G-Miner: An Efficient Task-Oriented Graph Mining System

Hongzhi Chen
The Chinese University of Hong Kong
hzchen@cse.cuhk.edu.hk

Miao Liu
The Chinese University of Hong Kong
mliu@cse.cuhk.edu.hk

Yunjian Zhao
The Chinese University of Hong Kong
yjzhao@cse.cuhk.edu.hk

Xiao Yan
The Chinese University of Hong Kong
xyan@cse.cuhk.edu.hk

Da Yan
The University of Alabama at Birmingham
yanda@uab.edu

James Cheng
The Chinese University of Hong Kong
jcheng@cse.cuhk.edu.hk

ABSTRACT

Graph mining is one of the most important areas in data mining. However, scalable solutions for graph mining are still lacking as existing studies focus on sequential algorithms. While many distributed graph processing systems have been proposed in recent years, most of them were designed to parallelize computations such as PageRank and Breadth-First Search that keep states on individual vertices and propagate updates along edges. Graph mining, on the other hand, may generate many subgraphs whose number can far exceed the number of vertices. This inevitably leads to much higher computational and space complexity rendering existing graph systems inefficient. We propose G-Miner, a distributed system with a new architecture designed for general graph mining. G-Miner adopts a unified programming framework for implementing a wide range of graph mining algorithms. We model subgraph processing as independent tasks, and design a novel task pipeline to streamline task processing for better CPU, network and I/O utilization. Our extensive experiments validate the efficiency of G-Miner for a range of graph mining tasks.

KEYWORDS

Distributed System, Large-Scale Graph Mining.

1 INTRODUCTION

Graph data exist ubiquitously in a broad range of domains such as social networks, mobile communication networks, financial networks, biological networks and semantic webs. In recent years, we have witnessed a significant increase in the scale of graph data and, concurrently, the growing importance of graph mining - the analysis of large-scale graphs to extract insights. To date, a large number of graph mining algorithms have been proposed, frequent subgraph mining [42], community detection [11], graph clustering [48], graph matching [30], to name a few.

Many distributed graph systems [35, 41], such as Pregel [18], Giraph [3], and PowerGraph [12], have been proposed to process large graphs. These systems follow the *vertex-centric* programming model [18], in which each vertex plays the role of a processing unit that maintains a local state and communicates with its neighbors by message passing (or through shared memory). The vertex-centric model is suitable for the distributed implementation of many graph algorithms such as PageRank, connected components, breadth-first search, etc. These algorithms share a common characteristic: the computation and communication on the vertices are usually light - mostly linear complexity in each iteration.

However, graph mining algorithms are fundamentally different from the aforementioned graph algorithms. Most graph mining algorithms are much more *computation-intensive* and/or *memory-intensive*, with the computational and/or space complexity often growing superlinearly or even exponentially due to the well-known combinatorial explosion problem in the generation of (candidate) subgraphs. While various pruning strategies are commonly applied, even pruning algorithms may have polynomial time complexity.

The vertex-centric graph systems do not consider the characteristics of graph mining algorithms in their design. In particular, the computational model should be more coarse-grained, *subgraph-centric* instead of vertex-centric, since each subgraph now plays the role of a processing unit that maintains a local state and accordingly decides how to involve more vertices for updating. Although a more natural subgraph-centric model has been adopted in a number of recent distributed systems for graph mining (e.g., NScale [22], G-thinker [36], Arabesque [31]), other important elements that are critical to system performance have not been carefully studied. As we will analyze in §2-§3, these systems suffer from problems such as *low CPU utilization* (due to synchronization barrier) and *high memory consumption* (due to subgraph maintenance), both of which are critical issues for processing *computation-intensive* and *memory-intensive* graph mining workloads.

To address the limitations of existing systems, we propose **G-Miner**, based on a re-design of both the computational model and the system architecture. G-Miner adopts the subgraph-centric programming model [22, 36] and presents a general programming framework for expressing a wide range of graph mining algorithms. The key design in G-Miner is to streamline tasks so that CPU computation, network communication and disk I/O can process their workloads without waiting for each other. We summarize our main contributions as follows:

- We analyze several state-of-the-art graph processing systems and identify their limitations on graph mining (§2-§3). We also list the key design criteria for building a distributed graph mining system (§3).

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190545

- We encapsulate the processing of a graph mining job as an independent task, and streamline task processing with a novel *task-pipeline* design, which removes synchronization barrier in existing systems and allows various resources (i.e., CPU, network, disk) to work on tasks concurrently (§4.3).
- We develop G-Miner to realize our designs with various optimizations that further improve the scalability and resource utilization. We also propose two graph-mining specific strategies for load balancing - BDG partitioning for static load balancing and work stealing for dynamic load balancing (§5-§7).

To validate the performance and generality of G-Miner, we implemented five typical graph mining algorithms on G-Miner: triangle counting (TC), maximal clique finding (MCF), graph matching (GM), community detection (CD) [33], and graph clustering (GC) [21]). To the best of our knowledge, no existing graph processing system can handle such a broad range of graph mining problems on large-scale graphs, especially when the vertices have a high-dimension attribute list.

Experimental results on real-world graphs show that G-Miner consistently and significantly outperform existing systems on various performance metrics (e.g., running time, CPU utilization, memory consumption). For simple graph mining problems (e.g., TC) on small graphs, G-Miner is an order of magnitude faster than Arabesque [31]. For the medium-heavy workloads (e.g., GM), G-Miner is 2-6 times faster and uses much less memory than G-thinker [36], which is the only other system that can run. For the heavy workloads (e.g., CD, GC) that no other system can handle, G-Miner still records good performance.

2 RELATED WORK

Vertex/Edge-centric Systems. Most existing graph processing systems, such as Pregel [18], Giraph [3], PowerGraph [12], GPS [24], GraphX [13], Pregel+ [38], GraM [34], PowerLyra [8], GraphD [40], follow the vertex-centric model. In addition to the limitations discussed in §1, it is also difficult to express a graph mining algorithm with the vertex-centric programming paradigm. This is because the vertex-centric programming paradigm requires users to specify the algorithm logic for each individual vertex so that the runtime may execute it in parallel. However, for many graph mining algorithms, they may generate a lot of subgraphs and the algorithm logic is more attached with each subgraph instead of on any single vertex. The edge-centric model is another popular framework for graph processing, mainly adopted in out-of-core systems such as GraphChi [14], X-stream [23], GridGraph [49] and Mosaic [17], which also suffer from similar limitations for handling graph mining workloads as with vertex-centric systems.

Graph-centric Systems. Giraph++ [32] and Blogel [37] proposed the graph-centric model, which partitions the input graph into disjoint subgraphs and assigns them to different nodes. The vertices inside a subgraph send messages to each other through in-memory message buffer while the communication among subgraphs is conducted via the network. However, these systems only aim to reduce the amount of message passing among machines and are still designed to solve the same class of problems as vertex-centric systems.

System	*Cores*	*Mem. (GB)*	*Net. (GB)*	*CPU Util.*	*Time (s)*	*Note*
Single-thread	1	23.29	0	100%	86640.3	Succeed.
Arabesque	#24 x 8	268.43	39.87	74.25%	-	Run over 24h.
Giraph	#24 x 8	NA	26.81	43.25%	x	OOM.
GraphX	#24 x 8	277.71	27.14	11.58%	-	Run over 24h.
G-thinker	#24 x 8	66.68	36.75	16.20%	164.6	Succeed.

Table 1: Performance of max-clique finding ("-": >24 hours; "x": job failed as out of memory)

Thus, they have similar problems in expressing a graph mining algorithm and their system designs are not suitable for graph mining.

