Systems for Scalable Graph Analytics

Yue Zhang, School of Data Science, 223040247, *Staff, IEEE,*

***Abstract*—Graph-theoretic algorithms and graph machine learning models are essential tools for addressing many real-life problems, such as social network analysis and bioinformatics. To support large-scale graph analytics, graph-parallel systems have been actively developed for over one decade, such as Google's Pregel and Spark's GraphX, which (i) promote a think-like-a-vertex computing model and target (ii) iterative algorithms and (iii) those problems that output a value for each vertex. However, this model is too restricted for supporting the rich set of heterogeneous operations for graph analytics and machine learning that many real applications demand.**

In recent years, two new trends emerge in graph-parallel systems research: (1) a novel think-like-a-task computing model that can efficiently support the various computationally expensive problems of subgraph search; and (2) scalable systems for learning graph neural networks. These systems effectively complement the diversity needs of graph-parallel tools that can flexibly work together in a comprehensive graph processing pipeline for real applications, with the capability of capturing structural features. This tutorial will provide an effective categorization of the recent systems in these two directions based on their computing models and adopted techniques, and will review the key design ideas of these systems.

***Index Terms*—graph mining, subgraph-centric, frequent subgraph, graph, parallel, task, Fraction-Score.**

I. Introduction

Many computationally expensive problems can be solved by divide and conquer: the computation over a big dataset can be recursively divided into independent tasks over smaller subsets of the dataset, exposing great parallelism opportunities. To illustrate, we provide 2 examples described as follows.

Given a graph $G = (V, E)$ where V (resp. E) is the vertex (resp. edge) set, we consider the problem of finding those subgraphs of G that satisfy certain conditions. It may enumerate or count all these subgraphs, or simply output the largest subgraph. Examples include maximum clique finding [1], quasi-clique enumeration [2], triangle listing and counting [3], subgraph matching [4], etc. These problems have a wide range of applications including social network analysis and biological network investigation. These problems have a high time complexity (e.g., finding maximum clique is NP-hard), since the search space is the power set of V: for each subset $S \subseteq V$, we check whether the subgraph of G induced by S satisfies the conditions. Few existing algorithms can scale to big graphs such as online social networks.

Subgraph mining is usually solved by divide and conquer. A common solution is to organize the giant search space of V's power set into a set-enumeration tree [2].

This paper was produced by the IEEE Publication Technology Group. They are in Piscataway, NJ.

Manuscript received April 19, 2021; revised August 16, 2021.

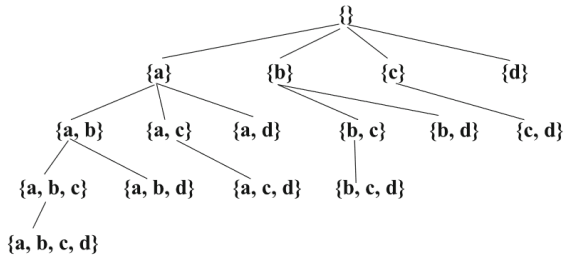

Fig. 1. Set-Enumeration Tree.

Fig. 1 shows the setenumeration tree for a graph G with four vertices $\{a, b, c, d\}$ where $a < b < c < d$ (ordered by ID). Each node in the tree represents a vertex set S, and only vertices larger than the last (and also largest) vertex in S are used to extend S. For example, in Fig. 1, node $\{a, c\}$ can be extended with d but not b as $b < c$; in fact, $\{a, b, c\}$ is obtained by extending $\{a, b\}$ with c. Edges are often used for the early pruning of a tree branch. For example, to find cliques, one only needs to extend a vertex set S with those vertices in $(V\text{-}S)$ that are common neighbors of every vertex of S, since all vertices in a clique are mutual neighbors. Also, [2] shows that to find γ-quasi-cliques ($\gamma \geq 0.5$), one only needs to extend S with those vertices that are within 2 hops from every vertex of S.

The problems we consider above share two common features: (1) pattern-to-instance: the structural or label constraints of a target subgraph (i.e., pattern) are pre-defined, and the goal is to find subgraph instances in a big graph that satisfy these constraints; (2) there exists a natural way to avoid redundant subgraph checking, such as by comparing vertex IDs in a set-enumeration tree, or partitioning by different vertex instances of the same label as in [5], [6].

Some graph-parallel systems attempt to unify the above problems with frequent subgraph pattern mining (FSM), in order to claim that their models are "more general". However, FSM is an intrinsically different problem: the patterns are not pre-defined but rather checked against the frequency of matched subgraph instances, which means that (i) the problem is in an instance-to-pattern style. Moreover, frequent subgraph patterns are usually examined using pattern-growth, and to avoid generating the same pattern from different sub-patterns, (ii) expensive graph isomorphism checking is conducted on each newly generated subgraph, as in Arabesque [7], RStream [8] and Nuri [9]. This is a bad design choice since graph isomorphism checking should be totally avoided in pattern-to-instance subgraph mining. After all, FSM is a specific problem whose parallel solutions have been well-studied, be it for a big graph [10] or for many graph transactions [11], [12], and they

0000–0000/00$00.00 © 2021 IEEE

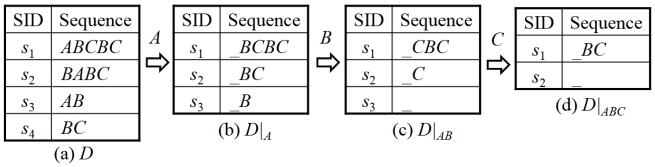

Fig. 2. Frequent Sequential Pattern Mining.

can be directly used.

We consider the pattern-growth approach: we check whether a pattern is frequent, and if so, we grow the pattern for further examination. Figure 3 illustrates the PrefixSpan algorithm for mining frequent sequential patterns, where the sequence database D in Figure 3(a) is projected by prefix pattern A (i.e., Figure 3(b)), and then AB (i.e., (c)), and finally ABC (i.e., (d)). We see that the projected database $D|_P$ is shrinking in size.

Without loss of generality, we illustrate the use of T-thinker by considering the application of mining subgraphs in the rest of this paper. Earlier work attempts to tackle this problem using MapReduce but is found to be 10 times slower than a single-threaded program [13] due to a communicatio execution pattern that underutilizes CPU cores. Other attempts face a similar problem [7], [14] which motivates the development of task-centrn-boundic graph mining systems like G-thinker [6] and G-Miner [15]. However, the latter two systems still suffer from design problems such as (1) expensive initial graph partitioning and task generation, (2) threads contend on a single data cache for one-at-a-time access, (3) a disk-based task queue that is expensive to insert new tasks.

II. Related Work

This section explains the concepts of IO-bound and CPU-bound workloads, and reviews existing graph-parallel systems. And then present the SOTA subgraph pattern enumeration algorithm gSpan [16] and the SOTA subgraph matching algorithm by [17], which are two primitives used by T-FSM [18].

IO-bound v.s. CPU-bound. The throughput of CPU computation is usually much higher than the IO throughput of disks and the network. However, existing Big Data systems dominantly target IO-bound workloads. For example, the word-count application of MapReduce [19] emits every word onto the network, and for each word that a reducer increments its counter, the word needs to be received by the reducer first. Similarly, in the PageRank application of Pregel [20], a vertex needs to first receive a value from each in-neighbor and then simply adds it to the current PageRank value. IO-bound exe- cution can be catastrophic for computation problems beyond those with a low time complexity. For example, even for tri- angle counting with time complexity $O(|E|^{1.5})$, [13] reported that the state-of-the-art MapReduce algorithm uses 1,636 machines and takes 5.33 minutes on a small graph, on which their single-threaded algorithm uses less than half a minute. In fact, McSherry et. al [21] have noticed that existing graph-parallel systems are comparable and sometimes slower than a single-threaded program. In another recent post by McSherry , he further indicated that the current dis- tributed implementations "scale" (i.e., using aggregate IO bandwidth), but their performance does not get to "a simple single-threaded implementation."

