community detection, classification, and dense subgraph clustering.

Key Contributions. The key contributions of our work include:

- A comprehensive library that implements eleven state-of-the-art scalable graph clustering algorithms, providing a unified codebase for researchers and practitioners.
- A benchmarking *toolkit* that facilitates the systematic evaluation of graph clustering algorithms across diverse datasets, parameter settings, and experimental configurations, enabling rigorous and comprehensive comparative analyses.
- The first extensive evaluation of parallel graph clustering algorithms, encompassing their runtime performance, clustering quality, and the trade-off between these two critical dimensions. We also compare our library against other existing libraries and graph databases.

Key Results. Some of our key takeaways and findings of our study of graph clustering algorithms include:

- Our clustering implementations in PCBS are very fast compared to other clustering implementation in state-of-the-art graph libraries and databases. While graph databases, such as Neo4j [5] and TigerGraph [7], provide a richer functionality, on different graph clustering implementations, they are slower. For example, on the LiveJournal graph from SNAP [49], PCBS is on average 32.5x faster than Neo4j and 303x faster than TigerGraph. Compared with state-of-the-art parallel graph library NetworKit [80], PCBS is on average 4.54x faster. We compute the average using the geometric mean.
- Correlation clustering consistently obtains the highest quality in our experiments compared to other clustering algorithms that we benchmarked across the three tasks that we studied.
- Approximate parallel graph hierarchical agglomerative clustering (ParHAC) [29] also achieves relatively high quality. While a single run of ParHAC may be slower than other methods, it offers the advantage of being a hierarchical approach. This means that in a single run, it can generate clusterings at multiple levels of granularity. In contrast, methods like correlation and modularity clustering can only produce a clustering of a single granularity in a single run.
- Parallel affinity clustering obtains high quality on the classification task and is consistently faster than correlation clustering and ParHAC on large graphs.

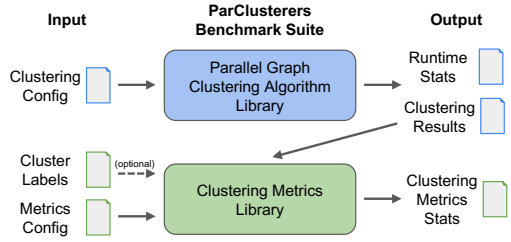

Figure 1: Overview of our PCBS library.

2 PARCLUSTERERS BENCHMARK SUITE

In this section, we introduce our PCBS benchmark suite, which includes a large number of scalable parallel graph clustering algorithms, parallel implementations of clustering evaluation metrics, efficient graph I/O routines, and a convenient benchmark interface. An overview of the library is shown in Figure 1.

2.1 Graph Input

PCBS supports both **weighted and unweighted simple undirected graphs** in edge list format [49] and compressed sparse row format [27, 40]. A simple graph is a graph with no self-loops and no parallel edges. The graph clustering algorithms in PCBS that use edge weights assume edge weights are nonnegative **similarities**, i.e., a higher weight of an edge (u, v) means that u and v are more similar. This allows for a natural way of defining how a missing edge affects the clustering: in most of our algorithms a missing edge is equivalent to an edge of similarity 0.

2.2 Datasets

In this work, we benchmark on both unweighted and weighted graphs. For unweighted graphs, the SNAP [49] collection of datasets is a popular choice for benchmarking community detection tasks. To the best of the authors' knowledge, there currently exists no real-world *weighted* graph dataset with ground truth for evaluating weighted graph clustering algorithms. In this work, we present a suite of weighted graph clustering datasets that are the k-nearest neighbor graphs of real-world vector datasets with ground truth clustering labels. We describe more details of constructing these

datasets in Section 3. We believe that these datasets will allow researchers to thoroughly benchmark the performance of weighted graph clustering algorithms.

2.3 Parallel Graph Clustering Algorithms

Our library includes eleven scalable parallel graph clustering algorithms, which are described below. We focus on algorithms that are scalable, widely used, and have high precision (i.e., most vertices in a retrieved cluster are from the same ground truth cluster). We discuss more about our algorithm selection in Section 2.4. We select a mix of algorithms that are popular in the literature and algorithms that we found to have high quality in our experience. The implementations of TECTONIC, label propagation, and speaker-listener label propagation (SLPA) are new. Other implementations are taken from previous work and integrated into PCBS. All implementations, except for SLPA, produce non-overlapping clusters.

We categorize the algorithms into *weighted graph algorithms*, which take into account the similarities, and *unweighted graph algorithms*, which only use the topology of the graph.

2.3.1 Weighted Graph Algorithms. **Affinity Clustering [12, 28].** Affinity clustering is a hierarchical clustering algorithm based on Boruvka's minimum spanning forest algorithm. PCBS includes the shared-memory parallel affinity clustering from Dhulipala et al. [28], which is adapted from the original MapReduce affinity clustering algorithm [12]. In each step of the algorithm, every vertex picks its best edge (i.e., that of the maximum similarity) on each round, and the connected components spanned by the selected edges define the clusters. The algorithm can produce a clustering hierarchy by contracting the clusters and running the algorithm recursively. The edge weights between contracted vertices can be computed using different linkage functions, such as single, maximum, or average linkage. In our experiments we use average linkage.

Users can control the resolution of the clustering by picking an edge weight threshold and only considering edges whose weight is at least the threshold in each step. Recent work by Monath et al. [63] showed that the quality of affinity clustering can be improved if the threshold decays geometrically at each step. We use this technique, also known as the SCC algorithm [63], in our experiments.

Correlation and Modularity Clustering [75]. Our library includes a shared-memory parallel framework for optimizing the LambdaCC objective [87]. This objective generalizes both modularity [66] and correlation clustering [11] and is defined as follows.

Let k_v be the vertex weight of v and λ be the resolution parameter. Let V be the set of all vertices, and w_{uv} be the weight (similarity) of edge (u, v). Then,

$$\texttt{LambdaCC}(x) = \sum_{(i,j)\in V\times V} w'_{ij} \cdot x_{ij},$$

where x_{ij} is a Boolean indicator, which is equal to 1 iff i and j belong to the same cluster. Here, $w'_{uv} = 0$ if $u = v$, $w'_{uv} = w_{uv} - \lambda k_u k_v$ if $(u, v) \in E$, and $w'_{uv} = -\lambda k_u k_v$ otherwise. By default our framework uses $k_u = 1$ for any $u \in V$.

As shown in [75], the special case of the modularity objective can be obtained by appropriately defining vertex weights k and setting λ. Specifically, assume we set the vertex weight k_v to be equal to the weighted degree of v (i.e., the sum of its incident edge weight) and set the resolution $\lambda = 1/(2m)$. Then, optimizing LambdaCC objective is equivalent to optimizing the modularity objective [37], as the two objectives are monotone in each other. Furthermore, if we set the resolution $\lambda = \gamma/(2m)$, the LambdaCC objective becomes monotone in the generalized modularity objective of Reichardt and Bornholdt [73] with a fixed scaling parameter $\gamma \in (0, 1)$.

Our framework optimizes the LambdaCC objective using a parallel Louvain-style algorithm [17, 75].

Approximate Parallel Hierarchical Agglomerative Clustering (ParHAC) [29]. Given n vertices, the HAC algorithm starts by forming a separate cluster for each vertex, and proceeds in $n - 1$ steps. Each step replaces merges the two most similar clusters, i.e., replaces them by their union. The similarity of two clusters is the total weight of edges between the clusters divided by the product of their sizes.

The output of HAC is a binary tree called a dendrogram, which describes the merges performed by the algorithm. A flat clustering can be obtained from the dendrogram by cutting it at a given level. The threshold for cutting the dendrogram controls the clustering resolution. Because the output of HAC is a dendrogram, one can easily postprocess the dendrogram to obtain a flat clustering of any given resolution. The running time of this postprocessing is negligible compared to the clustering time. PCBS includes a shared memory parallel implementation of *approximate* HAC called ParHAC [29]. A HAC algorithm is called $1 + \epsilon$ approximate, when each merged pair of clusters has similarity at least $W_{\max}/(1 + \epsilon)$, where $W_{\max}$ is the largest similarity between any two clusters.

Connected Components. [27, 44] Given a threshold parameter τ, the clusters are the connected components with edge similarity $< \tau$ removed. Our implementation uses the state-of-the-art implementation of concurrent union-find capable of processing several billions of edges per second from the ConnectIt library [30, 44] to find the connected components.

2.3.2 Unweighted Graph Algorithms. **Low-Diameter Decomposition (LDD) [27, 62].** LDD partitions vertices of an n-vertex, m-edge unweighted graph into clusters, where each cluster has $O((\log n)/\beta)$ diameter and there are at most βm edges connecting distinct clusters. The choice of β controls the clustering resolution. Our implementation is based on the parallel MPX algorithm [62].

k-core Decomposition (KCore) [26, 27]. k-core decomposition is a graph clustering technique that recursively prunes vertices from the graph whose degree is less than k, until all remaining vertices have at least k neighbors. This process partitions the graph into k-cores, where each k-core is a maximal subgraph in which every vertex has degree at least k within that subgraph. Non-empty k-cores for large value of k tend to represent a densely connected core of the graph, while k-cores for small values of k represent more peripheral regions. For a given value of k, k-core decomposition identifies clusters by treating each maximal k-core subgraph as a cluster and treating all other vertices as singleton clusters. The choice of k can be used to control the clustering resolution.

The algorithm outputs *connected components* of vertices with core number at most k. Vertices with core number less than k are in their own singleton clusters.

Structural Graph Clustering (SCAN) [27, 85]. This implementation of SCAN is a parallel version [85] of the index-based SCAN algorithm introduced in Wen et al. [88]. Following [91], our implementation of SCAN defines a structural similarity σ between adjacent vertices u and v as follows:

$$\sigma(u,v) = \frac{|\overline{N}(u) \cap \overline{N}(v)|}{\sqrt{|\overline{N}(u)|}\sqrt{|\overline{N}(v)|}}.$$

Here, $\overline{N}(u)$ is a set including u and the neighbors of u.

Two vertices are structurally similar if their structural similarity is at least ϵ, an input parameter. A vertex is called a *core* vertex, if it is structurally similar to at least μ other vertices, where μ is another parameter. A cluster is constructed by a core expanding to all vertices that are structurally similar to that core.

Specifically, the output of the algorithm is equivalent to the following algorithm. First, construct an auxiliary graph on the core points, in which we add an edge between any two core vertices, which are structurally similar. Then, compute the connected components of this graph. Finally, assign each non-core point which is structurally similar to a core point to the cluster of that core point (in case there is more than one structurally similar core point, one is chosen arbitrarily). The remaining points are left as singleton clusters.

Triangle Connected Component Clustering (TECTONIC). PCBS contains a parallel version of the original TECTONIC clustering algorithm [86]. The algorithm first weighs the graph edges based on the number of triangles each edge belongs to. Specifically, if $t(u,v)$ is the number of triangles that edge (u,v) participates in, the weight of an edge (u,v) is set to $\frac{t(u,v)}{\deg(u)+\deg(v)}$. Then, it removes all edges with weight below a threshold parameter θ, where θ controls the clustering resolution, and computes connected components.

Label Propagation (LP). PCBS contains a parallel version of the original label propagation algorithm [72]. The label propagation algorithm works by propagating vertex labels through the graph. Initially, each vertex is assigned a unique label. Then in each iteration, vertices adopt the label that the majority of their neighbors currently have, with ties broken lexicographically. This causes label updates to propagate across the graph, with dense regions quickly reaching consensus on a label. All vertices start in their own singleton cluster. The algorithm runs until no vertex updates its label or a maximum number of iterations has been reached. The groups of vertices converging to the same label represent the clusters. In our parallel implementation, on each round, all vertices asynchronously update their labels in parallel. Specifically, on each round, a vertex v's neighbor's label might be updated from l to l' after v has already used the label l to update its own label.

Speaker-Listener Label Propagation (SLPA). PCBS contains a parallel version of the original SLPA [90] algorithm. SLPA is a variant of the label propagation algorithm, where each vertex maintains a list of labels (its memory) instead of a single label. On each round, in parallel every vertex passes a random label to each neighbor, chosen from its memory with probability proportional to the occurrence frequency of this label in the memory (the choice can be different for each neighbor). Whenever a vertex receives a new label from a neighbor, it immediately updates its own memory, so the implementation is asynchronous. SLPA may produce overlapping and nested clusters.

2.4 Scope

Due to the long history of research on clustering algorithms, hundreds of clustering algorithms have been developed over the past century, and exhaustively comparing all existing proposals is an impossible task. Instead, we aimed to choose a *representative* collection of clustering algorithms that are (1) scalable and efficient and (2) can produce high-precision results. We now explain our rationale in more detail.

(1) Scalability and Efficiency. To keep up with the rapid growth of real-world data, a requirement for modern clustering algorithms is that they should be scalable to very large datasets, and scalable with increasing computational power (e.g., cores and machines). In this paper, we focus on the shared-memory (single-machine) multicore setting, which has been successfully used to process graphs up to hundreds of billions of edges [27].

Our focus on scalable and efficient clustering algorithms rules out algorithms such as spectral clustering and other algorithms based on numerical linear algebra, whose scalability is limited due to their poor computational efficiency.

(2) High Precision. In many applications of clustering, such as spam and abuse detection, classification, and deduplication, a usual requirement is to produce a *high precision* clustering. In other words, it is more important to ensure that all entities within a cluster are closely related, than to ensure that each group of related entities ends up in a single cluster. Because of this reason we do not include balanced graph partitioning in our comparison, which primarily optimizes recall (i.e., most vertices in each ground truth cluster are grouped into the same output cluster). We note that balanced partitioning algorithms are *not* included in many of the popular graph libraries and databases [3–5, 7, 80].

2.5 Clustering Quality Metrics

PCBS includes parallel implementations of many popular metrics for evaluating clustering quality, including precision, recall, F-score, adjusted rand index (ARI), normalized mutual information (NMI), edge density, triangle density, and more.