Others. NScale [22] was designed to solve graph mining problems using the MapReduce framework. It proposed a neighborhood-centric model, in which a k-hop neighborhood subgraph of an interest-point is constructed with k rounds of Map-Reduce and each round of Map-Reduce extends the 1-hop new neighbors. Once all the candidate subgraphs have been constructed, NScale executes a final round of Map-Reduce job to verify each candidate. However, the total number of all candidate subgraphs to be constructed can be very large, and the overhead of MapReduce is also high. G-thinker [36] extends the neighborhood-centric model of NScale to a subgraph-centric model, which regards each growing subgraph in the mining process as the basic processing object. This fine-grained computational model, together with in-memory processing, enables G-thinker to achieve better performance than NScale. However, G-thinker still follows a batch processing framework to execute the computation and communication parts of a job in batches, which makes it hard to fully utilize the CPU and network resources. Arabesque [31] proposed an graph exploration model with the concept of *embedding*. Specifically, the exploration proceeds in rounds, where in each round the existing embeddings are expanded by one neighboring vertex or edge. The newly generated embedding (called *candidate*) is further processed by a *filter* function for pruning. Limited by its MapReduce-based framework, the pruning step is only executed after the exploration steps, which can generate a large number of candidates and thus waste a substantial amount of computation and memory on invalid embeddings. Compared with G-Miner, a fundamental design in these systems is that they all adopt a batch processing framework, which results in synchronization barrier. The barrier cannot be easily removed unless they abandon batch processing, which may lead to a complete system re-design. In addition, they also do not consider dynamic load balancing such as work stealing in G-Miner.

3 MOTIVATION

In the previous section, we have briefly analyzed the pitfalls in the designs of existing systems for solving graph mining problems. To demonstrate the limitations of existing systems and motivate the design of G-Miner, we tested the state-of-the-art graph mining systems, Arabesque [31] and G-thinker [36], as well as two popular vertex-centric systems, Giraph [3] and GraphX [13], using 8 nodes in our cluster (see configuration in §8). The workload is maximum clique finding and the dataset is Orkut (details in Table 2). We did not include NScale [22] because it is not open source. The performance of a single-threaded implementation is provided as a baseline.

We report the results in Table 1, from which we can identify the following problems with these systems.

Low CPU utilization. The single-threaded implementation took 86,640 seconds with 100% CPU utilization, which shows that graph mining is CPU-intensive (note that Orkut is only a medium-sized graph). However, most of the distributed systems have low CPU utilization. G-thinker's low CPU usage is due to its batch processing between communication and computation as explained in §2, while that of the other systems are mainly caused by their bulk synchronous parallel (BSP) mechanism.

High memory consumption. Arabesque, Giraph and GraphX all consume a large amount of memory as they need to construct all the 1-hop neighborhood subgraphs before the start of computation. The high memory consumption may even cause a system to run out-of-memory (OOM) on a medium-sized graph and hence severely limits the scalability.

Mismatch of computational model. The performance problems of the above systems are mainly due to their computational models, which in turn lead to unsuitable system designs for graph mining. As we briefly explained in §2, the synchronization barrier these systems inherited from the BSP model exacerbates the straggler problem as graph mining problems usually have a heavy workload. Extending these systems to include an asynchronous model will require fundamental changes to their system design and implementation, which would likely change them into completely new systems.

To conclude, the bottlenecks for processing graph mining workloads are intensive CPU computation and high memory consumption. Our design addresses these bottlenecks with *(1) a computational model specialized for solving graph mining problems; (2) fully utilizing all CPU cores by removing synchronization barriers, and hiding network communication and disk I/O overheads; (3) bounded memory consumption to avoid OOM; (4) transparent load balancing and fault tolerance.*

One interesting observation is that G-thinker has superlinear speedup compared with the single-threaded implementation (528x speedup with 192 cores). This is because the *global currently-maximum* clique is used to prune unpromising search space in each local worker. Thus, parallel maximum clique finding enjoys not only the aggregate computing power of the machines, but also a faster reduction in the search space in parallel. In other words, here we have two factors that lead to the superlinear speedup: (1) parallel computing power, and (2) parallel pruning. In the case of maximum clique finding, (2) apparently leads to more speedup. As such pruning is common in graph mining algorithms, this interesting result reveals an extra benefit of distributed computing for graph mining.

4 SYSTEM DESIGN

To address the bottlenecks of existing systems identified in §3, we propose a novel **task-pipeline** design that allows CPU computation, network communication, and disk I/O to be processed asynchronously throughout the entire job process, so that the communication and disk I/O overheads can be hidden within the cost of CPU computation. We first present a general graph mining schema and a basic task model, which are necessary for illustrating the idea of the task-pipeline.

Graph notations. A graph is represented by $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each vertex $v \in V$ has an ID $id(v)$, an adjacency list $\Gamma(v)$ that keeps the set of neighbors of v, and an optional attribute list $a(v)$ that is usually a vector storing a user's property such as age, sex and location. For simplicity, our discussion focuses on undirected graphs, though our system can also handle directed graphs.

4.1 General Graph Mining Schema

G-Miner aims to provide a unified programming framework for implementing distributed algorithms for a wide range of graph mining applications. To design this framework, we need to first identify a common pattern for existing graph mining algorithms.

We focus on the following five categories of typical graph mining problems: *(1) subgraph/graphlet enumeration (e.g., triangles [26], cliques [6], quasi-cliques [1], size-k graph- lets [2]); (2) subgraph matching [30] (i.e., listing all occurrences of a set of query subgraphs); (3) subgraph finding (e.g., maximum clique finding [5], densest subgraph finding [10], etc.); (4) subgraph mining (e.g., frequent graph mining [43], community detection [11], correlated subgraph mining [28], etc.); (5) graph clustering [25, 48].*

By carefully studying the above mentioned graph mining problems, we devise a general computing schema as follow. The graph mining job usually starts from a set of seed subgraphs (typically initialized as individual seed vertices), which are selected by the *init* operation based on some initial conditions, and then recursively performing an *update* operation on each subgraph g, which may grow (by adding neighbor vertices), shrink (by pruning some parts), split (into more subgraphs), delete or report g (i.e., g is not or is a match). Normally, the computation of *update* has high complexity (e.g., polynomial) and is closely dependent on the intermediate subgraph.

4.2 Task Model

To support asynchronous execution of various types of operations (i.e., CPU, network, disk) and efficient load balancing, we model a graph mining job as a set of *independent tasks*. A *task* consists of three fields: (1) a *subgraph* g, (2) *candidates*, and (3) *context*. The computation of a task proceeds in rounds. In each round, the task accesses these three fields and updates them according to the user-defined algorithm logic. During the process, g is used to keep or update the topology of the intermediate subgraph generated in each round, until it is reported as a result or deleted (i.e., no result can be produced from g). The *candidates* records the IDs of the candidate vertices that will be used to update g in the next round. The *context* is used to hold other essential information (e.g., the current round number, the count of matched patterns).

The candidates are usually generated from the 1-hop neighbors of g, i.e., $\Gamma(v)$ of each v in g. A filter function, which implements pruning strategies in a graph mining algorithm, can be applied to exclude irrelevant neighbors in order to reduce the search space. If a candidate vertex v is not in the local machine (either the local graph partition or local cache), we will pull v with the associated data (e.g., $\Gamma(v)$, $a(v)$) from a remote machine. To avoid repetitive pulling, we use a local cache to store the remote v.

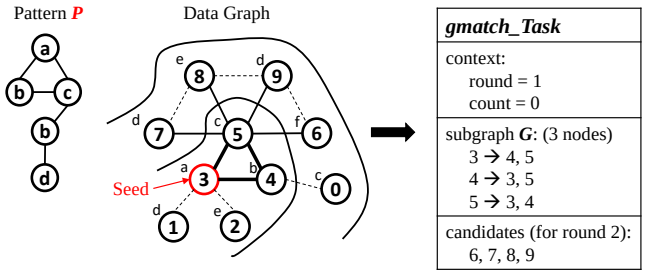

Figure 1: Task model for graph matching

The task model is inspired by the task concept in G-thinker [36] (also implicitly adopted in NScale [22]), we model it as an independent graph mining object for asynchronous execution with the task-pipeline (§4.3) and load balancing with task stealing (§6.2).