Subgraph-Centric Systems. Recently, a few systems began to explore a think-like-a-subgraph programming model, including distributed systems NScale [14], Arabesque [7] and G-Miner [15] and single-machine systems RStream [8] and Nuri [9]. Despite more convenient programming inter- faces, their execution is still IO-bound. Assume that subgraphs of diameter k around individual vertices need to be examined, then NScale (i) first con- structs those subgraphs through breadth-first search (BFS) around each vertex, implemented as k rounds of MapReduce computations to avoid keeping the numerous subgraphs in memory; (ii) **NScale** then mines these subgraphs in parallel by reducers. Since this design requires that all subgraphs be constructed before any of them can begin its computation, it leads to poor CPU utilization and the straggler's problem. **Arabesque** [7] is a distributed system where every machine loads the entire input graph into memory, and sub- graphs are constructed and processed iteratively. In the i-th iteration, Arabesque expands the set of subgraphs with i edges/vertices by one more adjacent edge/vertex, to construct subgraphs with $(i + 1)$ edges/vertices for processing. New subgraphs that pass a filtering condition are further processed and then passed to the next iteration. For example, to find cliques, the filtering condition checks whether a subgraph g is a clique; if so, g is passed to the next iteration to grow larger cliques. Obviously, Arabesque materializes subgraphs represented by all nodes in the set-enumeration tree (recall Fig. 1) in a BFS manner which is IO-bound. As an in-memory system, Arabesque attempts to compress the numerous materialized subgraphs using a data structure called ODAG, but it does not address the scalability limitation as the number of subgraphs grows exponentially. The task-based vertex-pulling API of **G-thinker** is first proposed by our G-thinker preprint [6], but our execution engine design is now significantly improved to eliminate the bad designs mentioned there. In our task-based vertex- pulling API, tasks are spawned from individual vertices, and a task can grow its associated subgraph by requesting adja- cent vertices and edges for subsequent computation. This API is then followed by $G - Miner$ [15], as indicated by the statement below Fig. 1 of [15]: "*The task model is inspired by the task concept in G-thinker*." The original G-thinker prototype in our preprint [6] is to verify that our API can significantly improve the performance of subgraph finding compared with existing systems, but the execution engine there is still a simplified IO-bound design that does not even consider multithreading; it runs multiple processes in each machine for parallelism which cannot share data.

G-Miner adds multithreading support to our old prototype to allow tasks in a machine to share vertices, but the design is still IO-bound. Specifically, the threads in a machine share a common list called RCV cache for caching vertex objects which becomes a bottleneck of task concurrency. G-Miner also requires graph partitioning as a preprocessing job, but real big graphs often do not have a small cut and are expen- sive to

partition; we thus adopt the approach of Pregel to hash vertices to machines by vertex ID to avoid this startup overhead.

All tasks in G-Miner are generated at the beginning (rather than when task pool has space as G-thinker does) and kept in a disk-resident priority queue. Each task t in the queue is indexed by a key computed via locality-sensitive hashing (LSH) on its set of requested vertices, to let nearby tasks in the queue share requested vertex objects to maximize data reuse. Unfortunately, this design does more harm than good: Because tasks are not processed in the order of their generation (but rather LSH order), an enormous number of tasks are buffered in the disk-resident task queue since some partially computed tasks are sitting at the end of the queue while new tasks are dequeued to expand their subgraphs. Thus, reinserting a partially processed task into the disk-resident task queue for later processing becomes the dominant cost for a large graph.

RStream [8] is a single-machine out-of-core system which proposes a so-called GRAS model to emulate Arab- esque's filter-process model, utilizing relational joins. Their experiments show that RStream is several times faster than Arabesque even though it uses just one machine, but the improvement is mainly because of eliminating network overheads. Recall that Arabesque materializes subgraphs represented by all nodes in a set-enumeration tree. Also, the execution of RStream is still IO-bound as it is an out-of-core system.

Nuri [9] aims to find the k most relevant subgraphs using only a single computer, by prioritized subgraph expansion. However, since the subgraph expansion is in a best-first manner (Nuri is single-threaded), the number of buffered subgraphs can be huge, and their on-disk subgraph management can be IO-bound.

DistGraph [10] partitions vertices to different workers so that the distributed memory can collectively hold a giant graph. DistGraph enumerates patterns in level-wise breadth-first search (BFS), where at level i it computes the support of candidate subgraph patterns comprising i edges. As a distributed system, it relies on efficient collective communication operations (AllToAll, AllGather and AllReduce) to minimize communication, and uses pruning techniques to avoid communication for definitely (in)frequent patterns.

Since each graph partition is expanded by 1-hop in each round, the partitions can become very large after a few rounds and overlap a lot, leading to redundant computation. Moreover, each worker not only holds its partition but also the matched subgraph instances, leading to prohibitive memory space cost.

Fractal, Arabesque, RStream and Pangolin. These systems focus on unifying several graph mining problems such as motif counting and FSM. Their programming models materialize all the matched subgraph instances of the subgraph patterns, and count these instances to determine pattern frequentness. Arabesque [7], RStream [8] and Pangolin [22] expand the matched subgraph instances in BFS manner to create and examine larger and larger subgraph instances, which is very costly since the number of subgraph instances grows exponentially. This is in contrast to GraMi's early-termination idea that determines a pattern S as frequent as soon as its current support becomes larger than τ. Pangolin [22] exposes the pattern extending phase so that programmers can more effectively prune the enumeration space by eagerly detecting duplicate embeddings. Pangolin also allows architectural optimizations (e.g., data structures) and can run not only on CPU but also on GPU like cuTS [23]. Fractal [24] mitigates the performance issue by allowing its execution engine to conduct depthfirst subgraph-instance backtracking without actually materializing the instances, but it still exhaustively mines all valid subgraph instances without any early termination (as in GraMi). Due to this algorithm inefficiency, Fractal requires users to specify a maximum pattern size n_{max}, so that patterns with more than n_{max} vertices will not be grown.

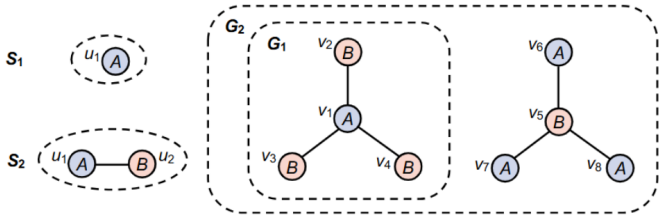

Fig. 3. Subgraph Patterns S_1, S_2 in Two Data Graphs G_1, G_2.

III. Preliminarie

This section first formally defines our single-graph FSM problem and the useful notations in Section III-A. Section 2.2 then introduces our new and more accurate support measure Fraction-Score.

A. Problem Definition

Without loss of generality, we consider an undirected graph $G = (V^G, E^G, L^G)$ with a vertex set V^G, an edge set $E^G \subseteq V^G \times V^G$, a label set L^G for vertices and edges. We only consider simple graphs without self-loops and multiple edges. Our algorithms can be easily generalized to a directed graph. Given a query graph S, **subgraph matching** finds all isomorphisms of S in data graph G, i.e., to find all mappings $\psi : V^S \rightarrow V^G$, such that (1) for each $u \in V^S$, we have $L^S(u) = L^G(\psi(u))$, and (2) for each $e = (u_i, u_j) \in E^S$, there exists $(\psi(u_i), \psi(u_j)) \in E^G$ and $L^S(e) = L^G(\psi(e))$. As an illustration, consider query graph S_2 and data graph G_1 in Figure 3, where A and B are vertex labels. Then, S_2 has 3 isomorphisms in G_1, namely $(v_1, v_2), (v_1, v_3) and (v_1, v_4)$.

Given a **support threshold** τ, FSM in G finds all subgraph patterns S with support $\geq \tau$, where support is an anti-monotonic measure such as MNI [25] or Fraction-Score (see Section 2.2). Recall that we say that a data vertex $v \in G$ is a valid match to a pattern vertex $u \in S$, denoted by $v \rightsquigarrow u$, iff there exists an isomorphism of subgraph pattern S in G that contains v, where u is mapped to v.