Precision, Recall, and F-score. PCBS computes the average precision and recall of a clustering compared to the ground truth. These metrics work for both non-overlapping and overlapping clusters. To compute average precision and recall, for each ground-truth community c, we match c to the cluster c' with the largest intersection to c. We list the formulae for computing the metrics below. This metric matches the methodology used by Tsourakakis et al. [86] in evaluating TECTONIC and Shi et al. [75] in evaluating correlation clustering.

$$\text{Precision} = |c \cap c'|/|c'|$$
$$\text{Recall} = |c \cap c'|/|c|$$
$$F_\beta = (1+\beta^2) \cdot \frac{\text{Precision} \times \text{Recall}}{(\beta^2 \times \text{Precision}) + \text{Recall}}$$

Here β is a parameter specifying the relative importance of precision and recall. When $\beta < 1$, the objective rewards optimizing precision over recall. When $\beta > 1$, optimizing recall becomes more important.

ARI and NMI. Both Adjusted Rand Index (ARI) [20, 42] and Normalized Mutual Information (NMI) [23] measure the similarity between two different clusterings of a dataset, and are widely used to evaluate the quality of non-overlapping clusters. See Hubert and Arabie [42] and Danon et al. [23] for a definition of these measures. Extensions to overlapping clusters have also been proposed (e.g., [48, 61]).

Edge and Triangle Density. These two metrics are not defined with respect to ground-truth clustering but rather measure the structural properties and density characteristics of the identified clusters themselves. Edge density quantifies the internal cohesiveness of a cluster by measuring the ratio of edges in the cluster to the maximum possible number of edges. Similarly, triangle density captures the proportion of triangle subgraphs within a cluster to the maximum possible number of triangles, given the existing wedges.

We compute a weighted mean of edge density, where the density of each cluster is weighted by its size when computing the mean. We use the weighting because taking an unweighted average of cluster densities can lead to counter-intuitive results. To illustrate this, consider a clustering with 99 clusters of size 2 and density 1, and one cluster of size 1 million and density 0.01. Then, the average density of the 100 clusters is over 0.99, even though over 99.99% of vertices are in a cluster of density 0.01.

Modularity and Correlation Objective. PCBS computes the modularity and correlation clustering objectives as described in Section 2.3.

Other Cluster Statistics. PCBS also reports the distribution (minimum, maximum, and mean) of cluster sizes, the diameter of clusters, and the number of clusters.

2.6 Benchmark

PCBS has a convenient and flexible framework for benchmarking clustering algorithms.

Listing 1 gives an example of the configuration file for benchmarking different clustering algorithms. Users can flexibly specify the graphs to benchmark, the number of threads to use, the number of rounds to run, the timeout, and the set of parameters to try for each clustering algorithm. Specifically, for each clustering algorithm, PCBS tries a Cartesian product of all parameter values. Besides clustering implementations in PCBS, PCBS also supports running many clustering implementations in NetworKit, Neo4j, and TigerGraph.

For each graph, each clusterer, each parameter set, and each round of the experiment, PCBS outputs a file for the resulting clustering in `Output directory`. It also outputs a CSV file including the running time of all runs specified by the configuration file to `CSV Output directory`.

PCBS also supports specifying the ground truth communities and computing statistics of the clustering results, such as precision and recall. Listing 2 gives an example of the statistics configuration file for computing the quality metrics. This configuration file is used together with the clustering configuration file (e.g., Listing 1).

Listing 1: Example Clustering Configuration File

```
Input directory: /input_dir/
Output directory: /output_dir/
CSV output directory: /output_dir_csv/
Clusterers: ParHacClusterer; TigerGraphLouvain
Graphs: lj.gbbs.txt; amazon.gbbs.txt
GBBS format: true
Weighted: false
Number of threads: 60
Number of rounds: 1
Timeout: 7h

ParHacClusterer:
  config:
    weight_threshold: 1.0; 0.3
    epsilon: 0.01; 0.1; 1

TigerGraphLouvain:
  config:
    maxIterations: 10; 20
```

Listing 2: Example Statistics Configuration File

```
Input communities: lj.cmty; amazon.cmty

statistics_config:
  compute_edge_density: true
  compute_precision_recall: true
  f_score_param: 0.5
```

`statistics_config` configures which metrics to compute and the parameters for the metrics. PCBS's metrics library outputs:

- a JSON file with the clustering metrics for each combination of the graph, clusterer, parameter set, and experiment round,
- a CSV file including the metrics of all runs specified by the clustering configuration file

To add a new clustering algorithm, the user just needs to add a new function that runs the clustering algorithm given an input graph and the parameters of the algorithm.

3 EMPIRICAL EVALUATION

This section presents the results obtained by running the PCBS benchmark. We study the performance of all algorithms described in Section 2.3 on a variety of different graphs.

We show that using our PCBS benchmark library, we are able to obtain new insights on comparing scalable parallel graph clustering algorithms. In our experiments, we mainly want to answer the following questions:

(1) For the same algorithm, how fast is the implementation in PCBS compared to the implementations in other state-of-the-art graph libraries and databases? (Section 3.1)
(2) How do the quality and running time of the PCBS implementations compare on different tasks and which implementation is the most suitable for the task? (Sections 3.2–3.4)
(3) How do different implementations that optimize the modularity/correlation objective compare with respect to the ground truth? (Section 3.5)
(4) How does the sparsity of the constructed similarity graph impact clustering quality? (Appendix B)

Table 1: Undirected graph clustering algorithms implemented in PCBS and popular graph libraries and databases. The starred (*) implementations are sequential. The rows are sorted by the number of ticks. We exclude clustering algorithms that do not consider graph edges (e.g., k-means clustering). Modularity Clustering includes both Louvain and Leiden implementations. Hop Preference & Node Preference (HANP) [50] and Conductance Minimization [78] are variants of the Label Propagation algorithm. Neo4j implements a parallel version of the sequential maximum k-cut algorithm [35]. Memgraph, NebulaGraph, and Oracle Graph are graph databases described in Section 4.

	Graph Libraries			Graph Databases				
	Ours	**NetworKit**	**SNAP**	**Neo4j**	**NebulaGraph**	**Oracle Graph**	**TigerGraph**	**Memgraph**
Modularity Clustering	✓	✓	✓*	✓	✓	✓	✓	✓
Connectivity	✓	✓	✓*	✓	✓	✓	✓	✓
KCore	✓	✓	✓*	✓	✓	✓	✓	
Label Propagation	✓	✓		✓	✓	✓	✓	
SLPA	✓			✓			✓	
Infomap			✓*		✓	✓		
SCAN	✓							
TECTONIC	✓							
Unweighted LDD	✓							
HAC	✓							
Affinity Clustering	✓							
Correlation Clustering	✓							
Approximate Maximum k-cut				✓				
Conductance Minimization						✓		
HANP					✓			

We evaluate the clustering algorithms on three different tasks: community detection, classification, and dense subgraph mining. For each task, the experiments use different input data and/or objectives. While we try to group the experimental results into tasks to help organize the results and analyze the behavior of the algorithms, we note that the boundary between the tasks is not strict.

Results Summary. We find that PCBS is consistently much faster than Neo4j, TigerGraph, and SNAP, and in most cases also faster than NetworKit. However, it is worth noting that graph databases (e.g. Neo4j and TigerGraph) offer more comprehensive database functionality in addition to clustering.

We also observed that correlation clustering produces top quality clusters in community detection, classification, and dense subgraph mining. While the quality of correlation clustering has been studied on unweighted graphs (e.g., [75, 87]), to the best of our knowledge, we are the first work to comprehensively evaluate it on weighted graphs.

ParHAC [29] also achieves relatively high quality. While a single run of ParHAC may be slower than other methods, it offers the advantage of being a hierarchical approach. This means that in a single run, it can generate clusterings at multiple levels of granularity. In contrast, methods like correlation and modularity clustering can only produce a single clustering in a one run.

Parallel Computation Environment. We use *c2-standard-60* instances on the Google Cloud Platform. These are 30-core machines with two-way hyper-threading with Intel 3.1 GHz Cascade Lake processors (with a maximum turbo clock speed of 3.8 GHz). We use all 60 hyper-threads for our experiments, except for the experiments specifically investigating how performance scales with the number of threads, and for Neo4j, where we only use 4 threads because that is the maximum number of threads supported in the community version.

Datasets. We use a variety of large real-world and synthetic datasets, summarized in Tables 2 to 4. We also present results on five small datasets from UCI machine learning repository [9] in Appendix E. All graphs are undirected.

Table 2: Unweighted graph datasets from SNAP [49].

Dataset	**Num. Vertices**	**Num. Edges**
Amazon (AM)	334,863	925,872
com-DBLP (DB)	425,957	2,099,732
YouTube-Sym (YT)	1,138,499	5,980,886
LiveJournal (LJ)	4,847,571	85,702,474
com-Orkut (OK)	3,072,627	234,370,166
Friendster (FS)	65,608,366	3,612,134,270

Table 2 presents unweighted real-world graphs from the Stanford Network Analysis Project (SNAP) [49]. For these graphs, we used the top 5000 communities as the ground truth, and the ground truth clusters may be overlapping.

Table 3 describes the real-world vector embedding datasets [95] from which we derive our weighted k-nearest neighbor graphs. The embeddings all have 1024 dimensions, except for MNIST, whose dimensionality is 768. The edge weights between vertices u, v are $\frac{1}{1+d(u,v)}$, where $d(u, v)$ is the Euclidean distance between points u, v. The ground truth clusters are non-overlapping. In this section, we present the results for $k = 50$. In Appendix B, we also present the results for $k = 10$ and $k = 100$.

Table 4 presents synthetic unweighted RMAT graphs [21] generated using GBBS [27] with parameters $a = 0.5$, $b = c = 0.1$, and $d = 0.3$. One of them is relatively dense (`RMAT-1`), while the other is relatively sparse (`RMAT-2`).

More details on these datasets are also presented in Appendix C.

Evaluation. We evaluate the clustering quality by metrics that compare with the ground truth as well as the edge density of the clusters, which is not associated with any ground truth. We use precision, recall, and F_β score with $\beta = 0.5$ to measure the difference between our clustering and the ground truth. We note that we use $\beta = 0.5$ instead of the default setting of $\beta = 1$, which puts more

Table 3: Embedding datasets from Yu et al. [95]. "Num. Clusters" is the number of ground truth clusters.

Dataset	Num. Vertices	Num. Clusters
MNIST	70,000	10
StackExchange	373,850	121
Reddit	420,464	50
ImageNet	1,281,167	1000

Table 4: Synthetic RMAT graphs.

Dataset	Num. Vertices	Num. Edges	Avg. Degree
RMAT-1	2^{15}	31,605,326	964.5
RMAT-2	2^{21}	396,360,032	189.0

weight on precision than recall. This is because our focus is on finding clusters of high precision.

We also compute the area under the precision vs. recall curve (AUC). Again, since we focus on the high-precision regime, we only consider results whose precision is in $[0.5, 1]$. A larger AUC score means the method obtains both high precision and recall. We report the AUC times 2, so a perfect algorithm gets an AUC of 1. We use AUC in addition to F_β score because it considers the overall performance of an algorithm for many different parameter values, while the F_β score measures the performance of an algorithm at a single set of parameters.

Pareto frontier. We present the Pareto frontiers of (a) precision vs. recall and (b) clustering quality ($F_{0.5}$ score) vs. runtime. The Pareto frontier comprises non-dominated points. To create the Pareto frontier, we run the clustering algorithms with different parameters that allow the resulting clustering to cover the whole precision range as much as possible. The result of each run is then represented as a point (x, y), e.g., in the context of clustering quality vs. runtime, x is the running time and y is the F_β score. After that, we keep each point (x, y), for which there does not exist another point (x', y'), such that $x' \geq x$ and $y' \geq y$. In the precision vs. recall Pareto frontier plots, the *top right* is better (high precision and high recall). In the clustering quality vs. runtime Pareto frontier plots, the *top left* is better (high quality and low running time).

Baseline Graph Libraries and Databases. We compare with NetworKit [80], Neo4j [5], TigerGraph [7], the original implementation of TECTONIC [86], and clustering implementations provided in SNAP [49]. All algorithms are parallel, except for the original implementation of TECTONIC and the implementations in SNAP, which are sequential. The list of algorithms implemented in each graph library or database can be found in Table 1.

The NetworKit algorithms are implemented in C++ with Python bindings. Neo4j's core components are implemented in Java. TigerGraph algorithms are implemented using GSQL on top of the core components implemented in C++.

The original TECTONIC implementation counts triangles in C, and then writes the results to disk. Finally, C++ is used to read the triangles from the disk and compute the clusters. It does not appear to be optimized for running time.

Several graph libraries and databases have multiple different modularity clustering implementations, and we describe them here. Our parallel modularity and correlation clustering implementation is Louvain-based and optimizes the LambdaCC objective, as described in Section 2.3.

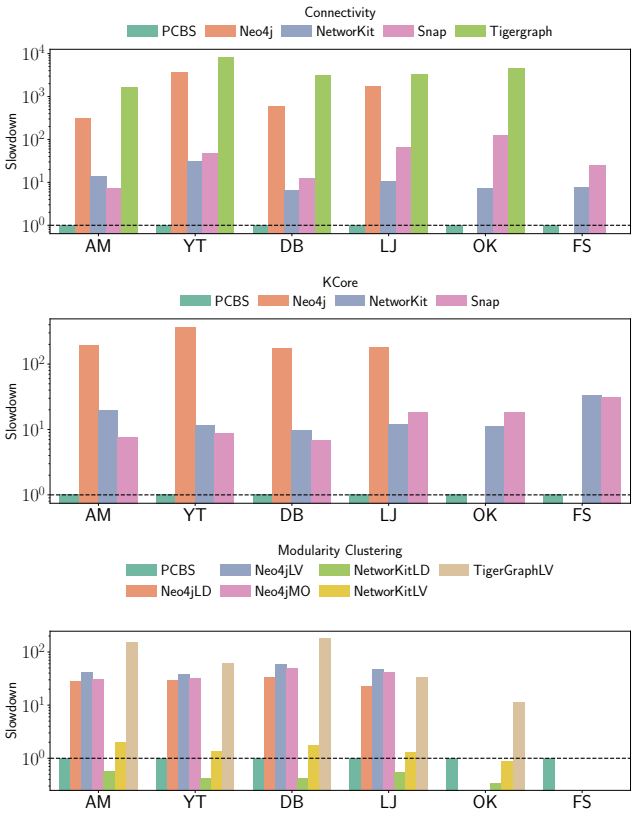

Figure 2: Slowdown of methods on SNAP graphs with respect to PCBS (`ParC`). Neo4j cannot load `orkut` and `friendster`. TigerGraph cannot load `friendster`. NetworKit failed to run on `friendster`. "LD" methods are Leiden-based. "LV" methods are Louvain-based. "MO" is a Girvan-Newman implementation. The horizontal dashed line is at slowdown=1.