Example. We use graph matching to illustrate the concepts introduced so far. As shown in Figure 1, we generate a task from the seed vertex, $v3$, as $v3$ matches the label '*a*' of the root of the query pattern. Accordingly, subgraph g is initialized as $v3$, and *candidates* is $\{v1, v2, v4, v5\}$. In round 1, g grows from $v3$ to include $v4$ and $v5$ since the labels of $v4$ and $v5$ are '*b*' and '*c*', matching the labels of the next level in the query pattern; while vertices $v1$ and $v2$ are filtered. Then, *candidates* is updated to be $\{v6, v7, v8, v9\}$, since filtering removes all vertices except the neighbors of the vertex with label '*c*', i.e., $v5$, which will be matched in the next round. As for the *context* field, we set the current *round* number to 1, and *count* is used to record the number of subgraphs matched with the query pattern in this task.

Task lifetime. Each task has a lifetime, which starts from the initialization from a seed vertex and ends at the completion of the processing on its subgraph. During its lifetime, a task takes one of the following four statuses: *active*, *ready*, *inactive* and *dead*. A task is *active* when it is currently being processed by the *update* operation. Then, the *candidates* in the task will be modified. If there is at least one vertex in *candidates* that needs to be pulled from a remote machine, we convert the status of the task to *inactive*. The task status will be changed to *ready* when all its remote candidates are pulled, i.e., now the task is ready to be processed. Note that if a task has no remote vertex in the *candidates* in the current round, it will directly enter the next round of *update* without any status change. Task processing in G-Miner has no barrier among workers or within a worker. A task is *dead* when either the results have been found and reported, or it is confirmed that no result can be found. G-Miner then deletes the dead task and releases all the resources taken by it.

4.3 Task-Pipeline

Now we present the task-pipeline, which is designed to asynchronously process the following three major operations in G-Miner: (1) *CPU computation* to process the *update* operation on each task, (2) *network communication* to pull *candidates* from remote machines, and (3) *disk writes/reads* to buffer those intermediate tasks on local disk.

We need (3) in order to bound the memory usage for the following reason. Most subgraph/graphlet enumeration or counting problems may generate an exponential number of candidate subgraphs,

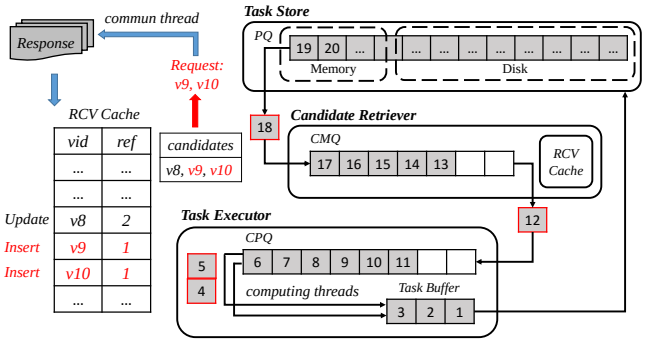

Figure 2: Task-Pipeline

while subgraph mining problems suffer from the well-known combinatorial explosion problem. Thus, there can be a large number of newly generated tasks to be processed in order to keep the CPU cores busy for high efficiency, but this may exhaust the available memory.

The task-pipeline consists of three components: *task store*, *candidate retriever*, and *task executor*, as illustrated in Figure 2. The task store manages all inactive tasks with a priority queue, the candidate retriever handles the pulling and caching of remote candidates, and the task executor is to execute the mining jobs of active tasks. We describe more details and how they interact with each other as follows.

Task Store. The task store manages all inactive tasks in a local worker. As different tasks may request common candidate vertices from remote machines, we use a local cache to avoid pulling these remote vertices repeatedly. However, a simple cache strategy may have poor hit rate as we illustrate in Figure 3. Assume that the cache has $size = 3$, and vertices with $vid \geq v8$ are in remote machines. We first process $Task_1$, and in round 2 we need the remote candidates $\{v8, v9, v10\}$. After pulling them, we insert $\{v8, v9, v10\}$ into the local cache. Then, $Task_2$ needs to pull $\{v12, v13\}$, which refreshes the cache to $\{v12, v13, v10\}$. Next, $Task_3$ needs to pull $\{v8, v9, v14\}$ in round 3, and refreshes the entire cache. $Task_4$ requires $\{v8, v12, v13\}$, but only $v8$ is in the cache. Thus, the total hit number of the cache is only 1 (i.e., $v8$) for executing these 4 tasks.

To improve the hit rate and hence avoid repetitive remote vertex pulling, we propose a *task priority queue* to order tasks such that tasks with common remote candidates are kept near each other. For example, at the bottom of Figure 3 we order $Task_3$ next to $Task_1$ based on their common remote candidates $v8$ and $v9$. When we process tasks in this order, the final hit number of the cache is improved to 5, as shown in Figure 3. The details about how we order tasks in the task priority queue will be presented in §7.

To bound the memory usage, the task store keeps a subset of higher-priority tasks in memory, while the remaining tasks are kept on local disk. The use of this disk-resident data structure does not affect the performance of G-Miner as we will explain at the end of this section.

Candidate Retriever. The candidate retriever handles the tasks dequeued from the task store and prepares the remote vertices in their *candidates* fields by pulling. Specifically, for each task, the

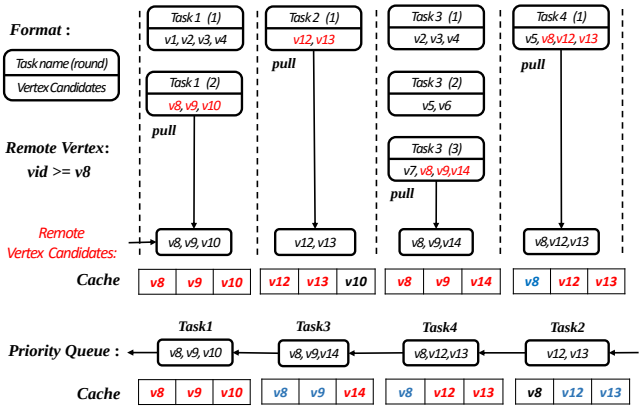

Figure 3: Task Priority Queue

candidate retriever first checks if there are any remote vertices already existing in the local cache. Then, it issues pull requests for those vertices that are not in the cache. After that, this task will be inserted into the *communication queue* (*CMQ*) waiting for the pull responses. Once all remote candidates for a task are ready (i.e., already pulled and accessible in the cache), the status of this task will be converted to *ready* to indicate it can be dequeued from the *CMQ* for further processing.

We also propose a *Reference Counting Vertex Cache* (*RCV Cache*) to manage those remote vertices obtained by pulling. The design utilizes the properties of the task-pipeline and task priority queue. When a remote vertex is inserted into RCV Cache, it has a high probability to be accessed again by the subsequent tasks in the pipeline, as enabled by the design of the task priority queue. RCV Cache keeps a reference count for each cached vertex to record the number of *ready* or *active* tasks referring to it. This count will be updated once a referred task changes its status to other cases. Such reference count plays an important part in our cache update strategy (see §7).

Task Executor. All *ready* tasks from the candidate retriever will be inserted into a *computation queue* (*CPQ*) managed by the task executor. The task executor consists of a pool of computing threads to process tasks in the CPQ in parallel. Each computing thread executes the *update* operation on the task, and after that checks if the status of the task should be changed to *inactive* or *dead*.

If the task becomes *inactive*, then the computing thread inserts it into a *task buffer* maintained by the task executor. The tasks in this buffer are inserted into the task store in batches, as batch processing can gather those tasks with common remote candidates together in advance. If the task remains to be *active*, meaning that all vertices in the updated *candidates* are already in RCV Cache and/or local graph partition, then the computing thread simply proceeds to execute it for the next round.

Example. Figure 2 illustrates the design of our task-pipeline. The communication thread of the candidate retriever dequeues task T_{18} from the priority queue of the task store, and is now handling the vertex pulling for T_{18}. T_{18} has 3 remote vertex candidates $\{v_8, v_9, v_{10}\}$, but v_8 is already in RCV Cache. Thus, the communication thread only requests v_9 and v_{10} from remote machines, and inserts T_{18} into the CMQ. When v_9 and v_{10} are pulled, they are inserted into RCV Cache and their *ref* count is initialized as 1 (while *ref* of v_8 was incremented to 2). Meanwhile, T_{18} is set as *ready* and will be inserted into the CPQ. The task executor dequeues T_4 and T_5 from the CPQ, and assigns them to available computing threads. They will be inserted into the task buffer if vertex pulling is needed.

