ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph

Ehab Abdelhamid  Ibrahim Abdelaziz  Panos Kalnis  Zuhair Khayyat  Fuad Jamour

Computer, Electrical and Mathematical Sciences and Engineering Division
King Abdullah University of Science and Technology (KAUST), Saudi Arabia

{firstName.lastName}@kaust.edu.sa

Abstract—Frequent Subgraph Mining is an essential operation for graph analytics and knowledge extraction. Due to its high computational cost, parallel solutions are necessary. Existing approaches either suffer from load imbalance, or high communication and synchronization overheads. In this paper we propose ScaleMine; a novel parallel frequent subgraph mining system for a single large graph. ScaleMine introduces a novel two-phase approach. The first phase is approximate; it quickly identifies subgraphs that are frequent with high probability, while collecting various statistics. The second phase computes the exact solution by employing the results of the approximation to achieve good load balance; prune the search space; generate efficient execution plans; and guide intra-task parallelism. Our approaches either suffer from load imbalance, or high communication and synchronization overheads. In this paper we propose ScaleMine; a novel parallel frequent subgraph mining system for a single large graph. ScaleMine introduces a novel two-phase approach. The first phase is approximate; it quickly identifies subgraphs that are frequent with high probability, while collecting various statistics. The second phase computes the exact solution by employing the results of the approximation to achieve good load balance; prune the search space; generate efficient execution plans; and guide intra-task parallelism. Our experiments show that ScaleMine scales to 8,192 cores on a Cray XC-40 (12x more than competitors); supports graphs with one billion edges (10x larger than competitors), and is at least an order of magnitude faster than existing solutions.

I. INTRODUCTION

Frequent Subgraph Mining (FSM) finds all subgraphs with support (i.e., frequency) larger than or equal to a user-defined threshold \( \tau \). FSM is an essential step for graph analysis and knowledge extraction. For example, it is used in graph indexing [1] and clustering [2]; protein functionality prediction [3]; privacy-preservation [4]; and image processing [5]. FSM relies on subgraph isomorphism, which is NP-complete. Therefore, massive parallelism is needed to scale to large graphs.

Several research efforts [6, 7, 8] focus on the transactional FSM setting, where the input consists of many small graphs. However, modern applications (e.g., social networks, the Web, protein-protein interactions, etc.) typically involve a single large graph. The single graph case is a generalization of the transactional setting and is more difficult [9] because of the complexity of computing the support (refer to Section II for details). This paper focuses on the single graph setting. The majority of existing solutions are either serial [10, 9, 11]; or multi-threaded only within a single machine [12, 13]. Therefore, they cannot scale to large graphs.

Recent systems implement parallel FSM for a single large graph on shared-nothing architectures. MRPF [14] and MRSUB [15] are based on Map-Reduce [16], whereas Arabesque [17] and Pegi [18] are implemented on top of vertex-centric Pregel-like [19] frameworks. Table I shows the existing parallel systems and the largest graph used in their evaluations (ScaleMine is the proposed system).

Figure 1 shows strong scalability for Baseline compared to the ideal scalability (see Section V for the experimental setup). Observe that, irrespectively of the number of workers, the total response time remains almost constant. Intuitively, FSM generates a search tree of candidate subgraphs. The shape of the tree is highly irregular and the computational cost of deciding whether a subgraph is frequent varies considerably depending on the number of workers. Therefore, Baseline is almost embarrassingly parallel.

### Table I

<table>
<thead>
<tr>
<th>System</th>
<th>Max #Nodes</th>
<th>Max #Edges</th>
<th>Max #Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRPF [14]</td>
<td>300</td>
<td>2,000</td>
<td>48</td>
</tr>
<tr>
<td>MRSUB [15]</td>
<td>81,306</td>
<td>1,768,149</td>
<td>80</td>
</tr>
<tr>
<td>Arabesque [17]</td>
<td>4,589,876</td>
<td>43,968,798</td>
<td>640</td>
</tr>
<tr>
<td>Pegi [18]</td>
<td>N/A</td>
<td>100,000,000</td>
<td>50</td>
</tr>
<tr>
<td>GraMi [11]</td>
<td>11,316,811</td>
<td>85,331,846</td>
<td>(serial) 1</td>
</tr>
<tr>
<td>ScaleMine</td>
<td>56,692,400</td>
<td>1,046,834,000</td>
<td>8,192</td>
</tr>
</tbody>
</table>