- NetworKit [80] has two parallel implementations for modularity clustering: Louvain method (NetworKitPLM) [79] and Leiden Method [83] (NetworKitParalleLeiden).
- SNAP [49] has two sequential implementations: CommunityGirvanNewman [37] and CommunityCNM [22].
- Neo4j [5] has three parallel implementations: Louvain method (Neo4jLouvain), Leiden method (Neo4jLeiden), and a Girvan-Newman method (Neo4jModularityOptimziation).
- TigerGraph [7] has a parallel Louvain clustering implementation (TigerGraphLouvain).

We compare these modularity implementations in Section 3.1 and Section 3.5. Other algorithms all have a single implementation in each library/database, and we compare them in Section 3.1.

3.1 Running Time Comparison with Baselines

In Figure 2, we show the running time comparison of PCBS and the baselines on selected clustering methods. The y-axis is the slowdown compared to the method in PCBS. Other methods show a similar trend, and we present the running time comparison of all methods and their scalability in Appendix A(Figure 9).

On all tested implementations, PCBS is more than an order of magnitude faster than Neo4j, TigerGraph, and SNAP. Although SNAP and Neo4j do not use all 60 threads, we show in Figure 9 that

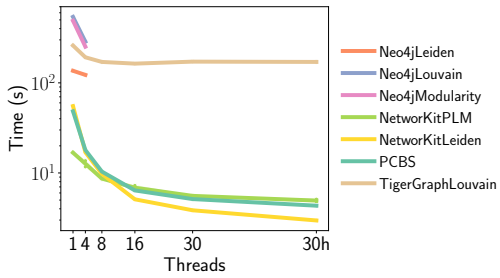

Figure 3: Scalability of modularity clustering implementations on `lj` with a resolution parameter of 1 and a maximum iteration count of 10. '30h' means using all 30-cores with two-way hyperthreading.

when using the same number of threads, PCBS is still significantly faster.

We attribute the performance of PCBS to three levels of optimization: (1) on the algorithm level, it uses theoretically-efficient algorithms; (2) on the implementation level, it is carefully engineered by, e.g., utilizing cache efficiently and minimizing the use of locks to reduce contention; and (3) on the system level, it uses an efficient C++ framework for parallel graph algorithms, i.e., GBBS [27] and ParlayLib [16].

Compared to NetworKit, PCBS is significantly faster on the connectivity clustering and k-core clustering tasks. On modularity clustering, PCBS's running time is comparable to NetworKitPLM (NetworKitLV), but slower than NetworKitParallelLeiden (NetworKitLD). However, in our experiments, NetworKit failed to run on `friendster`, and our implementation is the only implementation that successfully ran on `friendster`. Contrary to connectivity and k-core, these modularity clustering implementations can produce different clusterings, we further study the running time and quality of these different modularity clustering implementations under different parameters in Section 3.5. Though NetworKitParallelLeiden is faster, we will see that it obtains lower quality than PCBS's modularity clustering on some data sets.

On the largest two graphs, `orkut` and `friendster`, some bars are missing because Neo4j cannot load `orkut` and `friendster`, and TigerGraph cannot load `friendster` due to memory constraints. The SNAP modularity clustering implementation timed out on all graphs with a time limit of 7 hours. We suspect that slower performance of Neo4j and Tigergraph compared to PCBS and NetworKit comes from the fact that they support a more general graph database functionality, which incurs additional overheads.

We also compared our parallel TECTONIC implementation with the original TECTONIC [86] implementation, which is sequential. On the `youtube` data set with threshold 0.06, our implementation is 37x faster than the original TECTONIC when run on a single thread, and 387x faster when run on 60 threads.

For KCore, connectivity, TECTONIC, Label Propagation, and SLPA, we use our implementation in the rest of the experiments because PCBS's implementations are the fastest. For KCore, connectivity, and TECTONIC, PCBS's implementations compute the same clustering as baseline implementation. In the case of Label propagation and SLPA, while the algorithms are essentially the same, the resulting clusterings can be different because of non-determinism in thread scheduling and randomization.

Table 5: Area under precision-recall curve for precision ≥ 0.5 on SNAP graphs.

Clusterer	LJ	AM	DB	YT	OK	FS	Mean
Correlation	**0.75**	**0.94**	**0.71**	0.41	**0.32**	**0.43**	**0.59**
TECTONIC	0.67	0.94	0.67	0.42	0.21	0.29	0.53
Modularity	0.66	0.92	0.63	**0.46**	0.18	0.27	0.52
ParHAC-0.1	0.67	0.93	0.66	0.31	0.15	0.22	0.49
ParHAC-0.01	0.66	0.93	0.66	0.30	0.15	0.21	0.49
ParHAC-1	0.63	0.92	0.61	0.27	0.10	0.12	0.44
SLPA	0.65	0.86	0.57	0.00	0.00	0.00	0.35
SCAN	0.61	0.87	0.51	0.00	0.00	0.00	0.33
Affinity	0.49	0.81	0.47	0.00	0.00	0.00	0.30
LDD	0.42	0.60	0.38	0.25	0.00	0.09	0.29
KCore	0.20	0.55	0.23	0.39	0.03	0.12	0.25
LP	0.00	0.84	0.44	0.00	0.00	0.00	0.21

In Figure 3, we show the running time of different modularity clustering implementations using various numbers of threads. We see that PCBS is faster than Neo4j and TigerGraph on all thread counts. On lower thread counts, NetworKitPLM (NetworKitLV) is faster than PCBS and NetworKitParallelLeiden (NetworKitLD), but for more than 8 threads, NetworKitParallelLeiden (NetworKitLD) is the fastest. Though NetworKitLD is the fastest, its clustering quality is on average worse than PCBS's correlation clustering as shown in Section 3.5.

While graph databases, such as Neo4j [5] and TigerGraph [7], provide a richer functionality, on different graph clustering implementations, they are slower. For example, on the LiveJournal graph from SNAP [49], PCBS is on average 32.5x faster than Neo4j and 303x faster than TigerGraph. Compared with state-of-the-art parallel graph library NetworKit [80], PCBS is on average 4.54x faster. We compute the average using the geometric mean. For Neo4j, we use the execution time with 4 threads for both Neo4j and PBCS when computing the average speedup. We considered the following algorithms: connectivity, KCore, Neo4jLeiden, Label Propagation, and SLPA. For TigerGraph and NetworKit, we use the execution time with 60 threads for TigerGraph, NetworKit, and PBCS. We considered the following algorithms: connectivity, KCore, TigerGraphLouvain/NetworKitParallelLeiden, and Label Propagation.

3.2 Community Detection

In this section, we compare the algorithms on a community detection task. Here the input is an unweighted graph, whose edges describe real-world relations between individuals. The goal is to recover ground truth communities in the graph with high precision, by clustering the graph vertices.

In Figure 4, we show the Pareto frontier plots on four unweighted SNAP graphs. The results for all six graphs can be found in Figures 15 and 16 in the Appendix G (the two other graphs show a similar trend). We also present the area under the precision-recall curve (AUC) in Table 5. We exclude connectivity because for unweighted graphs, all vertices are in a single connected component.

Overall, correlation clustering usually achieves the best quality (for both $F_{0.5}$ score and AUC score), but it is relatively slow

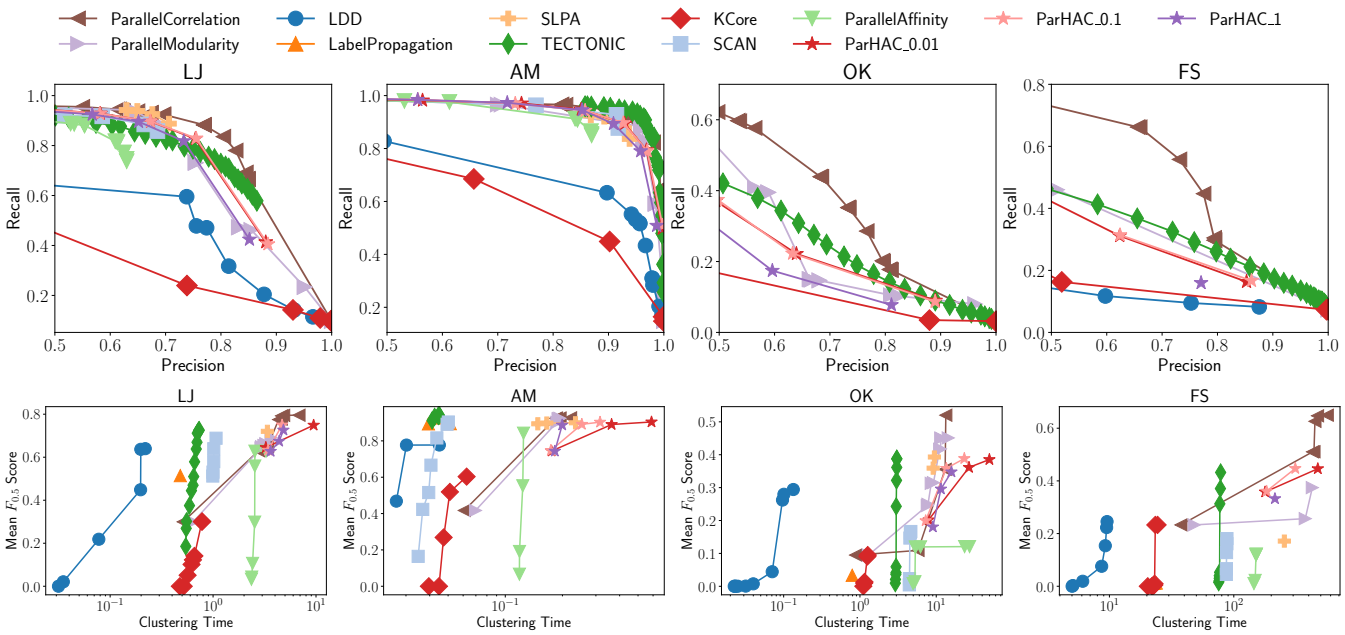

Figure 4: (Top) The Pareto frontier of the precision and recall of the unweighted SNAP graphs. (Bottom) The Pareto frontier of the $F_{0.5}$ and runtime graph for the unweighted SNAP graphs. "ParHAC_ϵ" shows the curve for ParHAC implementation with approximation parameter ϵ.

compared to other clustering implementations. TECTONIC usually achieves lower quality than correlation clustering, but it is much faster. LDD is the fastest method but has even lower quality than TECTONIC. TECTONIC often has higher quality than other methods such as LDD, KCore, LP, SCAN, and Affinity.

On the largest two graphs, `orkut` and `friendster`, correlation clustering gets a significantly higher $F_{0.5}$ score than the other methods. Modularity clustering obtains a lower score than correlation clustering on `friendster`, because for large threshold parameters which correspond to the highest $F_{0.5}$ scores, modularity clustering runs out of memory. In comparison, correlation clustering can run for parameter values spanning the entire precision-recall value range. ParHAC and affinity overall perform worse than correlation clustering, which is likely due to the fact that they heavily rely on edge weights (which in this task are all set to 1). On `friendster`, ParHAC with $\epsilon = 1$ also runs out of memory for some threshold values due to the high memory cost of performing a large set of changes to the contracted graph in the implementation when using large values of ϵ.

Compared to other methods, LDD and KCore do not have high quality. SLPA, LP, and SCAN can achieve good quality on the smallest `amazon` dataset, but have poor quality on the larger graphs, and have a hard time achieving high precision. These methods do not have a threshold parameter that controls the clustering resolution and so their Pareto frontier cannot span the entire precision range.

Overall, based on our findings, we make the following recommendations for community detection:

- For generating high-quality clusters, correlation clustering is recommended.
- If one wants to sacrifice some quality for faster running time, TECTONIC is a good option.
- If speed is the top priority, LDD can be used. However, this comes at a cost of a significant drop in quality.

3.3 Classification

In this task, the goal is to classify data points represented by vector embeddings. We solve this task by building a weighted k-nearest neighbor graph on the input vectors, i.e., a graph in which each vertex represents a single vector, and each vector is connected to its k nearest vectors. We use the embeddings described in Table 3 as input. Clustering the resulting k-nearest neighbor graph is an unsupervised method for classifying the vectors. While clustering by itself would not produce a label for each cluster, it is a powerful method when the number of classes is not known upfront.

We show Pareto frontier plots in Figure 5 and the area under the precision-recall curve (AUC) in Table 6. We exclude the k-core algorithm because in the k-nearest neighbor graphs, all vertices are in the k-core.

Here we can see that clustering algorithms that leverage edge weights (affinity, correlation, modularity, and ParHAC) do better than clustering algorithms that do not (LDD, SCAN, LP, SLPA, and TECTONIC). One exception is that the simple connectivity algorithm, which leverages edge weights by thresholding on them, is not always better than the algorithms that do not use edge weights. For example, on `ImageNet`, connectivity is noticeably better than LDD and TECTONIC, but it is worse than TECTONIC on `MNIST`.

Among the top-performing methods (affinity, correlation, modularity, and ParHAC), correlation clustering and modularity clustering are the fastest and achieve the highest $F_{0.5}$ scores and AUC scores. The quality of affinity clustering and ParHAC are comparable and are slightly lower than correlation clustering and modularity clustering on `Reddit` and `StackExchange`.

Table 6: Area under curve for precision ≥ 0.5 on weighted k-nearest neighbor graphs with $k = 50$.

Clusterer	MNIST	ImageNet	Reddit	StackExchange	Mean
Correlation	**0.88**	**0.77**	**0.33**	**0.20**	**0.54**
Modularity	0.87	0.73	0.32	0.19	0.53
ParHAC-0.01	0.87	0.73	0.28	0.18	0.51
ParHAC-0.1	0.83	0.73	0.28	0.17	0.50
ParHAC-1	0.73	0.70	0.21	0.11	0.44
Affinity	0.79	0.66	0.16	0.08	0.42
LP	0.64	0.57	0.06	0.16	0.36
SLPA	0.25	0.50	0.08	0.17	0.25
TECTONIC	0.34	0.27	0.08	0.06	0.18
Connectivity	0.16	0.42	0.07	0.04	0.17
LDD	0.11	0.25	0.04	0.02	0.11
SCAN	0.00	0.00	0.05	0.00	0.01

The propagation-based methods LP and SLPA can sometimes perform well (e.g., on `ImageNet`), but their performance is not stable and they have very low scores on `Reddit`. While LP is faster than the top-performing methods, SLPA is not significantly faster and does not show a clear advantage over other methods.