Also recall from Figure 4 that each pattern S is associated with a domain table, which maintains a column $D(u)$ of candidate data vertices to match to u for each $u \in S$. MNI [25] is a popular anti-monotonic support measure for single-graph FSM, which measures the least number of valid matches

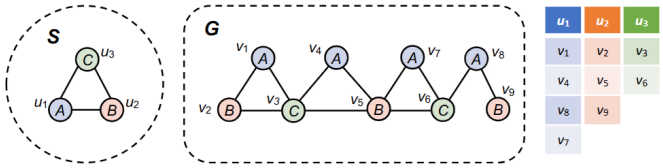

Fig. 4. Domain Illustration.

of every vertex $u \in S$, i.e., $mni(S) = \min_{u \in S} |D^*(u)|$. A pattern S is said to be frequent iff $mni(S) \geq \tau$.

IV. Proposed Methods

A. G-thinker Architecture and Challenges

Fig. 5 shows the architecture of G-thinker on a cluster of 3 machines. We assume that a graph is stored as a set of vertices, where each vertex v is stored with its adjacency list $\Gamma(v)$ that keeps v's neighbors. G-thinker loads an input graph from the Hadoop Distributed File System (HDFS). As Fig. 5 shows, each machine only loads a fraction of vertices along with their adjacency lists into its memory, kept in a local vertex table. Vertices are assigned to machines by hashing their vertex IDs, and the aggregate memory of all machines is used to keep a big graph. The local vertex tables of all machines form a distributed key-value store where any task can request $\Gamma(v)$ using v's ID.

G-thinker computes in the unit of tasks, and each task is associated with a subgraph g that it constructs and mines upon. For example, consider the problem of mining maximal γ-quasi-cliques ($\gamma \geq 0.5$) for which [2] shows that any two vertices in a γ-quasi-clique must be within 2 hops. One may spawn a task from each individual vertex v, request for its neighbors (in fact, their adjacency lists) in Iteration 1, and when receiving them, request for the 2nd-hop neighbors (in fact, their adjacency lists) in Iteration 2 to construct the 2-hop ego-network of v for mining maximal quasi-cliques using a serial algorithm like that of [2], [26]. To avoid double-counting, a vertex v only requests those vertices whose ID is larger than v (recall from Fig. 1), so that a quasi-clique whose smallest vertex is u must be found by the task spawned from u.

Such a subgraph mining algorithm is implemented by specifying 2 user-defined functions (UDFs):(1) $spawn(v)$ indicating how to spawn a task from each individual vertex in the local vertex table. (2) $compute(frontier)$ indicating how a task processes an iteration where `frontier` keeps the adjacency list of the requested vertices in the previous iteration. In a UDF, users may request the adjacency list of a vertex u to expand the subgraph of a task, or even decompose the subgraph by creating multiple new tasks to divide the mining workloads. As Fig. 3 shows, each machine also maintains a remote vertex cache to keep the requested vertices (and their adjacency lists) that are not in the local vertex table, for access by tasks via the input argument $frontier$ to UDF $compute(frontier)$. This allows multiple tasks to share requested vertices to minimize redundancy, and once a vertex in the cache is no longer requested by any task in the machine, it can be evicted to make room for other requested vertices. In UDF $compute(frontier)$, a task is supposed to save the needed vertices and edges in `frontier` into its subgraph, as the vertices in $frontier$ are released by G-thinker right after $compute(\cdot)$returns.

To maximize CPU core utilization, each mining thread keeps a task queue of its own to stay busy and to avoid contention. Since tasks are associated with subgraphs that may overlap, it is infeasible to keep all tasks in memory. G-thinker only keeps a pool of active tasks in memory at any time by controlling the pace of task spawning. If a task is waiting for its requested vertices, it is suspended so that the mining thread can continue to process the next task in its queue; the suspended task will be added back to the queue once all its requested vertices become locally available, in which case we say that the task is **ready**.

Note that a task queue can become full if a task generates many subtasks into its queue, or if many waiting tasks become ready all at once (due to other machines' responses). To keep the number of in-memory tasks bounded, if a task queue is full but a new task is to be inserted, we spill a batch of tasks at the end of the queue as a file to local disk to make room.

As the upper-left corner of Fig. 5 shows, each machine maintains a list of task files spilled from the task queues of mining threads. To minimize the task volume on disks, when a thread finds that its task queue is about to become empty, it will first refill tasks into the queue from a task file (if it exists), before choosing to spawn more tasks from vertices in the local vertex table. Note that tasks are spilled to disks and loaded back in batches to minimize the number of random IOs and lock-contention by mining threads on the task file list.

For load balancing, machines about to become idle will steal tasks from busy ones (could be spawned from their local vertex table) by prefetching a batch of tasks and adding them to the task file list on local disk. The tasks will be loaded by a mining thread for processing when its task queue needs a refill.

1) Desirabilities: Our design always guarantees that a mining thread has enough tasks in its queue to keep itself busy (unless the job has no more tasks to refill), and since each task has sufficient CPU-heavy mining workloads, the linear IO cost of fetching/moving data is seldom a bottleneck.

Other desirabilities include:

- **Bounded memory consumption**: only a pool of tasks is kept in memory at any time, the local vertex table only keeps a partition of vertices, and the remote vertex cache has a bounded capacity.
- **Efficient task spilling**: tasks spilled from task queues are written to disks (and loaded back) in batches to achieve serial disk IO, and spilled tasks are prioritized when refilling task queues of mining threads so that the number of tasks kept on disks is minimized (in fact negligible according to our experiments).
- **Vertex sharing**: threads in a machine can share vertex data in the remote vertex cache, to avoid redundant vertex requesting.
- **Independence of tasks**: tasks are totally independent (due to divide-and-conquer logic) and will never block each other.

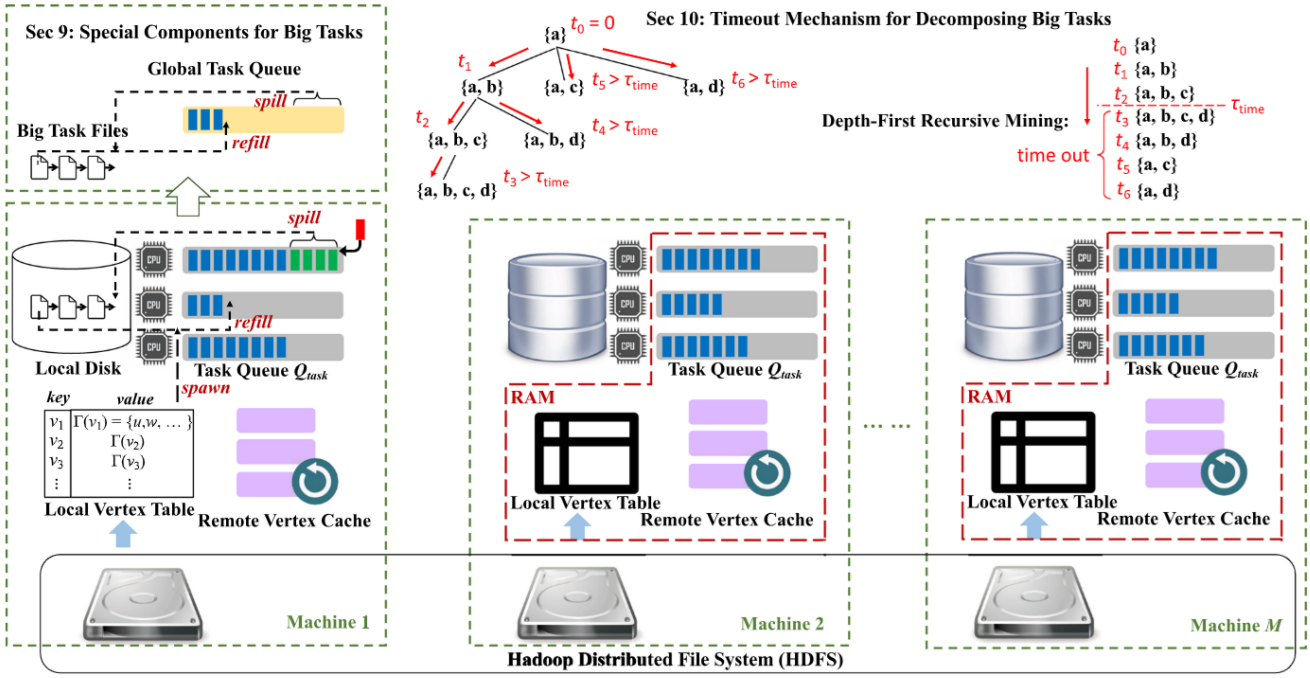

Fig. 5. G-thinker Architecture Overview.