1 Each worker is assigned to a separate CPU core.
among subgraphs. Due to the irregular space, there are periods when the task pool at the master is empty so that some workers may stay idle. Also, the variability of the computational cost generates stragglers. The result is a highly imbalanced system. In our example, only two workers were utilized more than 35%, whereas the utilization of the majority was below 0.2%.

A possible solution is to use intra-task parallelism: frequency computation for each candidate subgraph is divided into subtasks that run in parallel. We call this version TaskDivision. For the above-mentioned experiment, TaskDivision achieves almost perfect load balance with roughly 100% utilization for every worker. Unfortunately, TaskDivision is not embarrassingly parallel: communication and synchronization cost is substantial and straightforward pruning optimizations become extremely costly. Figure 1 shows that TaskDivision can be more than an order of magnitude slower than Baseline.

In this paper, we propose ScaleMine; a scalable parallel system for exact FSM in a single large graph. The main contribution of ScaleMine is the introduction of a two-phase approach consisting of an approximate and an exact phase. First, ScaleMine executes a novel approximate FSM algorithm that uses sampling to: (i) identify a set of subgraphs that are frequent with high probability; (ii) collect various statistics about the input graph; and (iii) build a model to predict the execution time for each subgraph frequency calculation task. The approximate phase is fast and comprises a small fraction of the total computational cost.

In the subsequent exact phase, ScaleMine implements a hybrid of Baseline and TaskDivision. The master maintains a pool of tasks, each corresponding to frequency calculation of a candidate subgraph. Available workers request tasks, calculate the frequency and return the result to the master. In contrast to existing approaches, if the task pool runs low, the master fills it with subgraphs identified in the approximate phase, meaning that workers do not stay idle. Furthermore, subgraphs identified in the approximate phase are, with high probability, frequent; therefore the algorithm prunes infrequent subgraphs early and does not waste time at wrong regions of the search space. For each task, the master uses the cost model built in the approximate phase, to simulate various scenarios of intra-task parallelism, and decides whether it pays off to split expensive tasks into subtasks, in order to improve load balance while reducing response time. Finally, statistics collected during the previous phase, are used by the workers. Frequency calculation can be mapped to a constraint satisfaction problem, where the order of constraint checking impacts the execution cost. Workers use the statistics to generate low-cost execution plans.

Note that the output of ScaleMine is the exact solution. The approximate phase is used to improve load balance, provide information that guides the search faster towards the correct solution, and decide the tasks for which intra-task parallelism is beneficial. Table I compares ScaleMine to state-of-the-art systems: ScaleMine supports 10x larger graphs (i.e., 1B edges), scales to 12x more workers (i.e., 8,192 cores on Shaheen II; a Cray XC40 supercomputer) and runs orders of magnitude faster. In summary, our contributions are:

- We develop ScaleMine, a scalable parallel frequent subgraph mining system for a single large graph.
- We propose a novel two-phase approach, consisting of an approximate phase that collects information and an exact phase that exploits the collected information to generate fast execution plans with good load balance.
- We conduct extensive experimental evaluation on a modest cluster and on Shaheen II, a high-end Cray XC40 supercomputer, using large real datasets. Our results show that ScaleMine outperforms the state-of-the-art by at least an order of magnitude in terms of the supported graph size, the number of workers, and execution time.

II. PRELIMINARIES

A. Support Metric

A graph $G = (V, E, L)$ consists of a set of nodes $V$, a set of edges $E \subseteq V \times V$ and a function $L$ that assigns labels to nodes and edges. Subgraph isomorphism finds matches of one graph in another. Given two graphs $G = (V_G, E_G, L)$ and $S = (V_S, E_S, L)$, there is a subgraph isomorphism relation from $S$ to $G$, if each node $v \in V_S$ has a matching node $u \in V_G$ with the same label, and each edge $e_s \in E_S$ has a matching edge $e_g \in E_G$ that has the same label and connectivity. Each match is called an embedding of $S$ in $G$.