Overall, based on our findings, we make the following observations for the classification task:

- For generating high-quality clusters at only a few granularities, correlation clustering and modularity clustering are recommended.
- For generating high-quality clusters at many different granularity levels, ParHAC and affinity clustering are recommended because of their hierarchical nature. Though affinity clustering is slower than ParHAC and correlation clustering on these four data sets, we note that it is faster on larger data sets, as shown in Section 3.2.
- For fast clustering without the need to tune parameters, LP is recommended, although the clustering quality can be unstable.
- LDD and connectivity are orders of magnitude faster than correlation clustering. While their quality is significantly lower, the fact that non-trivial clustering can be achieved extremely quickly raises a question whether a fast high-quality method exists.
- TECTONIC provides a middle ground in the runtime-quality trade-off. It is faster than correlation clustering, and has higher quality than LDD and connectivity. However, its quality is lower than that of correlation clustering in most cases.

3.4 Dense Subgraph Mining

In this section, we evaluate the algorithms on a dense subgraph mining task. Here, the goal is to group vertices into dense clusters. The main difference from the community detection task is that our goal is to directly optimize cluster density, rather than optimizing quality with respect to ground truth labels. In Figure 6, we show the *weighted edge density mean* of the clusters on artificially generated unweighted RMAT graphs. We refer the readers to Section 2.5 for the definition of weighted edge density.

Modularity and correlation clustering obtain the densest clusters. When the number of clusters is relatively small, modularity clustering produce clusters denser than correlation clustering on average, but when the number of clusters is very large, correlation clustering produces denser clusters. ParHAC also produces dense clusters, but overall less dense than Modularity and correlation clustering. LDD and TECTONIC produce less dense clusters. SCAN can produce dense clusters when the number of clusters is very small, but cannot produce very dense clusters with larger number of clusters. LP and affinity clustering can only produce small number of clusters, and the density on these two datasets are similar with SCAN.

Overall, we recommend using modularity clustering when the number of clusters is relatively small, and correlation clustering otherwise.

3.5 Comparing Different Modularity Clustering Implementations

Correlation and modularity clustering methods have shown a strong performance overall compared with other clustering algorithms in our benchmark. While parallel modularity clustering algorithms usually follow the same overall framework of local search and graph coarsening [17], the available implementations differ in a number of ways (e.g., in terms of synchronization, symmetry breaking, heuristic optimizations, or the use of refinement [75]) that result in differences in the running time and clustering quality. In this section, we perform a comparison between PCBS's correlation and modularity clustering and the different baseline clustering algorithms that optimize the modularity objective.

In Figure 7, we show the precision and recall comparison of different modularity implementations on a subset of datasets. The results on all datasets can be found in Figure 18 and Figure 19 in Appendix F. SnapCNM is only able to finish on `MNIST` so it is only shown on a single subplot.

We see that PCBS's correlation clustering and modularity clustering implementations are competitive with NetworKit's PLM and NetworKit's ParallelLeiden implementations. Other modularity objective-based methods are much slower and do not achieve higher quality. Methods that do not have a resolution parameter have unstable clustering quality (Neo4jModularityOptimization, Neo4jLouvain, and TigerGraphLouvain).

Compared to NetworKitPLM, PCBS (ParallelCorrelation and ParallelModularity) is slightly slower on the weighted graphs, but is faster and has higher quality on the unweighted graphs. Compared to NetworKitParallelLeiden, PCBS has higher quality on most datasets, and has similar running time.

Finally, we compare the modularity objective obtained by different modularity methods. In Table 7, we show the modularity scores with $\gamma = 1$ for `LJ`, using different modularity methods. In Figure 20, we show the scores for all unweighted graphs. We observe that PCBS's modularity clustering, Neo4jLouvain, NetworKitPLM, and Neo4jLeiden get similar scores, and are the highest ones. Neo4jModularity, and NetworKitLeiden get similar scores, and are slightly lower than the methods above. TigerGraphLouvain gets the lowest score.

4 RELATED WORK

Graph Libraries. Libraries for graphs, such as NetworkX [41], SNAP [49], GBBS [27], GraphMineSuite [15], and NetworKit [80], are software packages that provide data structures and (parallel)

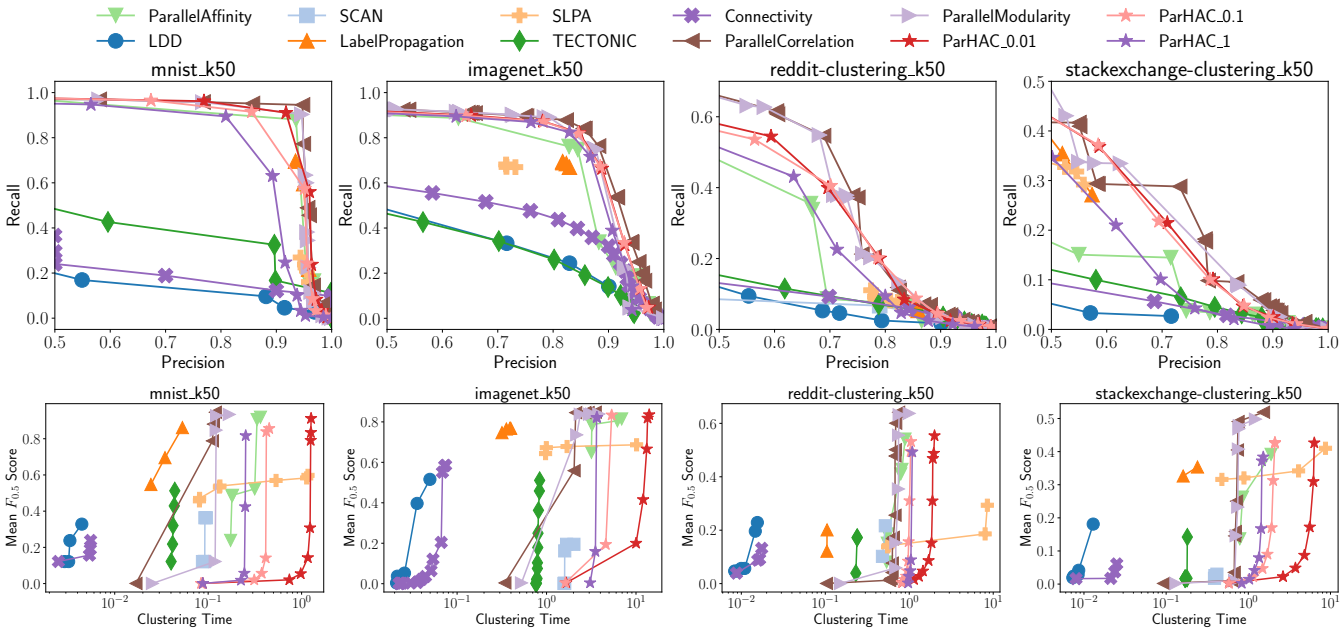

Figure 5: (Top) The Pareto frontier of precision and recall for the weighted k-nearest neighbor graphs ($k = 50$), using PCBS methods. (Bottom) The Pareto frontier of $F_{0.5}$ score and clustering time on k-nearest neighbor graphs ($k = 50$).

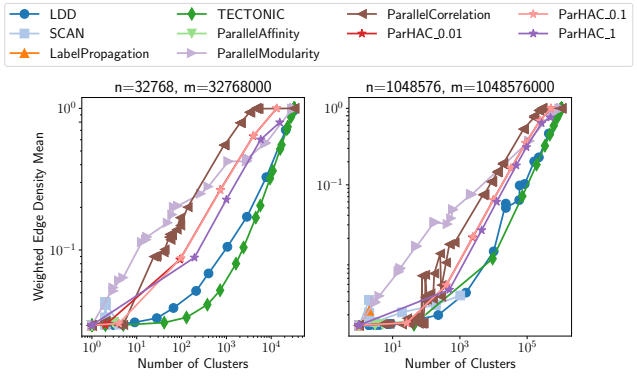

Figure 6: Weighted average edge density of the clusters in RMAT graphs, with different numbers of vertices (n) and edges (m).

Table 7: The modularity scores with $\gamma = 1$ for LJ, using different modularity methods.

Clusterer	Modularity Objective
PCBS Modularity	0.755035
Neo4j Louvain	0.754733
NetworKit PLM	0.753670
Neo4j Leiden	0.743583
NetworKit Leiden	0.662478
Neo4j Modularity	0.654594
TigerGraph Louvain	0.022660

algorithms for representing and operating on graphs within a programming language. These libraries focus on in-memory computation and analysis of graph data, offering flexibility to build custom graph processing applications and algorithms. Compared with graph databases, graph libraries are usually have no support for declarative graph query languages and do not provide database features. However, they can have more optimized graph algorithms because they support a smaller set of functionalities.

Graph Processing Frameworks. There have also been many graph processing frameworks developed (e.g., [38, 53, 57, 67, 77] among many others). They provide high-level abstractions and APIs for expressing parallel graph algorithms. We refer the reader to [60, 92] for surveys of existing frameworks. Several recent graph processing systems evaluate the scalability of their implementations by solving problems on massive graphs [26, 45, 55, 81, 96].

Graph Databases. Graph databases are specialized database management systems designed explicitly for storing and querying highly interconnected data represented as nodes, edges, and properties. They support features like traversing along data relationships, running graph queries, and analytics algorithms like finding shortest paths or detecting communities and patterns. Unlike libraries, graph databases support declarative graph query languages like Cypher, Gremlin, and SPARQL, and are optimized for efficiently persisting and retrieving graph data. Additionally, they provide features like transactions, access control, and data management capabilities. Examples of popular graph databases include Neo4j [5, 32], TigerGraph [7], Amazon Neptune [1], Memgraph [3], OrientDB [6], ArangoDB [2], and NebulaGraph [4]. Besta et al. [14] provide a comprehensive survey of graph databases.

Experimental Comparisons. Due to the importance of the clustering problem and the abundance of different algorithms, there are many comparative studies on clustering algorithms. Yang and Leskovec [93] compare clustering algorithms on social, collaboration, and information networks. Their work focuses on community scoring functions and does not compare the running times of the algorithms. Fortunato [36] survey clustering methods in graphs

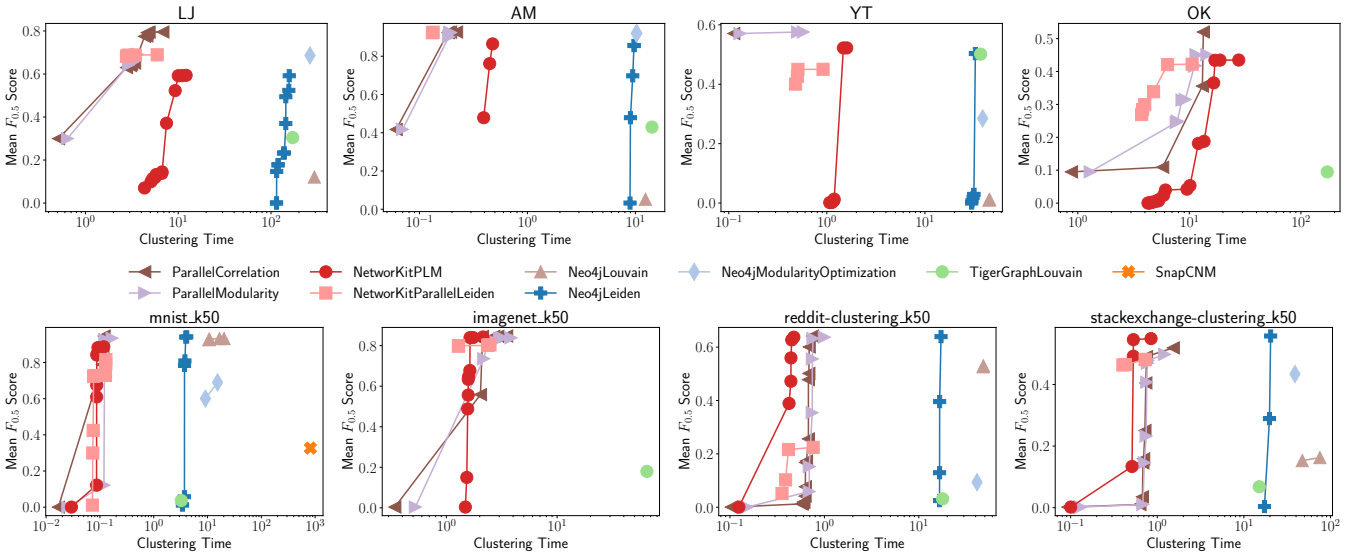

Figure 7: The Pareto frontiers for the unweighted SNAP graphs, using different modularity implementations.

and compared many algorithms, although they do not present experiments for parallel methods. Xie et al. [89] review overlapping clustering algorithms, quality measures, and benchmarks. There are also works that focus on evaluating algorithms on artificial networks [56, 68, 69, 76, 94]. Park et al. [70] show that popular clustering methods such as Leiden method and InfoMap do not find clusters that are well-connected. There are also graph clustering benchmarks for overlapping clusters and heterogeneous graphs [47, 48].

The aforementioned works usually only compare the quality or running time of the algorithms, but not both. For the works that compare both, they do not experiment with parallel algorithms. Bader et al. [10]'s graph clustering challenge has an objective function, where both quality and speed contributed to the final scores, but their quality measures do not compare with ground truth clusters. Moreover, they only released graph datasets but did not provide comprehensive experimental results. As far as we know, we are the first to compare the parallel graph clustering algorithms both qualitatively and quantitatively.