	0. An in-memory task pool for timely computation	1. Bounded memory consumption	2. Task spilling and keeping number of tasks on disk small	3. Vertex sharing to avoid redundant communication	4. Tasks execute independently	5. Batched communication	6. Load balancing
G-thinker	✓	✓	✓	✓	✓	✓	✓
NScale	x	✓	x	x	x	✓	x
Arabesque	✓	x	x	✓	x	✓	✓
G-Miner	✓	✓	x	✓	x	✓	✓
RStream	x	✓	x	N/A	x	N/A	N/A
Nuri	✓	✓	x	N/A	x	N/A	N/A

Fig. 6. Feature Comparison of Subgraph-Centric Systems.

- **Batch vertex requests**: we also batch vertex requests and responses for transmission to combat round-trip time and to ensure throughput.
- **Task division**: if a big task is divided into many tasks, these tasks will be spilled to disks to be refilled to the task queues of multiple mining threads for parallel processing; moreover, work stealing among machines will send tasks from busy machines to idle machines for processing.

We remark that G-thinker is the only system that achieves these desirabilities and hence CPU-bound mining workloads. Fig. 6 summarizes how existing subgraph-centric systems compare with G-thinker in terms of these desirabilities.

2) Challenges: To achieve the above desirabilities, we address the following challenges. For vertex caching, we need to ask the following questions:

- How can we ensure high concurrency of accessing vertex cache by mining threads, while inserting newly requested vertices and tracking whether an existing vertex can be evicted?
- How can we guarantee that a task will not request for the adjacency list of a vertex v which has been requested by another task in the same machine (even if response $\Gamma(v)$ has not been received) to avoid redundancy?

For task management, we need to consider: (1) How to accommodate tasks that are waiting for data. (2) How can those tasks be timely put back to task queues when their data become ready? (3) How to minimize CPU occupancy due to task scheduling.

B. PrefixFPM Architecture and Implementation

PrefixFPM is written as a set of C++ header files defining some base classes and their virtual functions for users to inherit in their subclasses and to specify the application logic. We call these virtual functions as user-defined functions (UDFs). The base classes also contain C++ template arguments for users to specify the proper pattern and data structures. We now introduce these base classes as shown in Figure 7.

Trans. The $Trans$ class implements a transaction (i.e., a data instance in the input database) with a predefined transaction ID field. Users implement their transaction subclass by inheriting $Trans$ and including additional fields to store the target data instance, such as a sequence or a graph. Initially, the input dataset is read into an in-memory transaction database D, which is simply an array of objects whose type is the $Trans$ subclass.

ProjTrans. The $ProjTrans$ class implements a projected transaction $s|\alpha$ in a projected database $D|_\alpha$. A $ProjTrans$ object also has a transaction ID field indicating which transaction $s \in D$ this projected transaction corresponds to. The

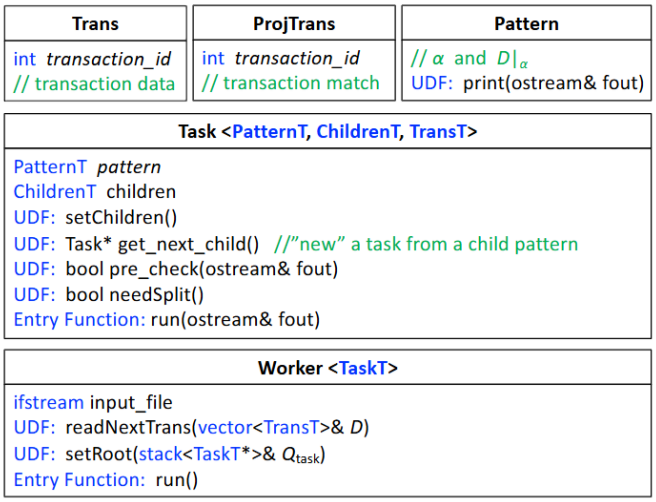

Fig. 7. PrefixFPM Programming Interface.

```
1   void run(ostream& fout){
2         if(!pre_check(fout)) return;
3         //generate new patterns
4         setChildren(children);
5         //run new child tasks
6         while(Task* t= get_next_child()){
7               if(needSplit()){
8                     q_mtx.lock();
9                     queue().push(t);
10                    q_mtx.unlock();
11              }
12              else{
13                    t->run(fout);
14                    delete t;
15              }
16        }
17  }
```

Fig. 8. The run(fout) function of base class Task.

user-defined $ProjTrans$ subclass should also indicate how $s|\alpha$ is currently matched on s, so that the matching status can be incrementally updated as the pattern α grows.

Pattern. The $Pattern$ class specifies the data structure of a pattern α, and contains a (pure) virtual function $print(fout)$ specifying how to output the object of a $Pattern$ subclass into an output file stream $fout$. Recall that PrefixFPM runs multiple task computing threads, and each thread appends the frequent patterns found by it to a file of its own (with file handle $fout$). When a job finishes, frequent patterns are simply recorded by the files written by all task computing threads.

A $Pattern$ subclass usually also includes the projected database $D|_\alpha$ as a field. In $print(fout)$, users may choose to output $D|_\alpha$ along with α, to capture the matched transactions.

Task. he $Task$ class specifies the algorithmic logic. Recall that a task t_α checks the frequentness of pattern α using $D|_\alpha$, and grows α by one more element to generate children patterns $\{\beta\}$ and their projected databases $\{D|_\beta\}$ for further mining. An object of the base class *Task⟨ PatternT, ChildrenT, TransT⟩* implements a task t_α with 3 template arguments:

- *PatternT*: the user-defined *Pattern* subclass (with $D|_\alpha$);
- *ChildrenT*: the type of the table *children* that keeps $\{D|_\beta\}$;
- *TransT*: the user-defined *Trans* subclass.

A $Task$ object t_α maintains 2 fields: a pattern α of type $PatternT$ (often containing $D|_\alpha$), and the children table $children$ that keeps $\{D|_\beta\}$, which is typically implemented as an $std :: map$ with $children[e] = D|\beta$ if β is grown from α with element e.

$TransT$ is needed since the $Task$ class provides a function to access the global static transaction database D for users to call in their $Task$ UDFs. This is useful since a projected transaction $s_i|\alpha$ usually only keeps a compact matching status towards $s_i \in D$, and to extend it with one more element e in s_i to generate $s_i|\beta$, we need to access s_i as $D[i]$, where i is the transaction ID field of $s_i|\alpha$ whose type is a $ProjTrans$ subclass.

For example, the projected transaction of PrefixSpan only keeps the position of the last match (i.e., '_' in Figure 2), to minimize the memory consumed by projected databases.

$Task$ has an internal function $run(fout)$ which executes the processing logic of the task t_α . The behavior of $run(\cdot)$ is specified by $Task$ UDFs which are called in $run(\cdot)$.