Given graph $G$ and a threshold $\tau$, the FSM problem is to find all subgraphs in $G$ with support larger than or equal to $\tau$; such subgraphs are called frequent. There are many definitions of the support metric, but most applications utilize an anti-monotone support metric because it facilitates search space pruning. An intuitive metric is to count the number of embeddings of a subgraph in $G$; however, this is not anti-monotone [20]. Several anti-monotone support metrics are proposed for FSM in a single graph, such as MIS [9], HO [21] and MNI [20]. Out of these, MNI is the most efficient, since the computation of MIS and HO are NP-complete. We adopt MNI in this work.

MNI computes the support of a subgraph $S$ as the minimum number of distinct graph vertices that match each $v_i \in V_S$. An $MNI_{table}$ consists of a set of $MNI_{col}$; the $MNI$ metric returns the length of the smallest $MNI_{col}$. An $MNI_{col}(v_i)$ contains a list of distinct valid nodes, i.e., nodes corresponding to
shows the input graph $G$.

Figure 2 shows an example of Fig. $S$ embeddings in $G$ with circles, three distinct nodes. Given the six embeddings highlighted in the list of embeddings of $i$, the following optimizations that significantly improve the other embeddings. To support large graphs, GraMi employs each iteration, it solves the CSP until it finds the minimal not maintain many embeddings. GraMi models the subgraph embeddings. GraMi [11] proposed an alternative approach that does not maintain many embeddings. Each node is assigned a label. Assuming $\tau = 3$, $\nu$ is infrequent.

![Input Graph $G$](image)

![Subgraph $S$](image)

![MNI Table](image)

**B. FSM Algorithms**

The FSM search space is composed of the set of all frequent subgraphs as well as the first layer of infrequent subgraphs. The search space is not known in advance; it is built through a series of evaluation/extension iterations. Each iteration involves a large number of subgraph evaluations (i.e., frequency calculation), which are expensive operations. Most FSM algorithms store embeddings of the previously evaluated subgraphs in order to utilize them in subsequent iterations. Such approach may avoid some computations, but suffers from storing and processing an excessive number of embeddings. GraMi [11] proposed an alternative approach that does not maintain many embeddings. GraMi models the subgraph evaluation as a constraint satisfaction problem (CSP). During each iteration, it solves the CSP until it finds the minimal set of embeddings that are sufficient to satisfy $\tau$ and ignores other embeddings. To support large graphs, GraMi employs the following optimizations that significantly improve the performance: (i) prioritize light-weight node evaluations and postpone expensive ones, and (ii) utilize the graph structure as well as the previous subgraph evaluations to prune the search space. In this work, we use the approach proposed by GraMi as it is shown to efficiently handle larger graphs compared to other algorithms.

**III. Approximate Phase**

Load balance is essential for scalability. Achieving good load balance is easier when the search space is known in advance; unfortunately, this is not the case for FSM. We tackle the load balancing problem by a novel two-phase approach. The first phase builds an approximation of the search space and collects statistics. The second phase, which returns the exact results, uses the approximation to balance the load among workers and optimize their execution plans.

An effective approximation of the FSM search space should be: (i) representative: the predicted search space should represent the exact search space within acceptable accuracy; (ii) efficient: the approximation phase should have minimal overhead; (iii) informative: the approximation should be accompanied with statistics that can be used to optimize the performance of the exact phase. Although approximate FSM is not a new idea, none of the existing approximation techniques meets the aforementioned requirements. That is, they either return an insignificant fraction of the search space; do not generate the required statistics; or have a high computational cost.

ScaleMine introduces a novel approximation phase, based on sampling, that satisfies the aforementioned requirements. The approximate phase resembles the typical FSM algorithm. It begins by finding small frequent subgraphs, which are then extended to larger ones by adding edges. We employ an adaptive sampling approach (Section III-A) to estimate quickly and accurately whether a subgraph is frequent. During this process, ScaleMine collects useful statistics for each candidate subgraph (Section III-B). These statistics are later utilized to optimize the exact FSM phase.