There have been numerous works experimentally comparing the performance of graph processing systems on parallel machines [18, 39, 43, 54, 82]. Monteiro et al. [64] compared JanusGraph, Nebula Graph, Neo4j, and TigerGraph, and found that Neo4j outperformed other graph databases in terms of node load time and query execution time in their experiments. Fernandes and Bernardino [34] compared the features of AllegroGraph, ArangoDB, InfiniteGraph, Neo4j, and OrientDB. Dominguez-Sal et al. [32] compared Neo4j, Jena, HypergraphDB, and DEX (now renamed to Sparksee). Angles et al. [8] compared five different database systems representing graph (DEX and Neo4j), RDF (RDF-3X) and relational (Virtuoso and PostgreSQL) data management. However, these benchmarks focus on graph algorithms such as breadth-first search, PageRank, and shortest paths, or simple queries such as k-hop neighborhood queries, and do not give a comparison among clustering algorithms. Some of them evaluate connected components algorithms, which is one of the algorithms that we evaluated.

There are some works that compare graph databases on a single community detection algorithm. Beis et al. [13] benchmarked the implementations of the Louvain algorithm in Titan, OrientDB, and Neo4j in 2015, but the implementations are not parallel. More recently, Kalogeras et al. [46] compared the label propagation algorithm in Neo4j and Apache Spark in the distributed setting.

5 CONCLUSION

In this paper, we presented the ParClusterers Benchmark Suite (PCBS), a comprehensive library for evaluating scalable parallel graph clustering algorithms. PCBS includes efficient implementations of many graph clustering algorithms and a benchmarking framework for comparing their performance on different tasks and datasets. Through extensive experiments, we showed that many of PCBS's clustering implementations are significantly faster than other graph libraries and databases, while obtaining very good, and in many cases the best, clustering quality. We also provided insights into which clustering algorithms are most suitable for different tasks like community detection, classification, and dense subgraph mining.

Besides the clustering applications explored in this work, there are also many other interesting future directions. One idea is to study how weighted graph clustering algorithms perform when derived weights are added to the unweighted graph (for example, weights derived from node degrees as in [31]). Moreover, we observe a big gap between the clustering time of the slower algorithms (e.g., correlation clustering and ParHAC) and the fastest (e.g. LDD and connectivity). It would be interesting to study if we can close the gap by developing much faster clustering algorithms with close-to-best clustering quality that obtain the best of both worlds.

REFERENCES

[1] [n.d.]. Amazon Neptune. https://aws.amazon.com/neptune/.
[2] [n.d.]. ArangoDB. https://www.arangodb.com/.
[3] [n.d.]. Memgraph. https://memgraph.com/.
[4] [n.d.]. NebulaGraph. https://nebula-graph.io/.
[5] [n.d.]. Neo4j. https://neo4j.com/.
[6] [n.d.]. OrientDB. https://orientdb.com/.
[7] [n.d.]. TigerGraph. https://www.tigergraph.com/.
[8] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. 2013. Benchmarking Database Systems for Social Network Applications. In *First International Workshop on Graph Data Management Experiences and Systems* (New York, New York) *(GRADES '13)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. https://doi.org/10.1145/2484425.2484440
[9] Arthur Asuncion and David Newman. 2007. UCI machine learning repository.
[10] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. 2014. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 73–82.
[11] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. 2004. Correlation clustering. *Machine learning* 56 (2004), 89–113.
[12] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. 2017. Affinity clustering: Hierarchical clustering at scale. *Advances in Neural Information Processing Systems* 30 (2017).
[13] Sotirios Beis, Symeon Papadopoulos, and Yiannis Kompatsiaris. 2015. Benchmarking graph databases on the problem of community detection. In *New Trends in Database and Information Systems II: Selected papers of the 18th East European Conference on Advances in Databases and Information Systems and Associated Satellite Events, ADBIS 2014 Ohrid, Macedonia, September 7-10, 2014 Proceedings II*. Springer, 3–14.
[14] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017* (2019).
[15] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. GraphMineSuite: enabling high-performance and programmable graph mining algorithms with set algebra. *Proc. VLDB Endow.* 14, 11 (jul 2021), 1922–1935. https://doi.org/10.14778/3476249.3476252
[16] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 507–509. https://doi.org/10.1145/3350755.3400254
[17] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.
[18] Angela Bonifati, George Fletcher, Jan Hidders, and Alexandru Iosup. 2018. A survey of benchmarks for graph-processing systems. *Graph Data Management: Fundamental Issues and Recent Developments* (2018), 163–186.
[19] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections *i*SAX2+. *Knowledge and Information Systems* 39, 1 (2014), 123–151.
[20] José E Chacón and Ana I Rastrojo. 2023. Minimum adjusted Rand index for two clusterings of a given size. *Advances in Data Analysis and Classification* 17, 1 (2023), 125–133.
[21] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining (SDM)*. 442–446.
[22] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical review E* 70, 6 (2004), 066111.
[23] Leon Danon, Albert Diaz-Guilera, Jordi Duch, and Alex Arenas. 2005. Comparing community structure identification. *Journal of statistical mechanics: Theory and experiment* 2005, 09 (2005), P09008.
[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.
[25] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.
[26] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[27] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[28] Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. 2021. Hierarchical agglomerative graph clustering in nearly-linear time. In *International conference on machine learning*. PMLR, 2676–2686.
[29] Laxman Dhulipala, David Eisenstat, Jakub Lacki, Vahab Mirrokni, and Jessica Shi. 2022. Hierarchical agglomerative graph clustering in poly-logarithmic depth. *Advances in Neural Information Processing Systems* 35 (2022), 22925–22940.
[30] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. Connectit: A framework for static and incremental parallel graph connectivity algorithms. *arXiv preprint arXiv:2008.03909* (2020).
[31] Laxman Dhulipala, Jakub Łącki, Jason Lee, and Vahab Mirrokni. 2023. TeraHAC: Hierarchical agglomerative clustering of trillion-edge graphs. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–27.
[32] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. 2010. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. In *Web-Age Information Management*, Heng Tao Shen, Jian Pei, M. Tamer Özsu, Lei Zou, Jiaheng Lu, Tok-Wang Ling, Ge Yu, Yi Zhuang, and Jie Shao (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–48.
[33] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).
[34] Diogo Fernandes and Jorge Bernardino. 2018. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB.. In *Data*. 373–380.
[35] Paola Festa, Panos M Pardalos, Mauricio GC Resende, and Celso C Ribeiro. 2002. Randomized heuristics for the MAX-CUT problem. *Optimization methods and software* 17, 6 (2002), 1033–1058.
[36] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.
[37] M. Girvan and M. E. J. Newman. 2002. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences (PNAS)* 99, 12 (2002), 7821–7826.
[38] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.
[39] Yong Guo, A. Varbanescu, A. Iosup, and D. Epema. 2015. An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* null (2015), 423–432. https://doi.org/10.1109/CCGrid.2015.20
[40] Fred G Gustavson. 1972. Some basic techniques for solving sparse systems of linear equations. In *Sparse Matrices and their Applications*. Springer, 41–52.
[41] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
[42] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of classification* 2 (1985), 193–218.
[43] A. Iosup, T. Hegeman, W. L. Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, M. Capotă, N. Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and P. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270
[44] Siddhartha V Jayanti and Robert E Tarjan. 2016. A randomized concurrent algorithm for disjoint set union. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. 75–82.
[45] Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2017. BigSparse: High-performance external graph analytics. *CoRR* abs/1710.07736 (2017). http://arxiv.org/abs/1710.07736
[46] Georgios Kalogeras, Vassilios Tsakanikas, Ioannis Ballas, Vassilios Aggelopoulos, and Vassilios Tampakas. 2023. Community Detection at Scale: A Comparison Study among Apache Spark and Neo4j *(PCI '22)*. Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/3575879.3575961
[47] Malik Sebastian Stær Knudsen, Laurits Almskou Brodal, Peter Kristoffer Peczalski, Atefeh Moradan, Davide Mottin, and Ira Assent. 2022. GraB: Graph Benchmark for Heterogeneous Graph Clustering. In *The First Learning on Graphs Conference*.
[48] Andrea Lancichinetti, Santo Fortunato, and János Kertész. 2009. Detecting the overlapping and hierarchical community structure in complex networks. *New journal of physics* 11, 3 (2009), 033015.
[49] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.
[50] Ian XY Leung, Pan Hui, Pietro Lio, and Jon Crowcroft. 2009. Towards real-time community detection in large networks. *Physical Review E* 79, 6 (2009), 066107.
[51] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive

learning. *arXiv preprint arXiv:2308.03281* (2023).

[52] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. A ConvNet for the 2020s. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022).

[53] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. 340–349.

[54] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.* 8 (2014), 281–292. https://doi.org/10.14778/2735508.2735517

[55] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. *Proceedings of the Twelfth European Conference on Computer Systems* (2017).

[56] Seiji Maekawa, Jianpeng Zhang, George Fletcher, and Makoto Onizuka. 2019. General generator for attributed graphs with community structure. In *Proceedings of the ECML/PKDD Graph Embedding and Mining Workshop*. 1–5.

[57] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*. New York, NY, USA, 135–146. http://doi.acm.org/10.1145/1807167.1807184

[58] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[59] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 270–285.

[60] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[61] Aaron F McDaid, Derek Greene, and Neil Hurley. 2011. Normalized mutual information to evaluate overlapping community finding algorithms. *arXiv preprint arXiv:1110.2515* (2011).

[62] Gary L Miller, Richard Peng, and Shen Chen Xu. 2013. Parallel graph decompositions using random shifts. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. 196–203.

[63] Nicholas Monath, Kumar Avinava Dubey, Guru Guruganesh, Manzil Zaheer, Amr Ahmed, Andrew McCallum, Gokhan Mergen, Marc Najork, Mert Terzihan, Bryon Tjanaka, et al. 2021. Scalable hierarchical agglomerative clustering. In *Proceedings of the 27th ACM SIGKDD Conference on knowledge discovery & data mining*. 1245–1255.

[64] Jéssica Monteiro, Filipe Sá, and Jorge Bernardino. 2023. Experimental Evaluation of Graph Databases: JanusGraph, Nebula Graph, Neo4j, and TigerGraph. *Applied Sciences* 13, 9 (2023), 5770.

[65] Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, Andreas Vlachos and Isabelle Augenstein (Eds.). Association for Computational Linguistics, Dubrovnik, Croatia, 2014–2037. https://doi.org/10.18653/v1/2023.eacl-main.148

[66] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.

[67] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 456–471.

[68] Günce Keziban Orman, Vincent Labatut, and Hocine Cherifi. 2011. Qualitative comparison of community detection algorithms. In *Digital Information and Communication Technology and Its Applications: International Conference, DICTAP 2011, Dijon, France, June 21-23, 2011, Proceedings, Part II*. Springer, 265–279.

[69] Günce Keziban Orman, Vincent Labatut, and Hocine Cherifi. 2012. Comparative evaluation of community detection algorithms: a topological approach. *Journal of Statistical Mechanics: Theory and Experiment* 2012, 08 (2012), P08001.

[70] Minhyuk Park, Yasamin Tabatabaee, Vikram Ramavarapu, Baqiao Liu, Vidya Kamath Pailodi, Rajiv Ramachandran, Dmitriy Korobskiy, Fabio Ayres, George Chacko, and Tandy Warnow. 2023. Identifying Well-Connected Communities in Real-World and Synthetic Networks. In *International Conference on Complex Networks and Their Applications*. Springer, 3–14.

[71] Md Mostofa Ali Patwary, Suren Byna, Nadathur Rajagopalan Satish, Narayanan Sundaram, Zarija Lukić, Vadim Roytershteyn, Michael J Anderson, Yushu Yao, Pradeep Dubey, et al. 2015. BD-CATS: big data clustering at trillion particle scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[72] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[73] Jörg Reichardt and Stefan Bornholdt. 2006. Statistical Mechanics of Community Detection. *Phys. Rev. E* 74 (Jul 2006), 016110. Issue 1.

[74] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. 2016. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 455–468.

[75] Jessica Shi, Laxman Dhulipala, David Eisenstat, Jakub Łăcki, and Vahab Mirrokni. 2021. Scalable community detection via parallel correlation clustering. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2305–2313. https://doi.org/10.14778/3476249.3476282

[76] Lizhen Shi and Bo Chen. 2020. Comparison and Benchmark of Graph Clustering Algorithms. *arXiv preprint arXiv:2005.04806* (2020).

[77] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[78] Jyothish Soman and Ankur Narang. 2011. Fast community detection algorithm with gpus and multicore architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 568–579.

[79] Christian L. Staudt and Henning Meyerhenke. 2016. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 171–184. https://doi.org/10.1109/TPDS.2015.2390633

[80] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530.

[81] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting Label Propagation for Distributed Connected Components. In *ACM International Conference on Web Search and Data Mining (WSDM)*. 540–546.

[82] Yuanyuan Tian. 2023. The World of Graph Databases from An Industry Perspective. *SIGMOD Rec.* 51, 4 (jan 2023), 60–67. https://doi.org/10.1145/3582302.3582320

[83] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific reports* 9, 1 (2019), 5233.

[84] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2021. Parallel index-based structural graph clustering and its approximation. In *Proceedings of the 2021 International Conference on Management of Data*. ACM.

[85] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2021. Parallel Index-Based Structural Graph Clustering and Its Approximation. In *International Conference on Management of Data*. 1851–1864.

[86] Charalampos E Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. 2017. Scalable motif-aware graph clustering. In *Proceedings of the 26th International Conference on World Wide Web*. 1451–1460.

[87] Nate Veldt, David F Gleich, and Anthony Wirth. 2018. A correlation clustering framework for community detection. In *Proceedings of the 2018 World Wide Web Conference*. 439–448.

[88] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient structural graph clustering: an index-based approach. *Proceedings of the VLDB Endowment* 11, 3 (2017), 243–255.

[89] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. 2013. Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)* 45, 4 (2013), 1–35.

[90] Jierui Xie, Boleslaw K Szymanski, and Xiaoming Liu. 2011. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *2011 ieee 11th international conference on data mining workshops*. IEEE, 344–349.

[91] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 824–833.

[92] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. 2017. Big graph analytics platforms. *Foundations and Trends® in Databases* 7, 1-2 (2017), 1–195.

[93] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. 1–8.

[94] Zhao Yang, René Algesheimer, and Claudio J Tessone. 2016. A comparative analysis of community detection algorithms on artificial networks. *Scientific reports* 6, 1 (2016), 1–18.

[95] Shangdi Yu, Joshua Engels, Yihao Huang, and Julian Shun. 2023. PECANN: Parallel Efficient Clustering with Graph-Based Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2312.03940* (2023).