Figure 8 shows the structure of $run(\cdot)$. In Line 2, t_α first runs UDF $pre\_check(fout)$ to see if α is frequent (and its encoding is canonical if applicable), and if so, it outputs α to $fout$. If α is not pruned, Line 4 then runs UDF $setChildren(children)$ to scan $D|_\alpha$ and compute $\{D|_\beta\}$ into the table field $children$. In this step, every infrequent child pattern β should be removed from the table $children$ as a post-processing step after $\{D|_\beta\}$ are constructed. Line 6 then wraps each child pattern β in table $children$ as a task t_β, and calls the UDF $needSplit()$ to check if t_β is time-consuming (e.g., if $D|\beta$ is large). If so, we add t_β to the task queue Q_{task} (Lines 8–10) to be fetched by available task computing threads (recall that Q_{task} is a global last-in-first-out task stack protected by a mutex), which divides the computing workloads by multithreading. Otherwise, we recursively call t_β's $run(fout)$ to process the entire checking and extension of β by the current thread, which avoids contention on Q_{task}.

Worker. A PrefixFPM program is executed by subclassing the $Worker < TaskT >$ base class and calling its $run()$ function. Here, $TaskT$ refers to the user-defined $Task$ subclass, from which $Worker$ derives the other necessary types such as $TransT$.

The *run()* function:(1) Keeps calling UDF *getNextTrans()* to read transactions and append them to the transaction database D; (2) Calls the UDF $setRoot()$ to generate root tasks (where α contains only one element) into Q_{task}; (3) Creates k task computing threads to concurrently process the tasks in Q_{task}.

Implementing $Worker :: setRoot(\text{cot})$ should be similar to implementing $Task :: setChildren(\text{cot})$: instead of constructing $\{D|_\beta\}$ from $D|_\alpha$, we construct $\{D|e\}$ from D . Each seed task $t_e = \langle e, D|e\rangle$ is added to Q_{task} to initiate parallel computation.

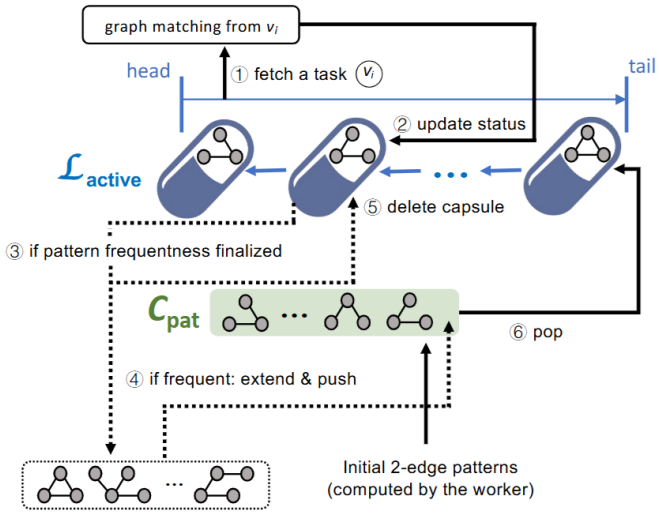

Fig. 9. System Architecture of T-FSM.

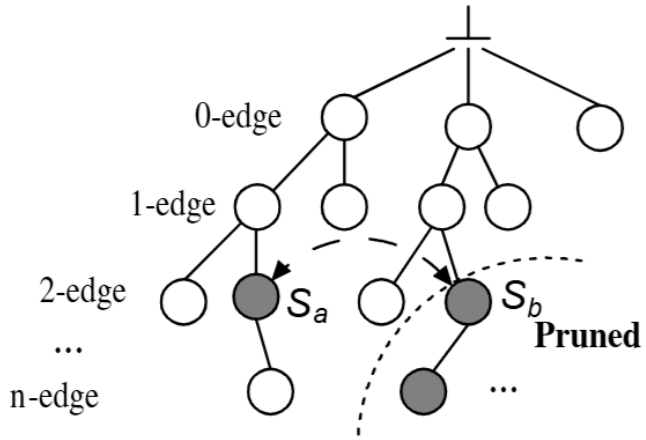

Fig. 10. Pattern-Growth Tree.

At the beginning of $Worker :: setRoot(\text{cot})$, we also need to get the element frequency statistics and eliminate infrequent elements (i.e., they are not considered when growing patterns).

During parallel task computation, each computing thread keeps fetching a task t_α from Q_{task} to call its $run(fout)$ function, and gets hung to release the CPU core when it cannot find a task in Q_{task} . Note that while Q_{task} is currently empty, another thread may be processing a task and could add more subtasks back to Q_{task} . The job terminates only if all k task computing threads are hung and no task is found in Q_{task}.

C. Overview of T-FSM Mining Process

T-FSM is implemented as a shared-memory parallel system focusing on parallel task scheduling and computation, but it can easily be extended for distributed execution using the vertex pulling technique of G-thinker [27]. In this section, we first overview the mining process of T-FSM, followed by a brief cost analysis, and then provide the technical details.

1) Mining Process Overview: **Job Initialization and Initial Pattern Candidates.** A T-FSM program begins by letting the worker perform the following steps:(1) Load the input graph G, (2) Scan G to obtain and output the set of frequent vertex labels V_{freq} and frequent 1-edge patterns E_{freq}, (3) Prune the edges of G that do not match any 1-edge pattern in E_{freq}, (3) Use E_{freq} to create an initial set of 2-edge candidate patterns, denoted by C^0_{pat}, for parallel task-based pattern frequentness evaluation and pattern extension by compers to maximize CPU utilization. To illustrate, consider the example in Figure 4 with a support threshold $\tau = 3$. Let: $V_{freq} = \{A, B\}$, $E_{freq} = \{(A, B)\}$. Label C is pruned because only 2 vertices in G have label C, which is below the threshold. Therefore, G is pruned to contain only the following 4 edges: $(v_1, v_2), (v_4, v_5), (v_5, v_7), (v_8, v_9)$. The initial candidate pattern set is: $C^0_{pat} = \{A\text{-}B\text{-}A, B\text{-}A\text{-}B\}$.

Pattern Containers Figure 6 provides an overview of the system architecture of T-FSM, which utilizes two pattern containers: (1) C_{pat}, which keeps candidate patterns to be evaluated. (2) $\mathcal{L}_{active}$, which keeps a list of active patterns currently under task-based frequentness evaluation.

Specifically, $\mathcal{C}_{pat}$ is implemented as a stack, protected by a mutex for concurrent access by compers. Initially, $\mathcal{C}_{pat}$ is populated with $\mathcal{C}^0_{pat}$, the set of 2-edge candidate patterns. When the capacity of $\mathcal{L}_{active}$ is not full, a comper pops a new candidate pattern S from $\mathcal{C}_{pat}$ for evaluation, and S is added to $\mathcal{L}_{active}$ with its status (e.g., domain table) being allocated and initialized.

If a frequent pattern is found, it will be extended with edges in E_{freq} to create more candidate patterns, which are then pushed into $\mathcal{C}_{pat}$. We use gSpan's pattern extension approach, which extends a new edge along the rightmost path of S's DFS code tree [16].

The stack structure of $\mathcal{C}_{pat}$ helps to grow existing frequent patterns into larger ones, achieving a near depth-first traversal order on the pattern-growth tree (as shown in Figure 10). This minimizes the number of candidates in $\mathcal{C}_{pat}$, in contrast to a queue-based breadth-first search (BFS) approach. Since only subgraphs are kept in $\mathcal{C}_{pat}$ without any status information, the memory consumption is low.

On the other hand, $\mathcal{L}_{active}$ is implemented as a linked list of active patterns. To keep memory bounded, we only allow $\mathcal{L}_{active}$ to contain at most n^{max}_{active} active patterns at any time, where n^{max}_{active} is a user-specified capacity parameter (default is 32). A pattern S appears earlier in $\mathcal{L}_{active}$ if it was fetched from $\mathcal{C}_{pat}$ and inserted into the tail of $\mathcal{L}_{active}$ for evaluation earlier.