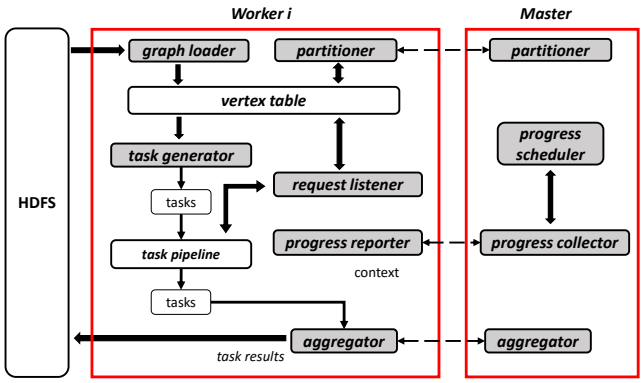

Figure 4: System Architecture

The task-pipeline not only streamlines task buffering and storing to disk, communication to obtain remote candidates, and task computation, but also allows disk I/O, network, and CPU cores to do their own work concurrently. In this way, the overheads of the disk I/O and network communication can be hidden in the higher cost of CPU computation. Note that for some problems, if the communication (or disk I/O) overhead is high, we can re-allocate the threads to the candidate retriever and the task executor proportionally.

5 SYSTEM ARCHITECTURE AND API

We now present the system architecture of G-Miner, introduce its API and demonstrate how to use the API with an example.

5.1 System Architecture and Components

G-Miner adopts a master-slave shared-nothing architecture, as shown in Figure 4. One node in the cluster serves as the master, and is in charge of graph partitioning, task stealing, scheduling, etc. Other nodes play the role of slaves to process tasks. By default we deploy only one G-Miner process (i.e., one worker) in each slave node in the cluster to enable cache sharing by all the cores, which maximizes both CPU utilization and cache efficiency. G-Miner also supports multi-process deployment in each slave node, but there is no cache sharing among threads from different processes in the same node.

We use HDFS as the underlying persistent storage. Each worker W_i loads a piece of graph data P_i by the *graph loader* and stores the vertices, including their state $(id(v), \Gamma(v), a(v))$, into its local *vertex table*. Then, the *partitioner* in the master communicates with the *partitioner* component in each worker to obtain the graph metadata of P_i, and executes a specific partitioning strategy to re-distribute the vertices to workers.

A graph mining job starts from task generation by the *task generator*. The task generator scans the vertex table to select the *seed vertices*, and then generates one task for each seed. These tasks are fed into the task-pipeline.

```
template <class KeyT, class ContextT, class AttrT>
class Task {
   typedef Vertex<KeyT, AttrT> VtxT;
   Subgraph<KeyT, AttrT> subG;
   ContextT context;
   vector<KeyT> candVtxs;
   void pull(vector<KeyT>& candVtxs);
   virtual void update(vector<VtxT> &candVtxObjs)=0;
};
template <class TaskT>
class Worker {
   typedef TaskT::VtxT VtxT;
   virtual VtxT vtxParser(string s)=0;
   virtual void output()=0;
   virtual TaskT init(VtxT v)=0;
};
template <class ContextT>
class Aggregator {
   virtual void agg(ContextT &context)=0;
};
```

Listing 1: API of G-Miner

An *aggregator* may be used to access the context of each task for global communication and monitoring. For example, for maximum clique finding, a maximum aggregator can be used to record and notify all workers of the current maximum size of a clique found globally, which helps local pruning by each worker.

Each worker also has a *request listener* to handle requests for vertex pulling or tasks stealing from other workers. To implement task stealing, each worker has a *progress reporter* that sends its local progress to the master periodically, while the master uses a *progress collector* to receive the reports. Thus, the master maintains a global view of the workers' progresses, which is used by the *progress scheduler* to facilitate dynamic migration of tasks from busy workers to idle workers.

5.2 G-Miner API

Following the general graph mining schema (§4.1) and task model (§4.2), G-Miner provides an user-friendly API to express a broad range of graph mining algorithms. To write a G-Miner program, users only need to subclass two predefined classes, *Task* and *Worker*, to implement two key virtual functions, *init()* and *update()*, for their specific mining problems (see Listing 1).

The three template arguments of *Task* class (i.e., *KeyT*, *AttrT*, *ContextT*) define the data type of vertex ID, attribute and context. Each *Task* uses three variables {*subG*, *candVtxs*, *context*} to maintain the three fields in the task model. Users override the *update()* function to implement the graph mining algorithms and accordingly update *subG* and *context* by accessing the vertex instances *candVtxObjs* pointed by *candVtxs*, and then reset it through *pull*() for the next round.

The *vtxParser()* and *output()* functions in *Worker* class define how to load and parse a vertex from HDFS into memory and how to output the results to HDFS, respectively. And the *init()* function handles seed selection and task generation. Moreover, users may also implement an *agg* method in *Aggregator* class to specify how to apply global aggregation periodically based on the results in *context*.

5.3 Programming With G-Miner

G-Miner's API allows us to implement a wide range of graph mining algorithms, for example, the five categories of algorithms listed in §4.1. We implemented five graph mining algorithms for performance evaluation in §8.2 (all code will be released). We also demonstrate how to implement a basic graph matching algorithm in Listing 2.

```
class GMTask: public Task<int, pair<int, int>, char>{
   void update(vector<VtxT> &candVtxObjs){
      // context is a pair of (round, count).
      r = context.round;
      S = match(candVtxObjs, labels at level r);
      if (!S.empty()) {
         subG.addNodes(S);
         //to set new candVtxs based on S
         candVtxs = ...
         if (r == max depth of pattern)
            context.count += # of matched patterns;
         else
            pull(candVtxs);
      }
   }
};
class GMWorker: public Worker<GMTask> {
   VtxT vtxParser(string s) { /* parse vertex */ }
   void output() { /* dump results to HDFS */ }
   GMTask init(VtxT v) {
      if (v.label == pattern.root.label) {
         t = new GMTask;
         t.subG.addNode(v);
         addTask(t);
      }
   }
};
```

Listing 2: Graph matching implementation on G-Miner

GMTask inherits the *Task* class by setting *KeyT* and *AttrT* as *int* and *char* to represent the vertex ID and label, respectively; while *ContextT* is defined as a pair of integers, (round, count), to record the current round and the count of matched subgraphs. In each round of *update*(), *GMTask* matches one level of pattern graph with *candVtxObjs* based on their labels by the function *match*(). The return value *S* of *match*() will be added into *subG* if *S* is not empty. After that, *GMTask* picks its new candidates from the 1-hop neighbors of *subG* by filtering out those vertices with wrong labels.

Before *update*() is invoked, the *init()* function in *GMWorker* computes the seed vertices whose labels match with the root of the pattern graph and then generates the corresponding tasks. In addition, the *agg()* function in *Aggregator* can be simply implemented as a general sum aggregation that reads the context.count to aggregate the global count of matched subgraphs in the data graph (omitted due to space limitation).

6 LOAD BALANCING

G-Miner supports both static load balancing (by graph partitioning) and dynamic load balancing (by task stealing).

6.1 BDG Partitioning

Most existing graph systems adopt random hashing as the default partitioning algorithm. However, as the computation of most graph mining problems focuses on local subgraphs, keeping the locality in the input graph can avoid expensive remote candidate pulling. To this end, we propose a *Block-based Deterministic Greedy* (*BDG*) partitioning, though G-Miner also allows users to implement their own partitioning strategy.

BDG first cuts an input graph into fine-grained blocks to avoid breaking the locality and then assigns these blocks to different workers according to a greedy algorithm. We obtain the blocks by

a multi-source distributed BFS. Each source is assigned a distinct color, and broadcasts its color to its neighbors. Each uncolored vertex, upon receiving the colors from its neighbors, sets its color to be one of the received colors, and then broadcasts its color to its neighbors. Vertices with the same color then form a block. To limit the size of a block, we set the number of steps taken by BFS from each source to a small value, and repeat the above process until all vertices are colored. The source vertices are randomly selected from the entire graph. In some cases when there are many tiny connect components (CCs) in the graph, selecting sources by sampling may not be effective. However, this can be easily fixed by running a CC finding algorithm (e.g., Hash-Min [39]) on the uncolored vertices after a few rounds of BFS coloring, and then simply consider each CC as a block.