[96] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

A RUNTIME AND SCALABILITY OF ALL ALGORITHMS

In Figure 8, we show the running time comparisons of different implementations of LP and SLPA on the unweighted graphs. PCBS, NetworKit, and TigerGraph use all 60 threads. SNAP is sequential. Neo4j only uses 4 cores due to community edition limitations. We see that PCBS's implementations are the fastest.

In Figure 9, we show the running time of different implementations using different numbers of threads. For all plots except Tectonic, `livejournal` graph is used. For Tectonic, `youtube` graph is used because the original TECTONIC implementation fails to run on `livejournal`.

We see that for all methods experimented in Figure 9, PCBS's implementations are the fastest on all threads and they have good scalability with respect to the number of threads.

B USING DIFFERENT k FOR k-NEAREST NEIGHBOR GRAPHS

In Figures 10 and 11 we present the result on the k-nearest neighbor graphs when $k = 10$ and $k = 100$.

We observe that for the weighted graphs with a small number of ground truth clusters, increasing k from 10 to 50 improves the clustering quality of `reddit` and `stackexchange`. Increasing k more from 50 to 100 doesn't significantly improve the clustering quality of any of the four data sets. Moreover, for all three k values (10, 50, and 100) the comparisons between the clustering algorithms generally stay the same.

C MORE DETAILS ON DATASET

The datasets in Table 3 are obtained from Yu et al. [95]. `MNIST` [25] is a standard machine learning dataset that consists of 28×28 dimensional images of grayscale digits between 0 and 9. The i^{th} cluster corresponds to all occurences of digit i. `ImageNet` [24] is a standard image classification benchmark with more than one million images, each of size $224 \times 224 \times 3$. The images are from 1000 classes of everyday objects. Unlike for MNIST, we do not cluster the raw `ImageNet` images, but instead first pass each image through ConvNet [52] to get an embedding. Each ground truth cluster contains the embeddings corresponding to a single image class from the original `ImageNet` dataset. `Reddit` and `StackExchange` are text embedding datasets studied in the recent Massive Text Embedding Benchmark (MTEB) work [65]. We restrict our attention to embeddings from the best model on the current MTEB leaderboard, GTE-large [51].

The datasets in Table 3 use ParlayANN's Vamana graph method [59] to compute the approximate k-nearest neighbors.

D MORE DETAILS ON EXPERIMENTS

We used Neo4j community version 5.8.0 with graph data science library version 2.4.3 and TigerGraph version 3.9.2.

For k-core clustering, all baseline implementations only return the k-core decomposition and do not return a clustering. We implement a k-core clustering for NetworKit using its parallel connected components. For Neo4j and TigerGraph, we report only the k-core decomposition time.

Table 8: Description of the weighted UCI datasets. $k = 10$ is used to generate the k-nearest neighbor graphs.

UCI Dataset	Num. Vertices	Num. Cluster
faces	400	40
iris	150	3
wine	178	3
wdbc	569	2
digits	1797	10

Table 9: The AUC scores for the weighted UCI k-nearest neighbor graphs $k = 10$.

	faces	iris	digits	wdbc	wine	Mean
Correlation	0.58	0.93	**0.94**	0.85	**0.54**	**0.77**
Modularity	0.58	**0.94**	0.93	0.86	0.52	0.77
ParHac-0.01	0.60	0.92	0.92	**0.87**	0.48	0.76
Affinity	**0.63**	0.92	0.91	0.82	0.48	0.75
ParHac-0.1	0.58	0.86	0.92	0.81	0.50	0.73
ParHac-1	0.32	0.90	0.90	0.80	0.44	0.67
Tectonic	0.53	0.94	0.87	0.48	0.49	0.66
Scan	0.35	0.90	0.65	0.80	0.49	0.64
Connectivity	0.53	0.86	0.71	0.50	0.43	0.61
LDD	0.24	0.59	0.44	0.50	0.48	0.45
SLPA	0.35	0.57	0.39	0.14	0.27	0.34
LP	0.36	0.59	0.34	0.14	0.25	0.34

E EXPERIMENTS ON UCI DATASETS

In Figures 13 and 14, we present the results on five small UCI [9] k-nearest neighbor graphs. A summary of the datasets is in Table 8. We see that on these small data sets, the quality and runtime difference is not as significant as in the larger weighted graph as shown in Section 3.3.

F COMPARE MODULARITY CLUSTERING IMPLEMENTATIONS

In Figure 18 and Figure 19, we present the Pareto frontier plots of the modularity clustering implementations. In Figure 20, we show the modularity scores with $\gamma = 1$ for the unweighted graphs, using different modularity methods.

G FULL EXPERIMENTS ON SNAP GRAPHS

We show the experimental results on all 6 SNAP graphs in Figures 15 and 16.

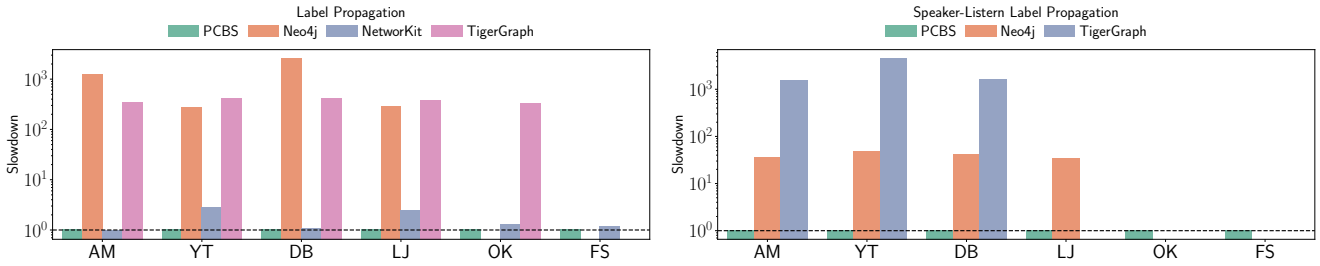

Figure 8: Slowdown of methods on the unweighted graphs.

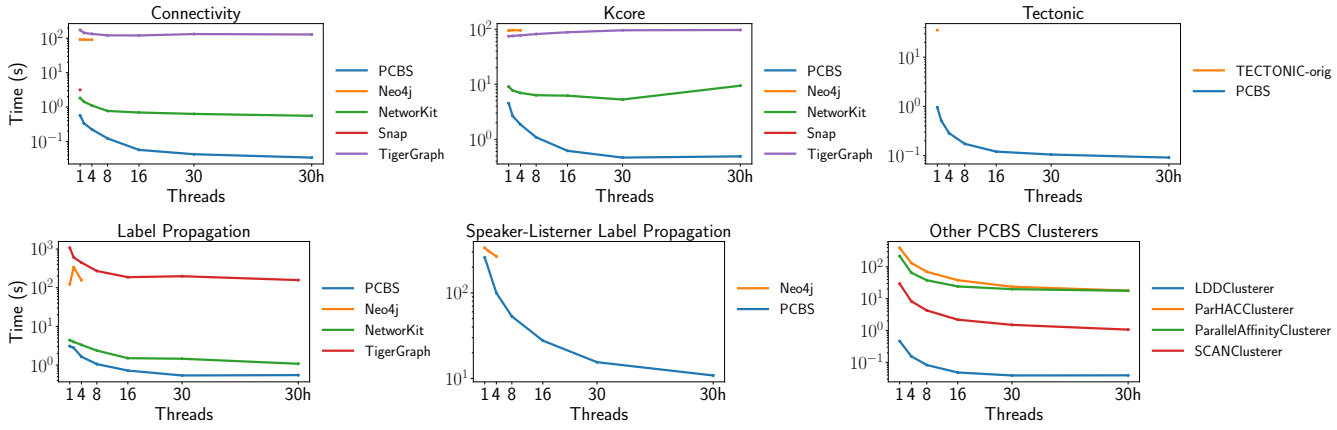

Figure 9: Running time of the clustering algorithms.

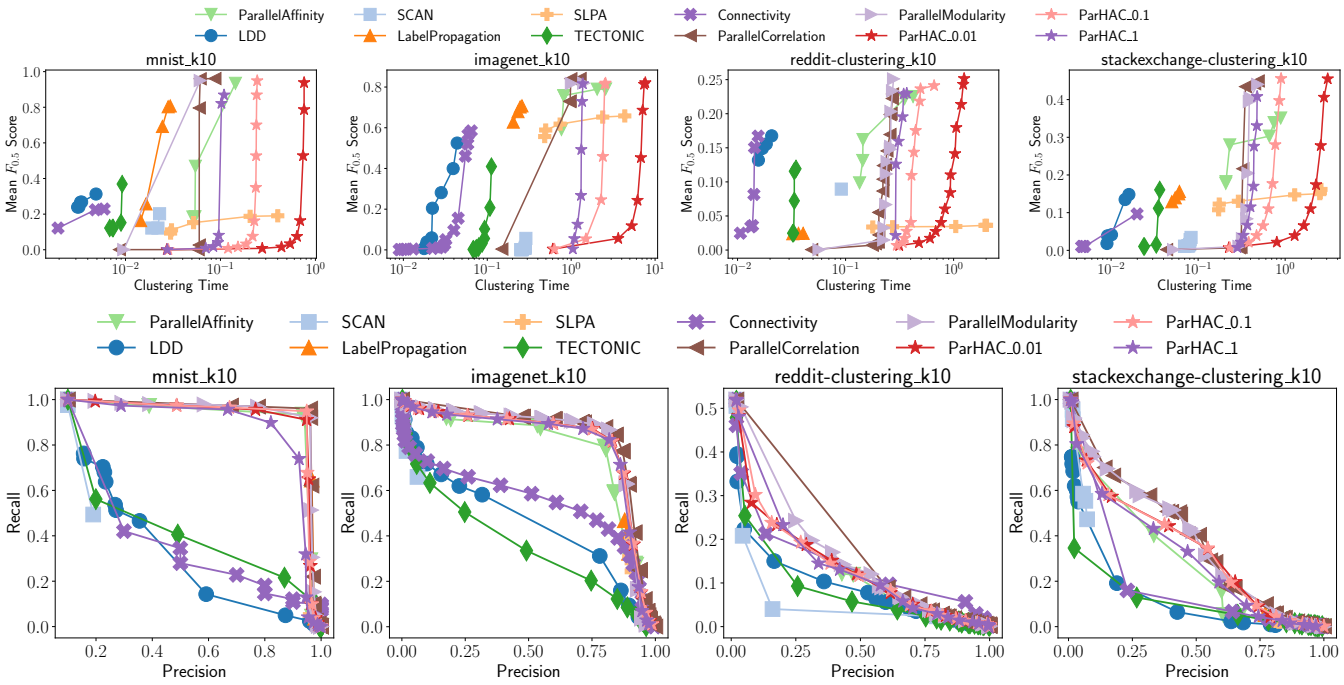

Figure 10: The Pareto frontier graphs for the weighted k-nearest neighbor graphs ($k = 10$), using PCBS methods.

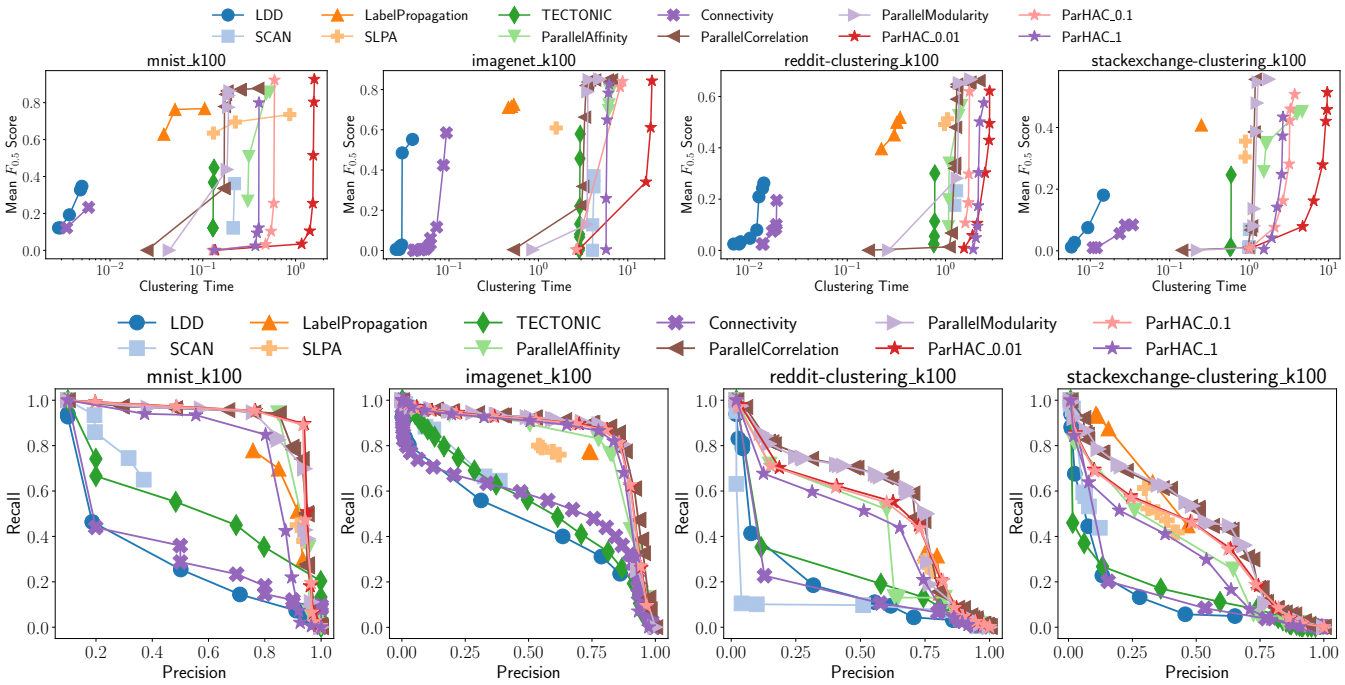

Figure 11: The Pareto frontier graphs for the weighted k-nearest neighbor graphs ($k = 100$), using PCBS methods.