Whenever a comper attempts to obtain a subgraph-matching task, it will check the task availability of active patterns starting from the head of $\mathcal{L}_{active}$. It will fetch a task from the first active pattern $S \in \mathcal{L}_{active}$ with an available task.

Memory Cost Analysis. Recall that $\mathcal{C}_{pat}$ only keeps candidate subgraph patterns without any status information. Therefore, the memory cost is primarily dominated by $\mathcal{L}_{active}$, which contains the active patterns. The memory cost of $\mathcal{L}_{active}$ is well bounded since it can hold at most n^{max}_{active} active patterns, and each active pattern is associated with a pattern capsule that contains a small, bounded number of active

subgraph-matching tasks.

Worker Mining Procedure. In T-FSM, each comper is a thread that keeps fetching the next task for processing if available, or sets its state to idle otherwise. The worker periodically checks if there are still tasks to be processed (i.e., if $\mathcal{L}_{active}$ or $\mathcal{C}_{pat}$ is not empty), or if some comper is still computing, which may generate new candidate patterns in $\mathcal{C}_{pat}$.**Case i:** If there are still tasks to be processed, the worker wakes up idle compers to process them (worker-compers notification is via condition variables). **Case ii:** If all compers are idle and both $\mathcal{L}_{active}$ and $\mathcal{C}_{pat}$ are empty, the worker terminates the T-FSM program.

Mining Procedure of a Comper: An Overview of the Steps. A comper keeps fetching tasks from $\mathcal{L}_{active}$ for evaluation as follows. It scans $\mathcal{L}_{active}$ from the head to obtain a subgraph-matching task from the capsule of the first active pattern $S \in \mathcal{L}_{active}$ with an available task (see Figure 9). **Case 1:** If a task is successfully fetched from the capsule of a pattern S, the task is executed, and the status of S is updated accordingly. **Case 1.1:** If the task determines that S is frequent, it will extend S to generate larger candidate patterns, push them into $\mathcal{C}_{pat}$, and delete S from $\mathcal{L}_{active}$. **Case 1.2:** If the task determines that S is infrequent, it directly deletes S from $\mathcal{L}_{active}$. The comper then continues to the next round, fetching another task from $\mathcal{L}_{active}$ for evaluation.

Case 2: If, during a round, a comper cannot find any task after scanning the whole $Active$, then: **Case 2.1:** If $|\mathcal{L}_{active}| < n^{max}_{active}$, the comper pops a new candidate pattern S from $\mathcal{C}_{pat}$, allocates a pattern capsule for S, and inserts it into the tail of $\mathcal{L}_{active}$. The comper then continues the next round to fetch another task from $\mathcal{L}_{active}$ for evaluation. **Case 2.2:** If the capacity of $\mathcal{L}_{active}$ is full, the comper goes idle directly, which may be awakened by the worker later either to process new tasks or to terminate the task probing loop if the worker flags the T-FSM program to terminate.

Time Cost Analysis. Recall that Step $\circ 4$ in Figure 6 extends each frequent pattern by an edge using rightmost path extension and conducts a canonicality check on each newly extended pattern, as in gSpan [16]. Therefore, no redundant patterns are inserted into $\mathcal{C}_{pat}$, and any pattern candidate in $\mathcal{C}_{pat}$ must be extended from a frequent pattern.

Let us denote the number of frequent patterns be n_{freq}, and assume that the average number of rightmost extensions of a frequent pattern's DFS code tree is n_{fout}. Thus, the total number of candidate patterns to examine is at most $n_{freq} \cdot n_{fout}$ While the value of n_{freq} and n_{fout} is difficult to analyze (e.g., gSpan's paper [16] does not provide such an analysis), they are well-bounded in practice when a selective support threshold τ is used to find only very frequent patterns.For each candidate pattern S with vertices $\{u_1, u_2, \ldots, u_k\}$, each entry in the domain table may initiate a subgraph-matching task, giving at most $\sum_{i=1}^{k} |D(u_i)|$ subgraph-matching runs in total. However, the actual number is much smaller due to various pruning techniques (see Section 3.4). Let the average cost of each subgraph-matching run be C_{match}, then the time complexity of FSM is: $O(n_{freq} \cdot n_{fout} \cdot C_{match} \cdot \sum_{i=1}^{k} |D(u_i)|)$. Although subgraph isomorphism is NP-complete [28], we use the latest algorithm for subgraph matching with pruning techniques [17], ensuring that C_{match} is well-bounded.

Dataset	|V|	|E|	Max Degree	Avg Degree
Youtube	1,134,890	2,987,624	28,754	5.27
Skitter	1,696,415	11,095,298	35,455	13.08
Orkut	3,072,441	117,184,899	33,313	76.28
BTC	164,732,473	361,411,047	1,637,619	4.39
Friendster	65,608,366	1,806,067,135	5,124	55.06

Fig. 11. GRAPH DATASETS.

V. Performance Evaluation and Experimental Results

We compared G-thinker with the state-of-the-art graphparallel systems, including (1) the most popular vertexcentric system, Giraph [29] (to verify that vertex-centric model does not scale for subgraph mining), (2) G-Miner [15] and (3) Arabesque [7]. NScale [14] is not open-sourced. We also tested the single-machine out-of-core subgraph-centric system such as RStream [8]. We used the 3 applications also used in the experiments of [6], [15] for performance study: (1) maximum clique finding (**MCF**), (2) triangle counting (**TC**), and (3) subgraph matching (**GM**). We compared with Giraph on MCF and TC as their vertex-centric algorithms exist [30], [31]. Arabesque also only provided MCF and TC implementations. All relevant code can be found at http://www.cs.uab.edu/yanda/gthinker. Figure 11 shows real graph datasets used: Youtube, Skitter, Orkut, BTC and Friendster. They have different characteristics, such as size and degree distribution. All our experiments were conducted on a cluster of 16 virtual machines (VMs) on Microsoft Azure. Each VM (model D16S V3 deployed with CentOS 7.4) has 16 CPU cores, 64 GB RAM, and is mounted with a 512 GB managed disk. Each experiment was repeated 10 times, and all reported results were averaged over the 10 runs. We observed in all our experiments that the disk space consumed by G-thinker is negligible (since compers prioritize spilled tasks when refilling their Qtask), and thus we omit disk space report.

Comparison among Distributed Systems. Figure 12 reports the (1) running time and (2) peak VM memory consumption (taking the maximum over all machines) of our 3 applications over the 5 datasets shown in Figure 11. We can see that Arabesque and Giraph incurred huge memory consumption and could not scale to large datasets like BTC and Friendster, since they keep materialized subgraphs in memory. G-Miner is memory-efficient as it keeps tasks (containing subgraphs) in a disk-resident task priority queue; it is also more efficient than Arabesque and Giraph. However, while GMiner scales to large datasets, its performance is very slow on them. This is caused by its IO-bound disk-resident task queue, where task insertions are costly when the data size and hence task number become large. G-Miner also failed to finish any application on BTC within 24 hours, which is likely because of the uneven vertex degree distribution of BTC where the dense part of BTC incurs enormous computation workloads, and G-Miner is not able to handle such scenarios efficiently.