We then apply a deterministic greedy algorithm on these blocks, which is motivated by [29]. Assume that we want to partition a graph into k parts for k workers and each partition has an expected capacity $C = |V|/k$. Let $\Gamma(B)$ be the 1-hop neighbor blocks of the block B, and $P(i)$ be the set of vertices which belong to the blocks that have already been assigned to partition i. Then, for each unassigned block B, we assign it to partition j based on Eq. 1.

$$j = \arg\max_{i \in [k]} \{|P(i) \cap \Gamma(B)| * (1 - \frac{|P(i)|}{C})\} \quad (1)$$

The block-based deterministic greedy strategy guarantees that each B is distributed to partition j whose $P(j)$ has high overlap with B and still has sufficient free capacity to take B. As the algorithm follows a greedy strategy, the assigning order of blocks can affect the final partitioning quality. We sort the blocks in descending order of their sizes and then start the assignment from the largest block.

6.2 Task Stealing

Although BDG partitioning provides static load balancing by balancing the number of vertices in each worker and preserving the locality of the partitioned graph data, the workload of a specific task is related to many factors (e.g., $|candVtxs|$, $|subG|$) and hard to predict. Thus, the workload distribution among each worker can be quite imbalanced. To address this issue, we propose a dynamic task stealing mechanism in order to improve the overall performance of G-Miner.

The progress reporter on each worker regularly reports the number of tasks with different statuses in the task-pipeline to the master, and the progress listener in the master maintains a global *progress table*. When a worker W_i completes all its local tasks, it sends a *REQ* message to the master to request more tasks to execute. The master checks its progress table for the most heavily loaded worker, W_j, and sends a *MIGRATE* message to W_j. Then, W_j migrates T_{num} tasks to W_i from its task priority queue, if the number of its inactive tasks is larger than T_{num}; otherwise it directly sends a *No_Task* message to W_i. The tasks are migrated in batches of size T_{num} for efficient network transmission.

While task independence in our task model design allows tasks to be easily migrated from one worker to another, task migration itself may involve expensive network communication. If we migrate a task whose *subgraph* is large, the extra migration cost may offset the benefit gained from task stealing. Similarly, if a task has high dependency on the local graph partition P_j, after migrating to W_i, it will incur costly communication for vertex re-pulling from W_j. To address the above issues, we define a cost function $c(t)$ to measure the cost of migrating task t, as well as a local rate $lr(t)$ to measure the dependency of t on its local graph partition, as shown in Eq 2 and Eq 3.

$$c(t) = |t.subG| + |t.candVtxs|, \quad (2)$$

$$lr(t) = \frac{|t.candVtxs| - |t.to\_pull|}{|t.candVtxs|}, \quad (3)$$

where $|t.candVtxs| - |t.to\_pull|$ gives the number of local vertices referred by t's current subgraph. Then, two thresholds, T_c and T_r, are set respectively to decide whether a task t can be migrated (i.e., $c(t) < T_c$ and $lr(t) < T_r$).

7 SYSTEM IMPLEMENTATION

We give more details on the implementation of some key components of G-Miner. We also discuss how G-Miner handles fault tolerance.

Task Priority Queue. To order inactive tasks that share common remote candidates to be near each other, we apply *locality-sensitive hashing* (*LSH*) [7, 9, 20] to generate a key for each task, based on the IDs of its remote candidates (denoted as *to_pull*) before pushing it into the task priority queue. This is motivated by the subgraph shingles in [22, 36]. LSH reduces each high-dimension *to_pull* data into a low k-dimension vector *key* and maps similar *to_pull*s to the same key. The task priority queue orders tasks by their keys, so that successively dequeued tasks share common *to_pull* data.

Tasks in the task priority queue are stored as a set of disjoint blocks with fixed capacity. To bound memory consumption, we only maintain the head block in memory but keep the remaining ones on disk. Each block has an index to indicate the range of tasks in it, which can be used for block loading into memory and new task insertion. When all tasks in the first block have been consumed by the task-pipeline, we load the next block into memory. This design allows the disk I/O cost to be hidden in the higher computation cost of subgraph-centric tasks.

RCV Cache. Task processing in the task-pipeline can be viewed as a task stream, and we need to guarantee that all the active tasks can find their remote candidate vertices in RCV Cache. The traditional FIFO or LRU caching strategy may cause some vertices to be replaced, in which case the missing vertices need to be re-pulled and the referred tasks cannot be processed.

RCV Cache maintains a reference count r for each vertex in it, which records the number of active tasks referring to the vertex. When a task is dequeued from the task priority queue, we check if each of its remote candidates is in RCV Cache. If yes, the reference of the vertex increments by 1; otherwise, the vertex will be pulled from a remote worker and inserted into RCV Cache with $r = 1$. When this task completes a round of computation, the reference of all its referred vertices in RCV Cache decrements by 1. Once the reference of a vertex becomes 0, we move this vertex to the tail of RCV Cache. We do not directly delete it, because even a vertex with $r = 0$ could be referred again by a subsequent task. We call such strategy as *lazy model*. Only when the cache is full, we replace the zero-referred vertices by new coming ones. In a rare case, if there is no vertex with $r = 0$ in RCV Cache, the candidate retriever will

go to sleep until some tasks finish their computation and release the referred vertices.

Fault tolerance. Similar to many distributed graph processing systems [12, 18], G-Miner achieves fault tolerance by saving a snapshot periodically. For each checkpoint, the master instructs each worker to dump the state of its partition to HDFS, where the state includes the inactive tasks on disk, the *active* tasks and *ready* tasks in the task-pipeline, RCV Cache and vertex table. Thanks to the task model design, we do not need to checkpoint any message.

When a slave is dead, the fault recovery only needs to re-run the tasks of the dead worker from the previous checkpoint, while the other live workers continue their progress as tasks are independent. The progress scheduler in the master applies task stealing to dynamically re-distribute the workload of the dead worker to other workers.

8 EXPERIMENTAL EVALUATION

We evaluate the performance of G-Miner on five graph mining applications and compare the results with those of four existing graph processing systems. We also report results on scalability and optimization techniques.

8.1 Applications, Datasets, and Settings

We implemented five typical graph mining applications, on both *non-attributed graphs* (i.e., each vertex only has an ID but no label/attributes) and *attributed graphs*, to demonstrate the expressiveness of G-Miner. The experimental results obtained also help validate the overall performance of G-Miner for various categories of applications and datasets.

Triangle Counting (TC) takes a non-attributed graph as input and uses only the 1-hop neighbors of each vertex in the computation [4, 26], which is a relatively light mining workload that vertex-centric systems can handle [16, 47].

Maximum Clique Finding (MCF) is also applied on non-attributed graphs and can be computed based on the 1-hop neighborhood, but the workload is heavy [5]. We followed [5] to implement an efficient algorithm with optimized pruning strategy on G-Miner.

Graph Matching (GM) is a fundamental operation for graph mining and network analysis. Existing parallel algorithms [27, 30] usually first execute vertex-centric exploration and then perform subgraph joining on the results. GM takes an attributed graph as input and has complex workload (vs. TC and MCF), which is hard to be implemented using the API of other distributed graph systems.

Community Detection (CD) defines a set of vertices which share common attributes and together form a dense subgraph as a community [46]. CD aims to detect all such communities in an attributed graph. CD has complex and heavy workload, and it is non-trivial for any existing distributed graph system to solve this graph mining problem. We adopted [33] to mine the dense subgraph topology and guarantee the similarity of attributes in a community by a filtering condition on newly added vertex candidates.

Graph Clustering (GC) is widely used for recommendation. We followed the FocusCO algorithm [21] to group focused clusters from an attributed graph based on user preference. The focused clusters are communities that have similar attributes with the given users' samples. We implemented it on G-Miner to show that it is capable of handling such a convergent algorithm on large attributed graphs, as FocusCO iteratively performs an expensive subgraph dynamic update until convergence.