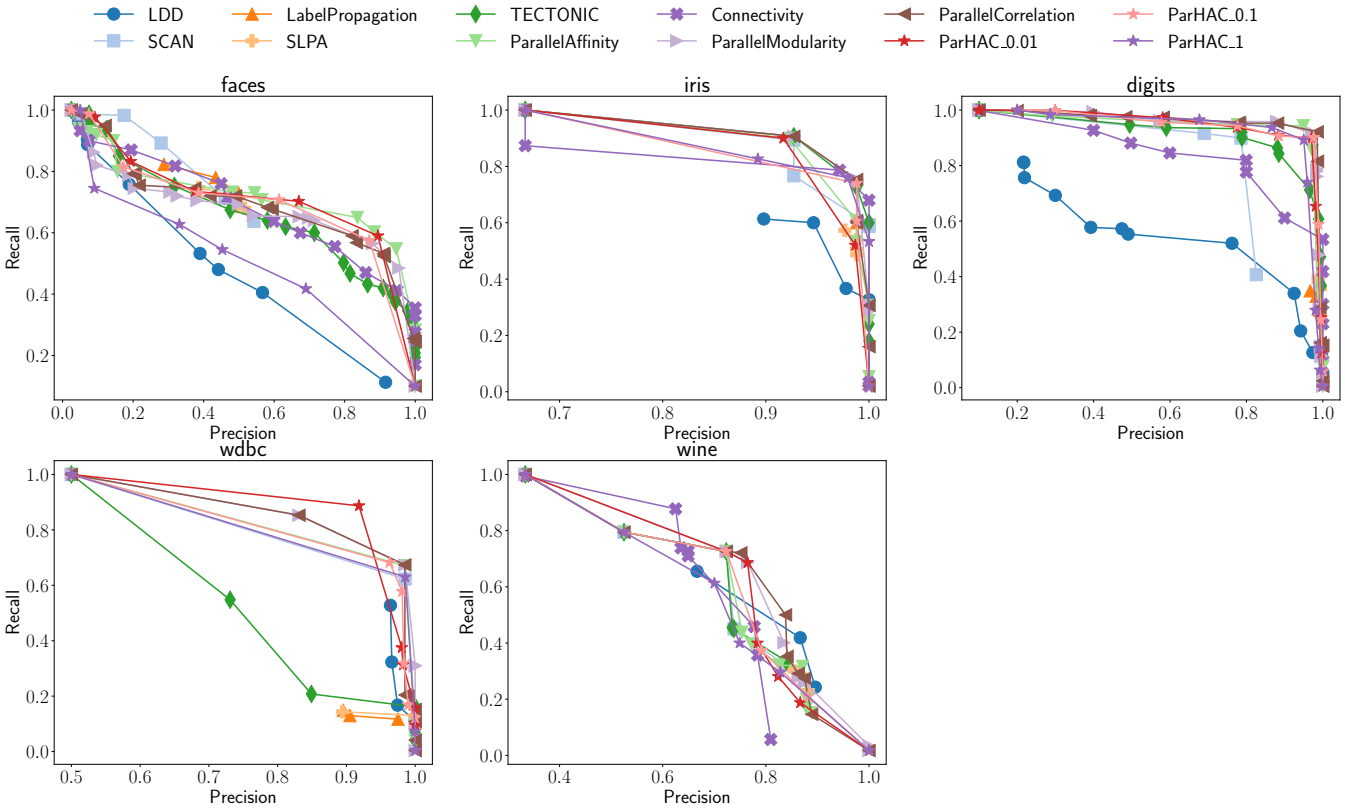

Figure 12: The Pareto frontier of the precision and recall for the weighted UCI k-nearest neighbor graphs $k = 10$.

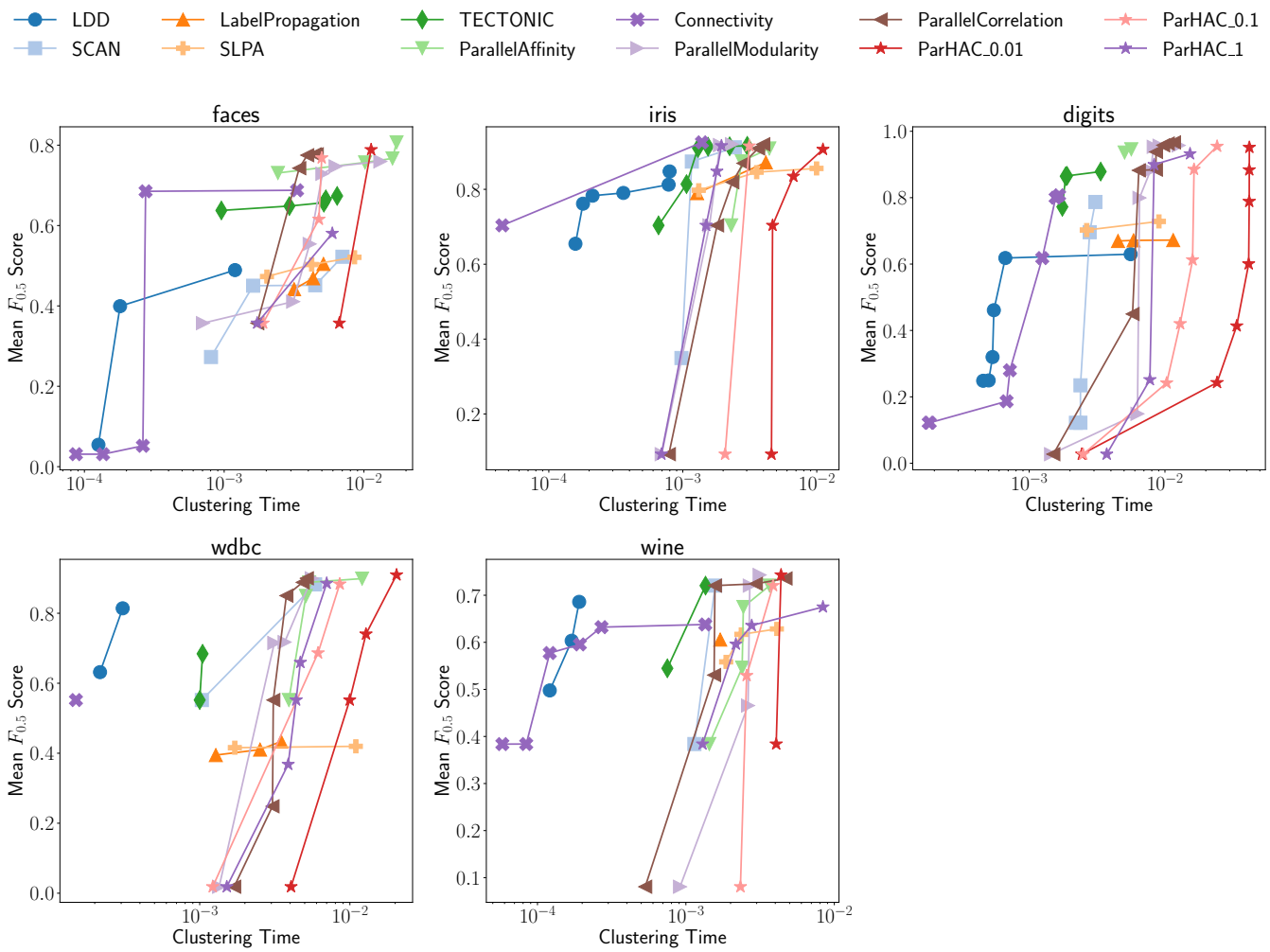

Figure 13: The Pareto frontier of the $F_{0.5}$ and runtime for the weighted UCI k-nearest neighbor graphs $k = 10$.

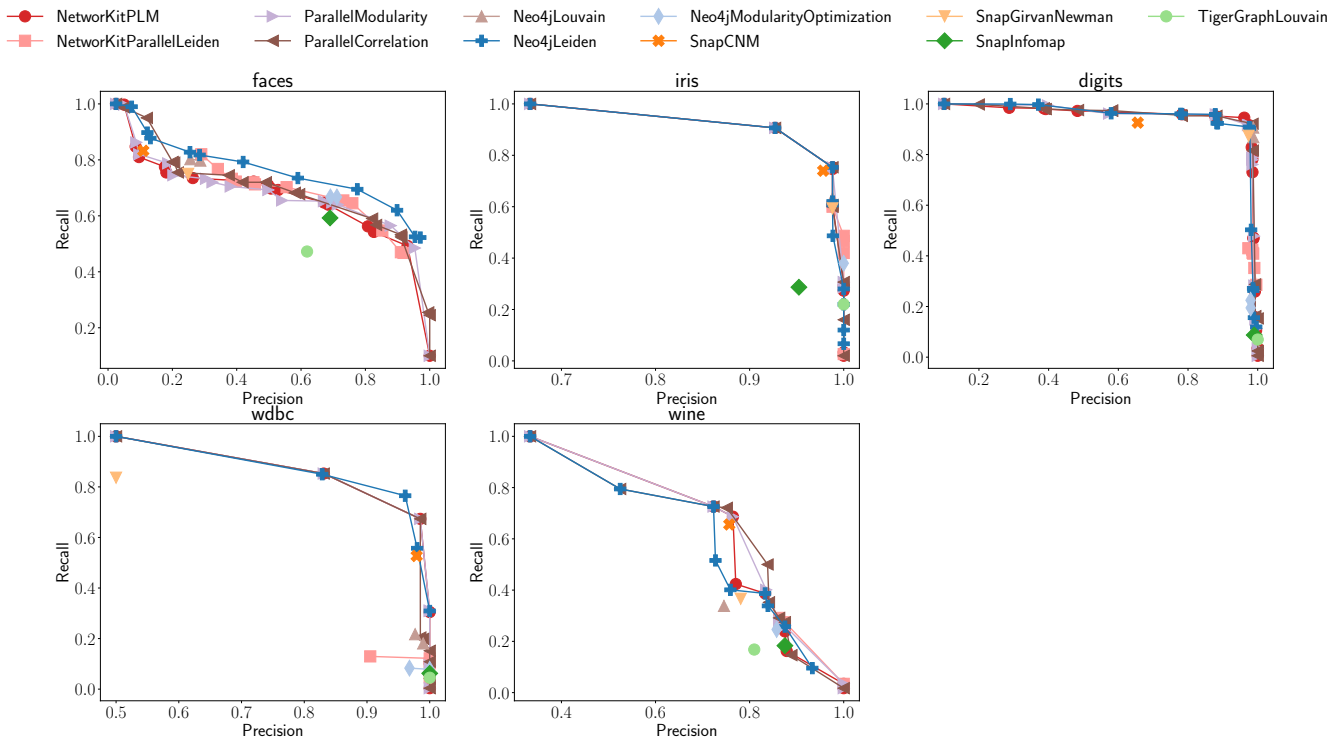

Figure 14: The Pareto frontier of the precision and recall for the weighted UCI k-nearest neighbor graphs ($k = 10$), using different modularity methods.

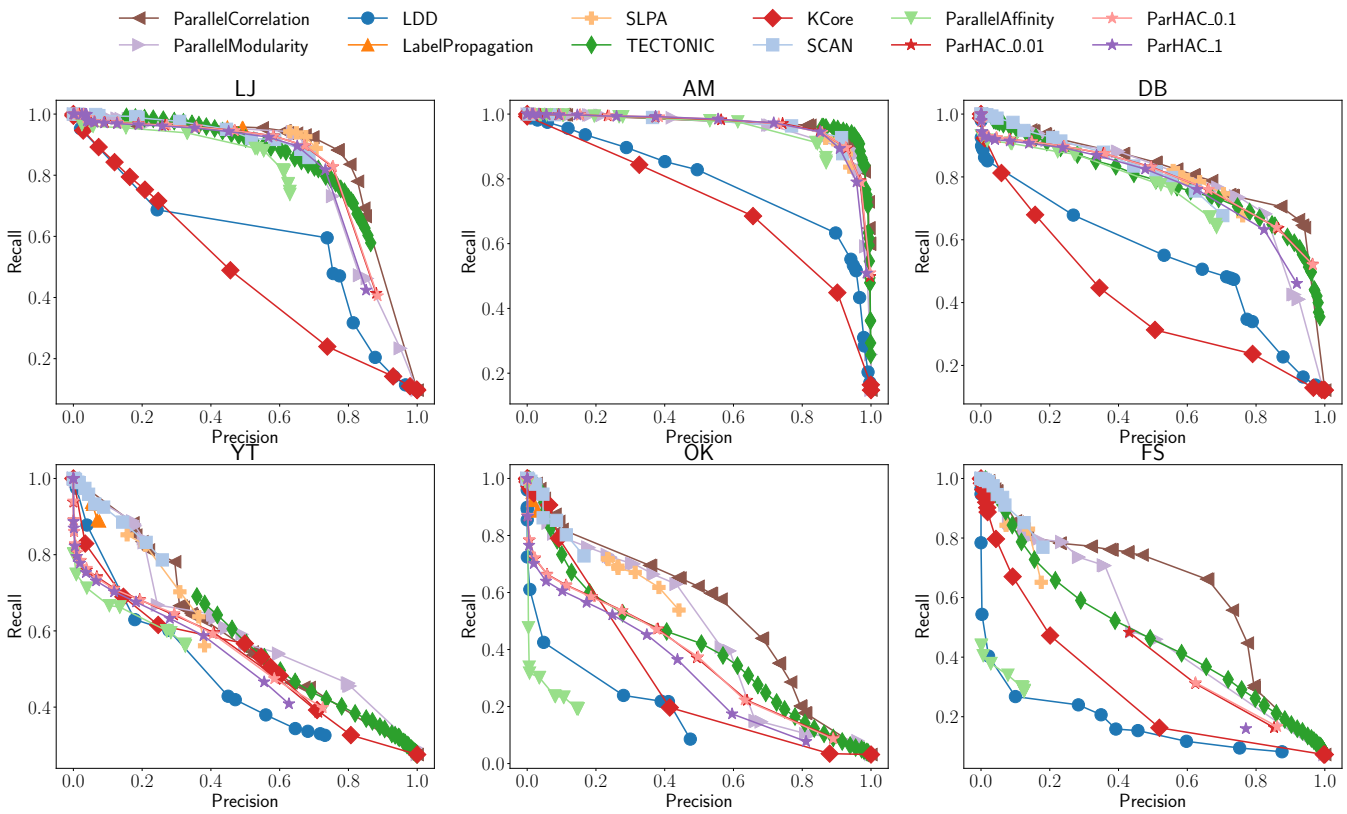

Figure 15: The Pareto fronriter of the precision and recall of the unweighted SNAP graphs.