Dataset	Arabesque	Giraph	G-Miner	G-thinker
(a) Triangle Counting (TC):				
Youtube	60.3 s / 2.8 GB	74.3 s / 1.5 GB	14.7 s / 1 GB	7.1 s / 0.4 GB
Skitter	90.8 s / 4.8 GB	73.9 s / 3.9 GB	17.1 s / 1.1 GB	9.5 s / 0.5 GB
Orkut	533 s / 17.7 GB	197 s / 15 GB	667 s / 2.3 GB	26.6 s / 1.2 GB
BTC	x	x	> 24 hr / 6.7 GB	181 s / 3 GB
Friendster	x	x	10915 s / 9.2 GB	516 s / 3.1 GB
(b) Maximum Clique Finding (MCF):				
Youtube	58.8 s / 4.7 GB	177 s / 6.2 GB	13.7 s / 1 GB	10.4 s / 0.5 GB
Skitter	145 s / 4.9 GB	x	26.2 s / 1.2 GB	85.6 s / 0.6 GB
Orkut	2007 s / 44.7 GB	x	691 s / 2.5 GB	95.9 s / 1.3 GB
BTC	x	x	> 24 hr / 7.3 GB	1831 s / 3 GB
Friendster	x	x	10644 s / 7.4 GB	252 s / 3.4 GB
(c) Subgraph Matching (GM):				
Youtube	–	–	13.2 s / 0.8 GB	5.2 s / 0.5 GB
Skitter	–	–	13.9 s / 1.1 GB	7.8 s / 0.6 GB
Orkut	–	–	66.2 s / 2.2 GB	46.7 s / 1.5 GB
BTC	–	–	> 24 hr / 6 GB	153 s / 4.4 GB
Friendster	–	–	3669 s / 9.2 GB	1762 s / 7 GB

Note: (1) x = Out of memory; (2) "–" means inapplicable.

Fig. 12. SYSTEM COMPARISON.

Dataset	$\|V\|$	$\|E\|$	d_{avg}	$\|L\|$	Category
GSE1730	998	5,096	10.21	2	Biology
Yeast*	3,112	12,519	8.05	71	Biology
Human*	4,674	86,282	36.92	44	Biology
WordNet*	76,853	120,399	3.13	5	Lexical
MiCo*	100,000	1,080,298	21.61	29	Citation
UK POI*	182,334	2,816,000	30.89	36	Spatial
DBLP	317,080	1,049,866	6.62	15	Social
Youtube	1,134,890	2,987,624	5.27	25	Social
Patent*	2,745,761	13,965,409	10.17	37	Citation
Twitter	11,316,811	85,331,846	15.08	25	Social

Fig. 13. Datasets.

As Figure 12 shows, G-thinker consistently uses less memory than G-Miner and is faster from a few times to 2 orders of magnitude (e.g., see TC and MCF on Friendster). One exception is MCF over Skitter, where we find that this is because of the different task processing order of G-Miner and G-thinker. Specifically, G-thinker processes tasks approximately in the order of how the vertices in Tlocal are ordered (as tasks are spawned from vertices in Tlocal on demand when memory permits), while G-Miner prioritizes tasks using locality sensitive hashing over $P(t)$, which is the set of vertices to pull by a task t. MCF uses the latest maximum clique found to prune search space, and G-Miner happens to process the maximum clique much earlier. However, this is irrelevant to system design and really depends on how vertices are ordered in the input file (and hence in Tlocal after graph loading).

We now evaluate T-FSM and compare it with SOTA systems ScaleMine, Fractal, DistGraph, Pangolin and Peregrine. These systems only support MNI so we use MNI by default.

Datasets and Environmental Setup. We select 10 real-world graph datasets with various sizes, densities and categories as summarized 13.

Comparison with Existing Systems. Figure 12 shows the performance of T-FSM compared with ScaleMine [26], Fractal [24], DistGraph, Pangolin [22] and Peregrine on all 10 graphs shown in Table 2 for different values of support threshold τ. For some graphs, mining large patterns are expensive, we stop extending a pattern if its size goes beyond a certain threshold. We also set the maximum running time as 104 seconds. In Figure 14, "M" means Out of Memory (64 GB), "T" means Out of Time (104 s), "A" means the program aborts (occurs only in Pangolin [22] due to an assertion error), and "X" means results returned are not exact. Note that even though a program fails and may terminate quickly for the case of "M" or "A," we still plot its hatched bar to the top like for "T" to help readers easily see which system the bar corresponds to. Also, there is no bar on MiCo for Fractal and Peregrine since they do not support edge labels.

Figure 12 shows that T-FSM generally has the best performance on all the graphs, especially the denser ones. For example, on Human, T-FSM is the only system that can mine all pattern with no more than 6 vertices, since Human is very dense with an average degree of 36.92. A similar observation can be reached for UK POI, where ScaleMine is the only other system that can finish for some tested values of τ.

On GSE1730 and DBLP, although ScaleMine sometimes has a lower running time than T-FSM, we observe that ScaleMine's results are frequently not exact and even inconsistent from two different runs. For example, on DBLP when $\tau = 1800$, T-FSM returns 1745 patterns which is the same as that returned by GraMi, but ScaleMine returns 830 patterns in one run and 832 patterns in another run. This shows that the approximate pruning techniques of ScaleMine can be far from accurate.

DistGraph can only handle two graphs DBLP and Patent. Since DistGraph extends patterns by BFS, it suffers frequently from Out-of-Memory errors on Human, MiCo, UK POI, Youtube and Twitter. The approach does work well when the number of patterns is small, where performance comparable to T-FSM is achieved on DBLP when $\tau = 1800$ and 2000. The problem of pattern extension by BFS also applies to Pangolin, which aborts on all datasets except for DBLP and Patent. Moreover, we find that the results of Pangolin are different from the exact results on DBLP, which is likely due to implementation issues.

Although Fractal supports depth-first pattern extension, it exhaustively mines all valid subgraphs without any early termination (after τ matches are found), so it is the slowest among all systems. Peregrine conducts breadth-first pattern extension but considers domain table as in T-FSM where subgraph matching is conducted in a depth-first manner. However, its subgraph matching and load balancing approaches are less efficient. As a result, Peregrine runs out of time (104 s) on most datasets except for DBLP and Patent, and Youtube only when τ = 2000. Both Fractal and Peregrine do not support edge labels, so we cannot show their results on MiCo.