Dataset	|V|	|E|	Max.Deg	Avg.Deg	|Attr|
Skitter	1,696,415	11,095,298	35,455	13.081	-
Orkut	3,072,441	117,184,899	33,313	76.281	-
BTC	164,732,473	772,822,094	1,637,619	4.69	-
Friendster	65,608,366	1,806,067,135	5,214	55.056	-
Tencent	1,944,589	50,133,369	456,864	55.562	122896
DBLP	1,805,882	8,439,133	2,134	9.346	1640

Table 2: Graph Datasets

Datasets and Setting. We ran the experiments on a cluster of 15 nodes connected by Gigabit Ethernet, where each node has 48GB RAM, two 2.0GHz Intel(R) Xeon(R) E5-2620 CPU (each CPU has 6 cores, and 12 virtual cores by hyper-threading) and a SATA disk (6Gb/s, 10krpm, 64MB cache), running 64-bit CentOS 6.5 with Linux kernel 2.6.32. We used HDFS (version 2.6.0).

We used six graphs as shown in Table 2. Skitter[1], Orkut[2], BTC[3] and Friendster[4] are non-attributed graphs. Tencent is an attributed graph provided by the KDD contest[5], where each vertex records a person with his/her interest tags (122, 896 in total), and each edge shows the friendship between two users. DBLP[6] is a co-authorship graph, where each vertex records the list of conferences or journals the author published. The dataset involves 1, 640 conferences and journals.

The graphs are various types of real-world networks and semantic graphs that are popularly used by the KDD community for evaluating graph mining algorithms. Some of the graphs are small or medium-sized because the systems we compared with could not handle the graph mining jobs (even on TC) for larger graphs.

8.2 Overall Performance Comparison

We first compare G-Miner with the four graph processing systems we mentioned in §3.

All systems. We tested only TC and MCF on the four non-attributed graphs, as most of the systems could not handle the other three applications. Table 3 presents the results.

Arabesque, Giraph and GraphX either ran out of memory (OOM) or over 24 hours for most of the cases. For the relatively small graphs (i.e., Skitter and Orkut), their performance (even for TC) is significantly worse than G-Miner. The main reasons for their poor performance are because (1) these systems do not have a mechanism to process a large number of subgraphs (e.g., pipelining them to disk) and hence result in OOM, and (2) they all suffer from high synchronization barrier costs. Thus, they cannot scale to handle heavier workloads and larger datasets.

Compared with G-thinker, G-Miner is nearly twice as fast for the largest graph, Friendster, but does not obtain an obvious advantage for the smaller datasets. They both adopt a subgraph-centric model,

[1] http://konect.uni-koblenz.de/networks/as-skitter
[2] http://konect.uni-koblenz.de/networks/orkut-links
[3] http://km.aifb.kit.edu/projects/btc-2009/
[4] http://snap.stanford.edu/data/com-Friendster.html
[5] https://www.kaggle.com/c/kddcup2012-track1
[6] http://dblp.uni-trier.de/xml/

Dataset	Arabesque	Giraph	GraphX	G-thinker	G-Miner
TC					
Skitter	117.4	72.4	168.2	29.1	10.7
Orkut	693.4	2191.6	343.5	83.5	73.6
BTC	x	x	x	300.3	282.4
Friendster	x	x	x	2854.6	1482.3
MCF					
Skitter	-	x	15129.3	136.2	34.4
Orkut	-	x	-	189.2	97.3
BTC	-	x	-	8470.5	7503.5
Friendster	-	x	-	2955.7	1595.9

Table 3: Elapsed running time in seconds ("-": >24 hours; "x": job failed due to OOM)

Dataset (Matched Pattern)	Skitter 3,995,063,219	Orkut 101,282,186,486	BTC 7,384,754,036	Friendster 425,807,851,924
Time (s)				
G-Miner	27.2	100.6	47248.5	2417.2
G-thinker	122.1	619.1	-	5572.9
CPU Util.				
G-Miner	52.75%	84.45%	76.25%	84.83%
G-thinker	29.85%	33.70%	11.90%	14.60%
Mem. (GB)				
G-Miner	5.72	22.49	41.02	241.51
G-thinker	28.27	64.27	63.27	478.05
Net. (GB)				
G-Miner	0.12	4.93	6.47	114.73
G-thinker	0.89	16.54	29.23	979.22

Table 4: Performance of G-Miner and G-thinker

and thus their performance demonstrates that such computational models are suitable for graph mining jobs. We further analyze their differences in the following experiment.

G-Miner vs. G-thinker. We study the detailed resource utilization of G-Miner and G-thinker, thus verifying the effectiveness of G-Miner's task-pipeline design for graph mining. We ran GM, as GM is more costly to process than TC and MCF so that we can show more performance differences between G-Miner and G-thinker. We ran GM with the pattern graph P given in Figure 1. We still used the four non-attributed graphs but randomly assigned a label from $\{a, b, c, d, e, f, g\}$ to each vertex with a uniform distribution. We did not use Tencent and DBLP, as G-thinker cannot efficiently handle the high-dimensional attribute lists of these two graphs. Table 4 reports the elapsed running time, the average CPU utilization, the peak aggregate memory usage, and the aggregate amount of network communication of the cluster. We also list the total number of matched patterns under the dataset names.

For this heavier workload, G-Miner is significantly faster than G-thinker for all the graphs. This can be partially explained by G-Miner's much higher CPU utilization. Note that the CPU utilization for Skitter is considerably low because it is the smallest graph and so the workload is not heavy enough to make full use of the resource. G-Miner also reduces the communication load and memory usage by its design of process-level cache shared by all the computing threads.

We further verify the effectiveness of G-Miner's task-pipeline design by plotting the CPU, network and disk I/O utilization rate of both G-thinker and G-Miner for the largest graph Friendster. As shown in Figure 5, although G-thinker also adopts a subgraph-centric computational model, its system design does not allow it to effectively overlap network communication with CPU computation,

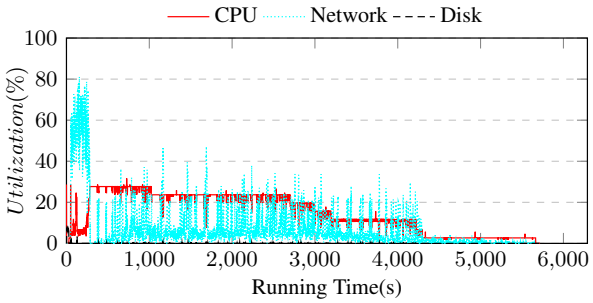

Figure 5: CPU, network and disk I/O utilization of G-thinker, running GM on Friendster (best viewed in color)

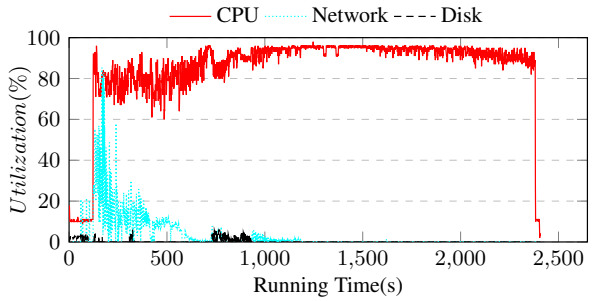

Figure 6: CPU, network and disk I/O utilization of G-Miner, running GM on Friendster (best viewed in color)

and CPU cores are waiting for data from network in short intermittent periods throughout the whole process. In contrast, Figure 6 shows that G-Miner allows all the CPU cores to be highly utilized all the time, as its streamlined task-pipeline continues to feed the computing threads with active tasks to be processed. The design of the LSH-based task priority queue and RCV Cache in G-Miner also effectively reduces the vertex pulling except at the early stage when the cache is not yet filled.

G-Miner on heavy workloads. We next assess the performance of G-Miner on CD and GC. These two applications are much more costly to process than the other applications, and require the system to deal with high-dimensional attribute lists. The purpose of this set of experiments is to present the ability of G-Miner on running convergent graph mining algorithms. None of the four systems we compared with could handle such workloads nor express the algorithms, and therefore there is no baseline for comparison. In addition to Tencent and DBLP, we also used Skitter, Orkut and Friendster by randomly assigning an attribute list[7] to each vertex. But we excluded Tencent for GC because its graph format does not fit the algorithm in [21] for GC.