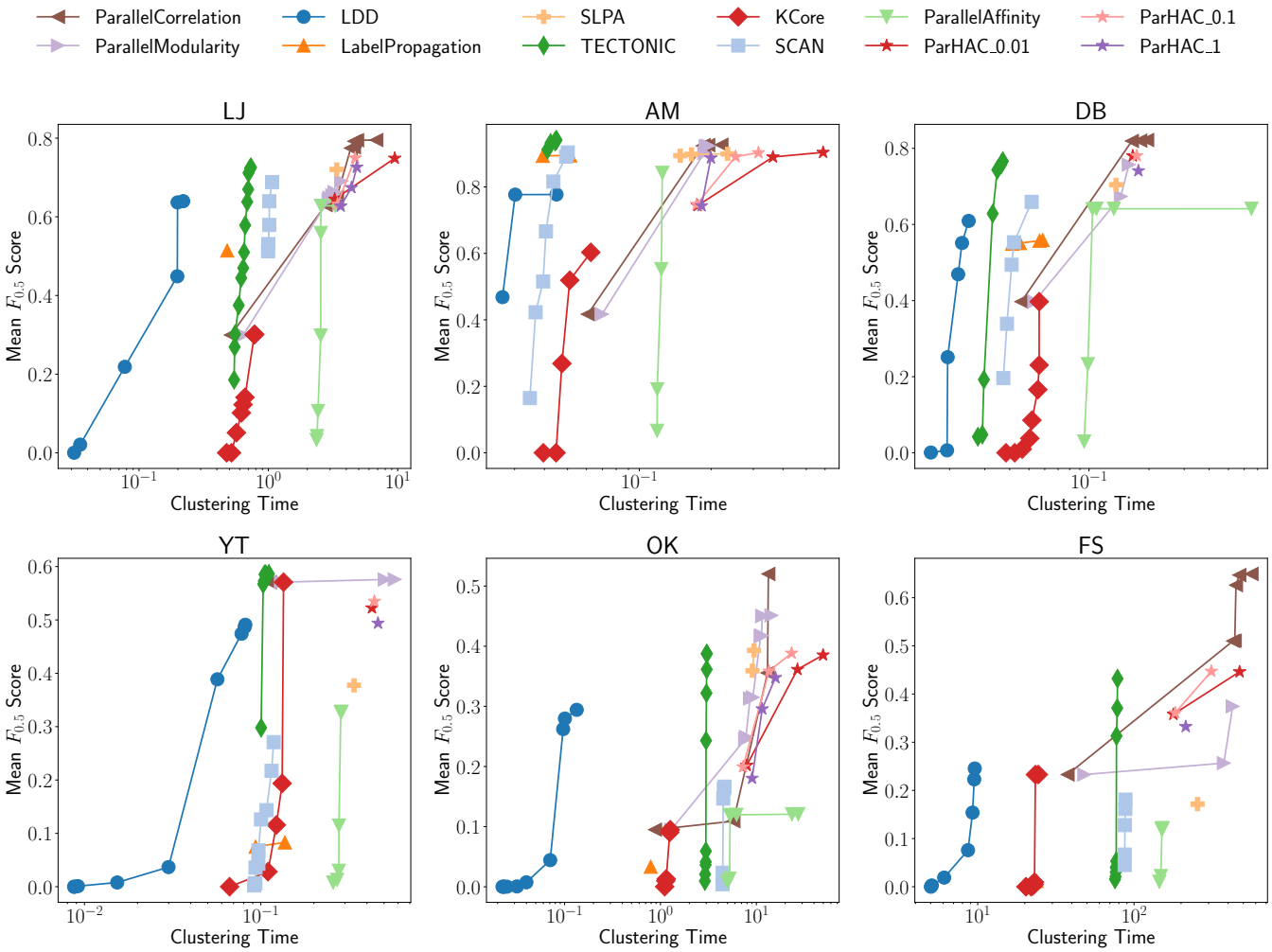

Figure 16: The Pareto frontier of the $F_{0.5}$ and runtime graph for the unweighted SNAP graphs.

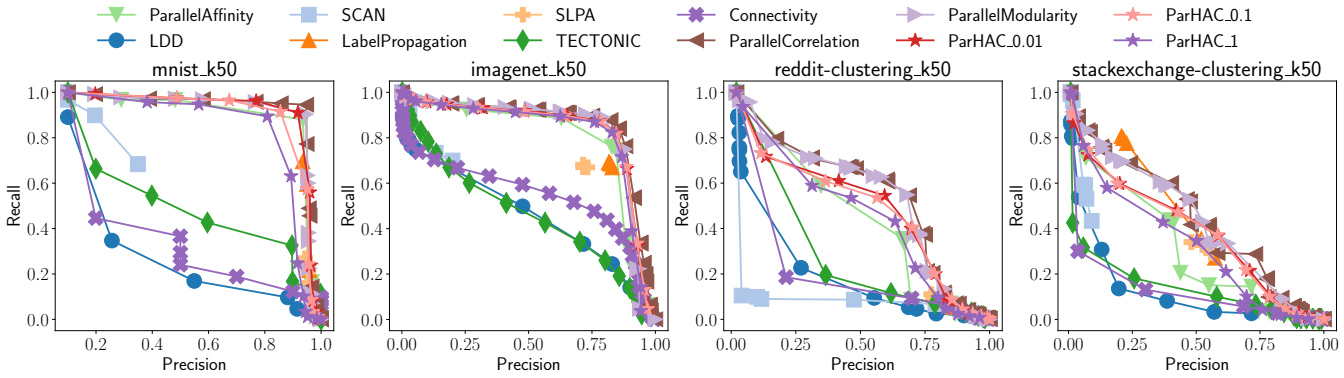

Figure 17: The full Pareto frontier of the precision and recall for the weighted k-nearest neighbor graphs ($k = 50$), using PCBS methods.

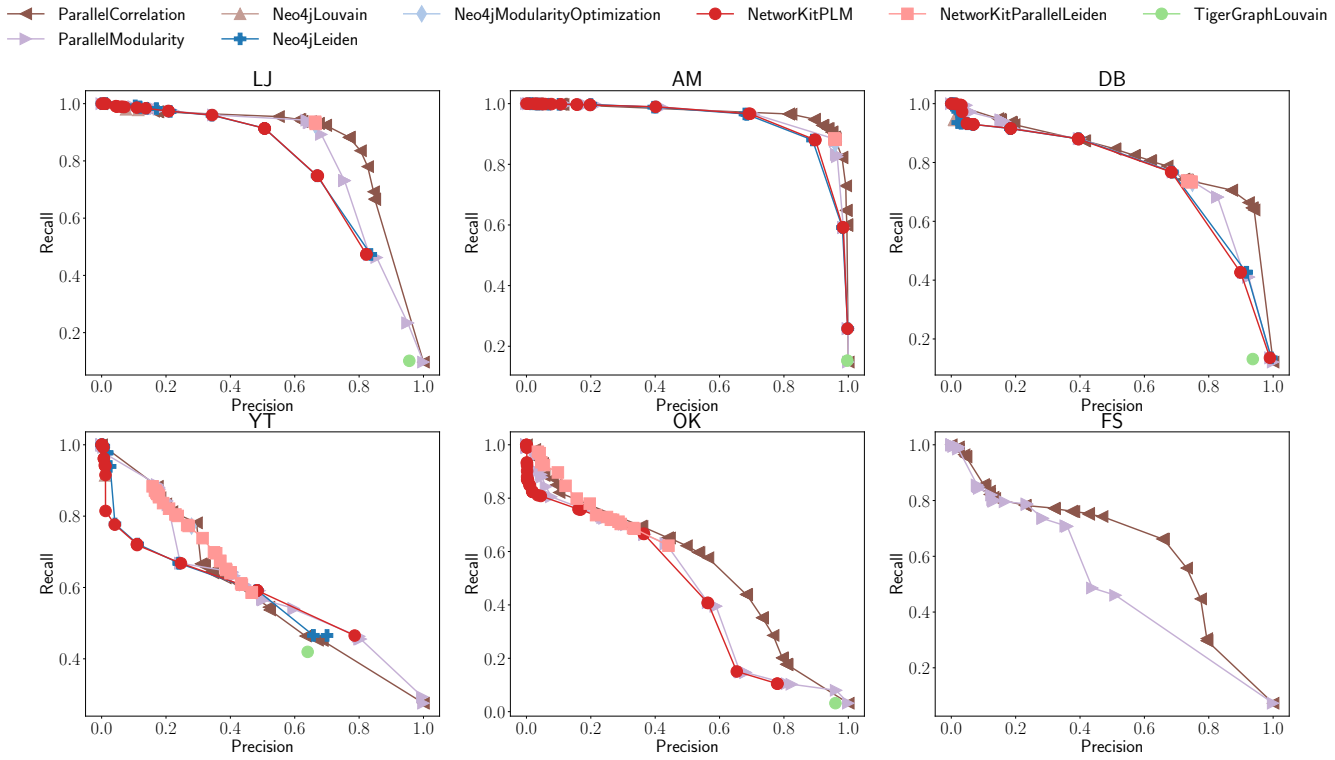

Figure 18: The Pareto frontier of the precision and recall for the unweighted SNAP graphs, using different modularity methods.

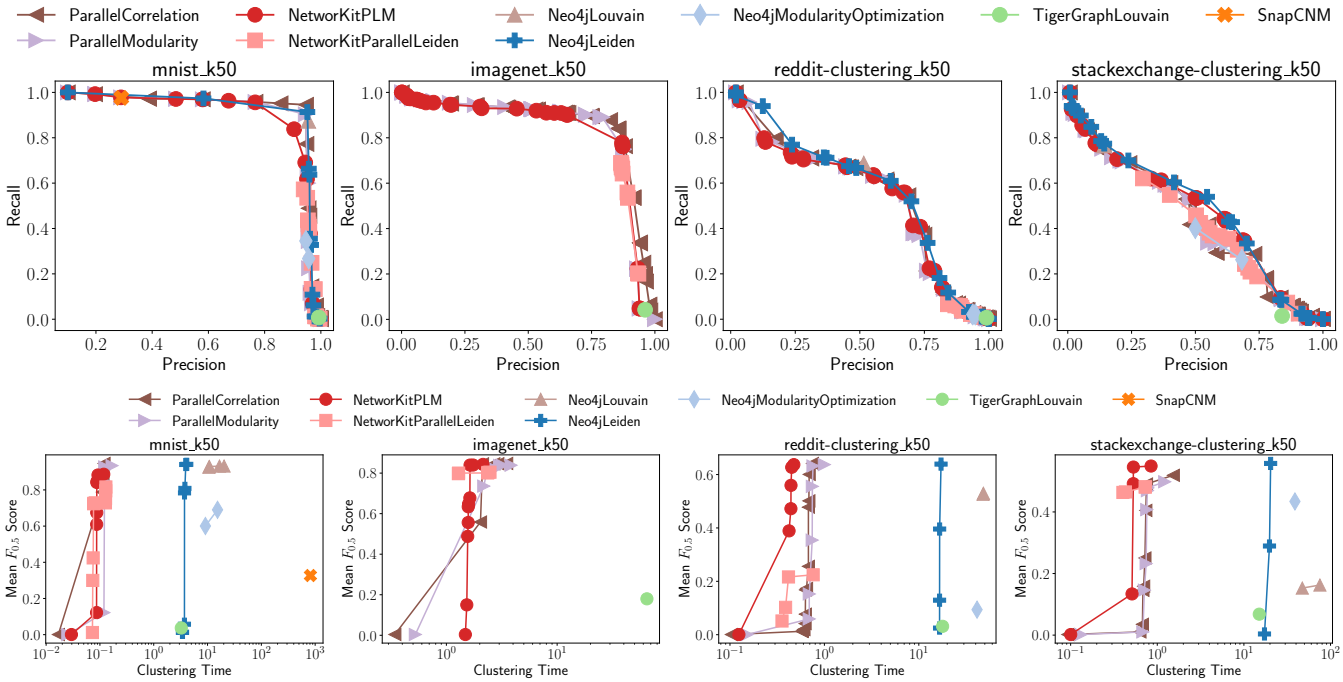

Figure 19: The Pareto frontier of the $F_{0.5}$ and runtime graph for the weighted large k-nearest neighbor graphs ($k = 50$), using different modularity methods.

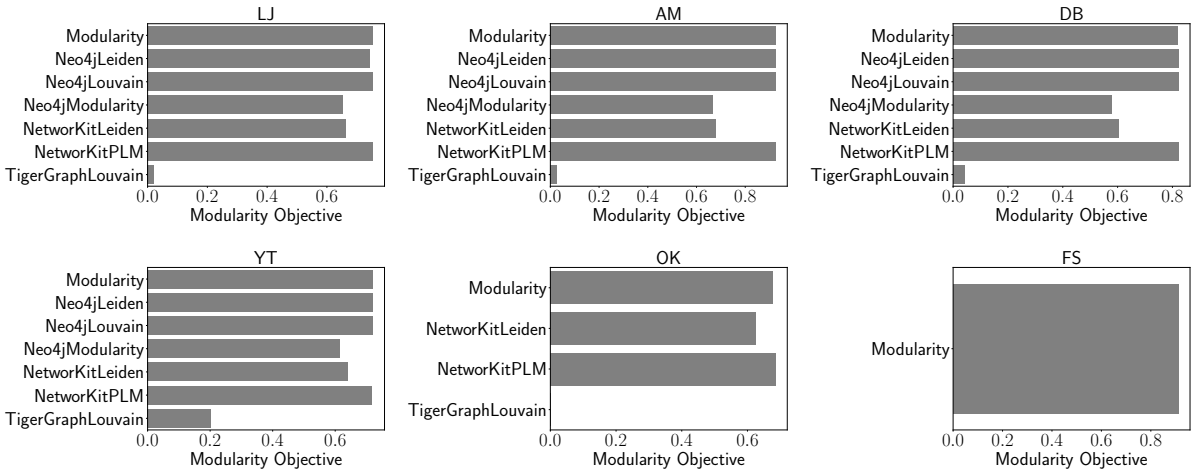

Figure 20: The modularity scores with $\gamma = 1$ for the unweighted graphs, using different modularity methods.