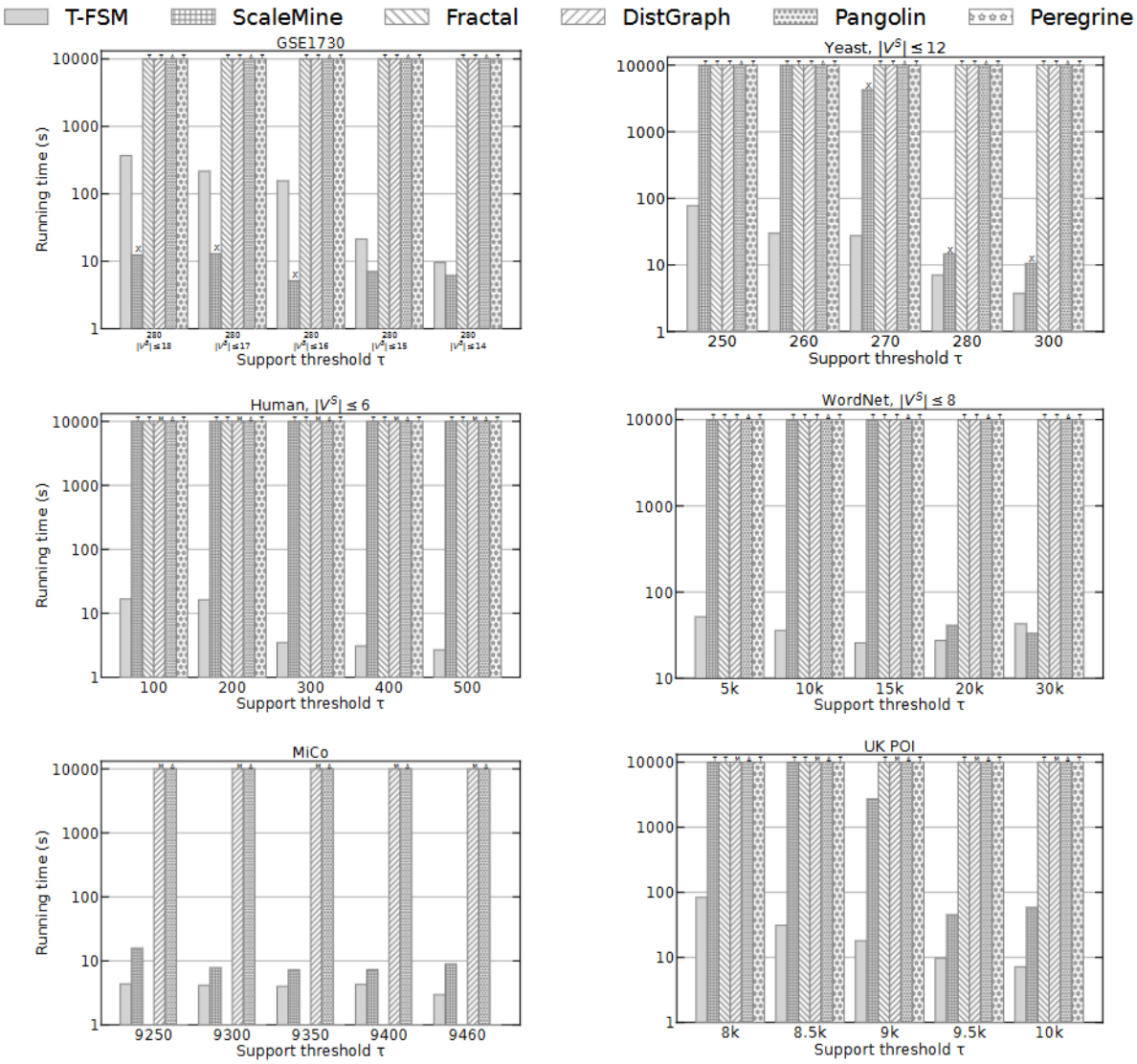

Fig. 14. T-FSM v.s. existing FSM systems with 32 Compers.

VI. Conclusions and Future Work

We presented the first truly CPU-bound graph-parallel system called G-thinker for large-scale subgraph finding, with a user-friendly subgraph-centric programming interface and a task-based execution engine. This journal extension further improved load balancing by proposing to add a global task queue to each machine for prioritized scheduling of big tasks. We also identified the performance weakness of a basic algorithm for maximum clique finding (MCF) and proposed hybrid task decomposition strategies (i.e., color-ignorant, plus color-based with timeout mechanism and candidate grouping) to scale to large graphs with a high vertex degree (e.g., over 1M on LiveJournal) and high density (e.g., $|E|/|V| = 142.42$ on MovieLens). Results show that our best MCF algorithm, MCF-H, is able to find large maximum cliques which require deep algorithmic recursion, such as 1,109 on WikiLinks in 70 minutes 51 seconds, and 944 on WebUK in 691 seconds, but for such graphs we need to use a large time- out threshold rather than the default value of 1 second to prevent task over-decomposition. Many future works on top of G-thinker are currently under way in our research group, including mining pseudo-clique structures that are even more expensive to find than cliques, such as quasi-cliques and k-plexes. Another interesting future work is to develop a general-purpose engine for subgraph matching on top of G-thinker, the subsection on "Motivations to Use G-thinker." Note that the search space of subgraph matching can be regarded as a state space tree for efficient backtracking similar to how clique-like structures can be recursively searched using a set-enumeration tree, so all our proposed task decomposition and scheduling techniques in this paper can still be applied for parallel subgraph matching.

We have used PrefixFPM to parallelize 3 state-of-the-art prefix-projection algorithms for mining 3 different kinds of frequent patterns, i.e., sequences, subgraphs and subtrees. Due to space limit, we refer interested readers to https://github.com/

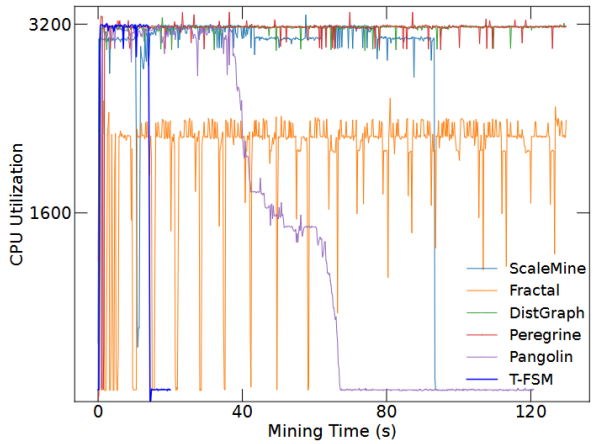

Fig. 15. CPU Utilization Rates.

yanlab19870714/PrefixFPM for their implementation. We plan to describe them in detail in a complete journal paper.

We presented an efficient system called T-FSM for parallel mining of frequent subgraph patterns in a big graph. T-FSM adopts a novel task-based execution engine design to ensure high concurrency, bounded memory consumption, effective load balancing. T-FSM integrates the latest subgraph matching algorithm to enable the efficient mining of much larger patterns. It also supports a new measure called Fraction-Score which is more accurate than the widely used MNI measure. Our experiments show that T-FSM is orders of magnitude faster than SOTA systems for FSM.

REFERENCES

[1] E. Tomita and T. Seki, "An efficient branch-and-bound algorithm for finding a maximum clique," in *International conference on discrete mathematics and theoretical computer science*. Springer, 2003, pp. 278–289.

[2] G. Liu and L. Wong, "Effective pruning techniques for mining quasi-cliques," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2008, pp. 33–49.

[3] X. Hu, Y. Tao, and C.-W. Chung, "I/o-efficient algorithms on triangle listing and counting," *ACM Transactions on Database Systems (TODS)*, vol. 39, no. 4, pp. 1–30, 2014.

[4] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 133–144, 2012.

[5] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *arXiv preprint arXiv:1205.6691*, 2012.

[6] D. Yan, H. Chen, J. Cheng, M. T. Özsu, Q. Zhang, and J. Lui, "G-thinker: big graph mining made easier and faster," *arXiv preprint arXiv:1709.03110*, 2017.

[7] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 425–440.

[8] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "{RStream}: Marrying relational algebra with streaming for efficient graph mining on a single machine," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 763–782.

[9] A. Joshi, Y. Zhang, P. Bogdanov, and J.-H. Hwang, "An efficient system for subgraph discovery," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 703–712.

[10] N. Talukder and M. J. Zaki, "A distributed approach for graph mining in massive networks," *Data Mining and Knowledge Discovery*, vol. 30, pp. 1024–1052, 2016.

[11] W. Lin, X. Xiao, and G. Ghinita, "Large-scale frequent subgraph mining in mapreduce," in *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 2014, pp. 844–855.

[12] D. Yan, W. Qu, G. Guo, and X. Wang, "Prefixfpm: A parallel framework for general-purpose frequent pattern mining," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1938–1941.

[13] S. Chu and J. Cheng, "Triangle listing in massive networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 4, pp. 1–32, 2012.

[14] A. Quamar, A. Deshpande, and J. Lin, "Nscale: neighborhood-centric large-scale graph analytics in the cloud," *The VLDB Journal*, vol. 25, pp. 125–150, 2016.

[15] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: an efficient task-oriented graph mining system," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–12.

[16] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE, 2002, pp. 721–724.

[17] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1083–1098.

[18] L. Yuan, D. Yan, W. Qu, S. Adhikari, J. Khalil, C. Long, and X. Wang, "T-fsm: A task-based system for massively parallel frequent subgraph pattern mining from a big graph," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[19] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[21] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what {COST}?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[22] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on cpu and gpu," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1190–1205, 2020.

[23] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam, "cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[24] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1357–1374.

[25] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," 2015.

[26] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil, "Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach," *arXiv preprint arXiv:2005.00081*, 2020.

[27] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W.-S. Ku, and J. C. Lui, "G-thinker: A distributed framework for mining subgraphs in a big graph," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1369–1380.

[28] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, automata, and computational complexity: The works of Stephen A. Cook*, 2023, pp. 143–152.

[29] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[30] L. Quick, P. Wilkinson, and D. Hardcastle, "Using pregel-like large scale graph processing frameworks for social network analysis," in *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2012, pp. 457–463.

[31] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big (ger) graph analytics on a dataflow engine," *arXiv preprint arXiv:1407.0455*, 2014.