Table 5 reports the elapsed running time and memory usage. Due to the much more complicated candidate filtering and processing of CD and GC, G-Miner used considerably more memory and time to complete the jobs. However, even for these heavy, complicated workloads on graphs with high-dimensional attribute lists, G-Miner still recorded good performance numbers, especially considering

[7]Each attribute list is generated by a 5-dimension (i.e., [A-E]) uniform distribution from [1-10]. Example attribute list: *{A1, B5, C10, D6, E4}*.

Dataset	Skitter	Orkut	Friendster	DBLP	Tencent
CD					
Time (s)	*34.2*	*169.5*	*22280.1*	*73.3*	*304.9*
Mem. (GB)	*9.83*	*98.19*	*367.89*	*10.88*	*38.22*
GC					
Time (s)	*105.2*	*278.4*	*32476.3*	*94.8*	~
Mem. (GB)	*11.83*	*120.08*	*436.64*	*6.61*	~

Table 5: Performance of G-Miner on CD and GC

that Arabesque, Giraph and GraphX actually obtained worse numbers even for the lightest workload, i.e., TC, as shown in Table 3.

In conclusion, this set of experiments verify both the expressiveness of G-Miner in implementing various graph mining applications and its efficiency in processing various workloads on various datasets.

8.3 Scalability

We then evaluate the scalability of G-Miner.

The COST of scalability. McSherry et al. proposed COST [19] to measure the cost of scalability using a distributed system. COST is defined as the minimum number of cores a parallel/distributed solution used to outperform an optimized single-threaded implementation. To measure the COST of G-Miner, we ran G-Miner on a single node using 1 to 24 virtual cores, running TC and GM for Skitter and Orkut, and compared with a single-threaded implementation. We did not use the larger datasets since we could not run them for the single-threaded implementation on a single machine.

Figure 7 reports the result, where the horizontal dotted lines plot the running time of a single-threaded implementation. The COST of G-Miner is 3, 3, 2, 2, respectively, for TC on Skitter, TC on Orkut, GM on Skitter, GM on Orkut. These numbers are much lower than the systems measured in [19]. This means that the overhead of parallel graph mining using G-Miner is quite low, thanks to its streamlined task-pipeline design, as there is no synchronization barrier existing in our system design. We remark that the low COST is also because graph mining workloads are computation-intensive and can thus be more benefited from parallel computation. The scalability is also better for heavier workloads and larger datasets. For example, in Figure 7, G-Miner has the best scalability for GM on Orkut, for which it achieves 1.9, 3.8, 7.3, 9.1 and 12.8 speedups at 2, 4, 8, 12 and 24 cores. The speedup is less significant when 24 cores are used because the resources have become sufficient for these two smaller datasets.

Vertical and horizontal scalability. To assess the overall scalability of G-Miner on large graphs, we ran MCF and GM on G-Miner for Friendster. For vertical scalability, we used all 15 nodes, by varying the number of cores in each node from 1 to 24 cores. For horizontal scalability, we fixed 24 cores in each node and ran G-Miner on 10, 15, 20 nodes.

Figure 8 and Figure 9 show that G-Miner achieves good speedups in most of the cases, especially for the heavier workload GM on Friendster. G-Miner's good scalability is due to two reasons: (1) our task-pipeline design enables the computing threads to totally focus on the local task processing without being interrupted by the network communication requests, which matches the computation-intensive nature of graph mining workloads; (2) graph partitioning and task stealing enable good load balancing among the nodes.

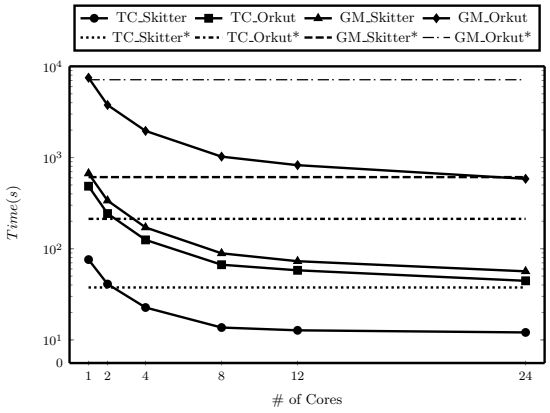

Figure 7: The COST of G-Miner

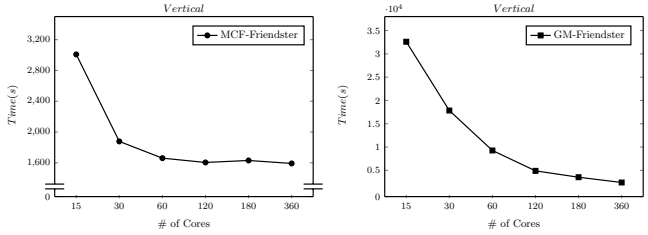

Figure 8: Vertical scalability of G-Miner

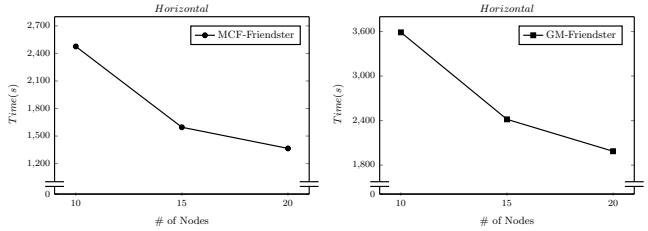

Figure 9: Horizontal scalability of G-Miner

However, some loss on the scalability still can be observed, mainly caused by the increase on the overhead of communication along with the decrease in CPU utilization when more nodes (more sufficient than needed) are involved.

In contrast, we also show the scalability of the four systems we compared with in Figure 10 as a reference. The figure shows that without a good system design and load balancing mechanism, there is no guarantee on the system scalability. Note that we were only able to measure the scalability for these systems on the smaller datasets using TC for reasons already explained in §8.2.

8.4 Evaluation on Optimization Techniques

We now evaluate the effects of some key techniques used in G-Miner. We emphasize that although the performance improvement brought by each individual technique may not particularly impressive, collaboratively (or cumulatively) they do optimize G-Miner significantly.

BDG partitioning. We first show the effectiveness of our BDG partitioning by comparing it with hash partitioning, which distributes each vertex to workers by hashing the vertex ID. Figure 11

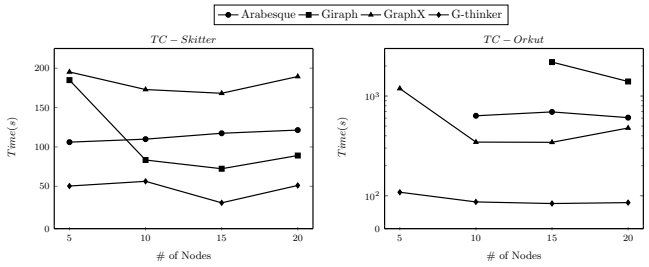

Figure 10: Scalability of other systems

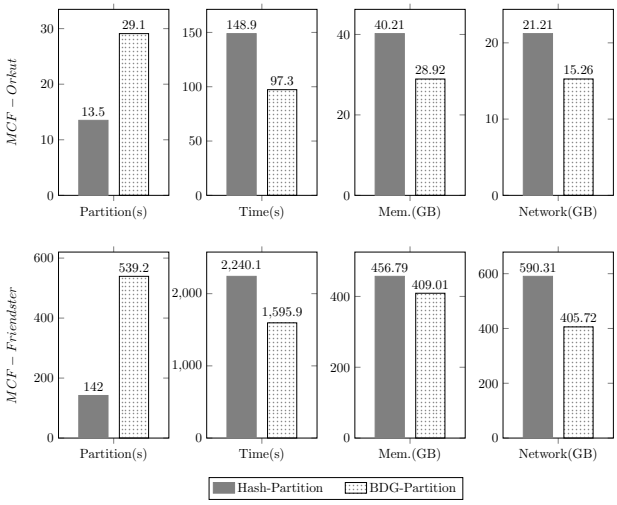

Figure 11: BDG partitioning vs. Hash partitioning

reports the performance of G-Miner for MCF, using the two partitioning methods on Orkut and Friendster. Although our BDG partitioning needs more processing time than hash partitioning, the benefit brought by it still makes the total job execution time considerably less than that of G-Miner using hash partitioning. In addition, BDG partitioning also leads to improvements on network communication and memory consumption, all of which come from the block layout of BDG partitioning that preserves the data locality. The block layout especially benefits the subgraph-centric nature of many graph mining algorithms. In contrast, hash-based approach destroys such data locality on the graph and leads to more memory references and vertex pulling through the network.

Task priority queue. Figure 12 shows the impact of our LSH-based task priority queue on the overall performance of G-Miner, where En-LSH and Dis-LSH represent G-Miner by enabling and disabling the LSH signatures on tasks. Without the LSH-based priority queue, the execution time on the four test cases can be worsen for up to 40% (from 100.7s to 143.4s for GM on Orkut, from 1596.9s to 2162.3 for MCF on Friendster). This is because without ordering the tasks, there are fewer common remote vertices to be pulled, which leads to lower cache hit rate and higher communication costs.

Task stealing. Figure 13 presents the benefit obtained by task stealing, which provides dynamic load balancing during job execution. On the smaller dataset Orkut, G-Miner has significant improvements, where the speedups are about 1.5 times. On the large dataset

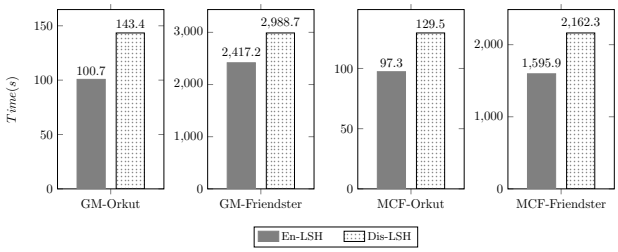

Figure 12: Impact of LSH-based task priority queue

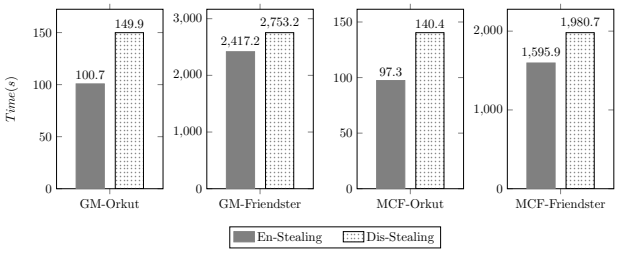

Figure 13: Impact of task stealing

Friendster, the relative speedups are not as significant since all the workers are busy until towards the later stage of the computation. However, task stealing can still directly save as much as 400 seconds at the later stage only. As it is impossible to guarantee that static load balancing will be always effective for all workloads and datasets, we believe task stealing will be a useful technique when we run into a skewed workload.

9 CONCLUSION

The complex algorithm logic and intensive computation of graph mining workloads make them hard to be expressed and inefficient to process in existing distributed graph processing systems. We presented G-Miner, which provides an expressive API and achieves outstanding performance with its novel task-pipeline that removes the synchronization barrier and hides the overheads of network and disk I/O. In addition, a set of optimization techniques are also implemented to further improve the scalability, resource utilization and load balancing of the system. As part of our future work, we plan to address limitations of G-Miner such as improving its cost model for task stealing, as we believe the effectiveness of task stealing can be significantly improved with a more sophisticated cost model. Other strategies such as recursive task splitting can also lead to better solutions for load balancing on diverse graph mining workloads. We will also address other weaknesses in the system implementation such as the overheads of spawning all tasks at the beginning and of maintaining the LSH priority queue and are considering integrating G-Miner into our general-purpose system Husky [44, 45] as we did for LoSHa [15].

Acknowledgments. We thank the reviewers and our shepherd Fabián E. Bustamante for their valuable comments and feedback, and Carter Cheng for proofreading the paper. This work was supported in part by Grants (CUHK 14206715 & 14222816) from the Hong Kong RGC.

REFERENCES

[1] James Abello, Mauricio Resende, and Sandra Sudarsky. 2002. Massive quasi-clique detection. *LATIN 2002: Theoretical Informatics* (2002), 598–612.

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

[3] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[4] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 16–24.

[5] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. 1999. The maximum clique problem. In *Handbook of combinatorial optimization*. Springer, 1–74.

[6] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.

[7] Jeremy Buhler. 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 17, 5 (2001), 419–428.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. 1.

[9] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[10] Uriel Feige, David Peleg, and Guy Kortsarz. 2001. The dense k-subgraph problem. *Algorithmica* 29, 3 (2001), 410–421.

[11] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3 (2010), 75–174.

[12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[13] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*, Vol. 14. 599–613.

[14] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.

[15] Jinfeng Li, James Cheng, Fan Yang, Yuzhen Huang, Yunjian Zhao, Xiao Yan, and Ruihao Zhao. 2017. LoSHa: A General Framework for Scalable Locality Sensitive Hashing. In *SIGIR*. 635–644.

[16] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *PVLDB* 8, 3 (2014), 281–292.

[17] 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. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.

[18] Grzegorz Malewicz, Matthew H Austern, Aart JC 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 ACM SIGMOD International Conference on Management of data*. 135–146.

[19] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*.

[20] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. 2010. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters* 31, 11 (2010), 1348–1358.

[21] Bryan Perozzi, Leman Akoglu, Patricia Iglesias Sánchez, and Emmanuel Müller. 2014. Focused clustering and outlier detection in large attributed graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1346–1355.

[22] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. NScale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal* 25, 2 (2016), 125–150.

[23] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[24] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. 22.

[25] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer science review* 1, 1 (2007), 27–64.

[26] Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *International Workshop on Experimental and Efficient Algorithms*. 606–609.

[27] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 625–636.

[28] Arlei Silva, Wagner Meira Jr, and Mohammed J Zaki. 2012. Mining attribute-structure correlated patterns in large attributed graphs. *Proceedings of the VLDB Endowment* 5, 5 (2012), 466–477.

[29] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1222–1230.

[30] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment* 5, 9 (2012), 788–799.

[31] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[32] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[33] Etsuji Tomita and Tomokazu Seki. 2003. An efficient branch-and-bound algorithm for finding a maximum clique. In *Discrete mathematics and theoretical computer science*. Springer, 278–289.

[34] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 408–421.

[35] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. 2016. Big Graph Analytics Systems. In *Proceedings of the 2016 International Conference on Management of Data*. 2241–2243.

[36] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. 2017. G-thinker: Big Graph Mining Made Easier and Faster. *CoRR* abs/1709.03110 (2017).

[37] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[38] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. 1307–1317.

[39] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1821–1832.

[40] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2018. GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit. *IEEE Trans. Parallel Distrib. Syst.* 29, 1 (2018), 99–114.

[41] Da Yan, Yuanyuan Tian, and James Cheng. 2017. *Systems for Big Graph Analytics*. Springer.

[42] Xifeng Yan and Jiawei Han. 2002. gspan: Graph-based substructure pattern mining. In *IEEE International Conference on Data Mining*. 721–724.

[43] Xifeng Yan and Jiawei Han. 2003. CloseGraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 286–295.

[44] Fan Yang, Yuzhen Huang, Yunjian Zhao, Jinfeng Li, Guanxian Jiang, and James Cheng. 2017. The Best of Both Worlds: Big Data Programming with Both Productivity and Performance. In *SIGMOD*. 1619–1622.

[45] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. In *PVLDB*, Vol. 9. 420–431.

[46] Jaewon Yang, Julian McAuley, and Jure Leskovec. 2013. Community detection in networks with node attributes. In *IEEE 13th international conference on Data Mining*. 1151–1156.

[47] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham Bangalore. 2017. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing*. 40–51.

[48] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. 2009. Graph clustering based on structural/attribute similarities. *Proceedings of the VLDB Endowment* 2, 1 (2009), 718–729.

[49] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning.. In *USENIX Annual Technical Conference*. 375–386.