**A. Sampling-based Subgraph Evaluation**

For a candidate subgraph $S$, a typical FSM algorithm populates the sets: $MNI_{col}(v_1), MNI_{col}(v_2), \ldots, MNI_{col}(v_d)$ with valid nodes. Iterating over all nodes in each column is expensive since it involves subgraph isomorphism. Our approach randomly samples a small fraction of these nodes, and estimates the size of each column. Given an $MNI_{col}(v_i)$, the process of validating each node resembles a binomial distribution. If the probability of success $p_i$ is known, then the number of valid nodes in $MNI_{col}(v_i) = N_i p_i$, where $N_i$ is the number of nodes in column $i$. Unfortunately, the value of $p_i$ is unknown. $p_i$ can be estimated by sampling a relatively small number of nodes. Since the problem is to decide whether a candidate subgraph is frequent or not, we can relax the problem to estimating whether $p_i$ is smaller or larger than $p_{i,\tau}$, where $p_{i,\tau}$ is called the $\tau$-probability of success and equals $\tau/N_i$. Having $p_i > p_{i,\tau}$ means that the number of valid nodes
in $MNI_{\text{col}}(v_i)$ is more than $\tau$, and consequently $MNI_{\text{col}}(v_i)$ is a valid column; otherwise, it is invalid.

ScaleMine employs the central limit theorem to estimate the probability that $p_i$ is larger than $p_{i,\tau}$. The theorem states that the distribution of the means of a large number of independent, identically distributed random variables is approximately normal, regardless of the underlying distribution [22]. For each $MNI_{\text{col}}(v_i)$ belonging to a subgraph $S$, $k$ sets of $n$ randomly selected nodes are sampled. The mean of each set is estimated as the number of valid nodes. The means of the generated $k$ sets constitute a normal distribution with mean $\hat{\mu} = np$ and standard deviation $\hat{\sigma} = \frac{s}{\sqrt{n}}$, where $\hat{p}$ is the probability of success estimated from the sampled nodes, and $s = \sqrt{np(1-p)}$. Sampling independence is guaranteed since each node belonging to an $MNI_{\text{col}}$ column has the same probability to be validated against the input graph.

After generating the distribution, a vague, inconclusive area is defined. Having a support threshold within this area means that the estimated support is not significantly different than the given threshold; therefore more sampling is needed to increase confidence. The vague area is bounded by:

$$\text{low} = \hat{\mu} - (z\hat{\sigma}) \quad \text{and} \quad \text{high} = \hat{\mu} + (z\hat{\sigma})$$

where $z$ is the value of the standard normal table for a specific confidence interval. A smaller vague area results in increasing the decision accuracy, which requires having more samples; the trade-off is an increased computational overhead.

Fig. 3 shows an example of a normal distribution generated by the sampling process, and mark three probability of success values multiplied by $n$: $p_{r_1}$, $p_{r_2}$, and $p_{r_3}$, for different support thresholds $\tau_1$, $\tau_2$, and $\tau_3$, respectively. Assume $\tau_2$ is the support threshold; the sampled nodes have larger mean than $p_{r_2}$, so the corresponding $MNI_{\text{col}}$ is predicted to be valid. As for $\tau_3$, the corresponding $MNI_{\text{col}}$ is predicted to be invalid since $p_{r_3} > \mu$. An interesting case is when $\tau_1$ is used: $p_{r_1}$ is inside the vague area and the difference between $\mu$ and $p_{r_1}$ is not significant. Thus, we cannot make a confident decision, and more sampling is required. In general, for subgraphs with support values close to $\tau$, more samples are evaluated until $p_{\tau}$ moves outside of the vague area. In some cases, $p_{\tau}$ never gets out of the vague area, so we set a maximum number of samples to stop the process regardless of the obtained accuracy.

We summarize in Algorithm 1 the proposed sampling technique. The list of domains is created for each node $v \in S$ (Line 1). For each domain, sampling is conducted, and the number of valid and invalid nodes are counted. This process iterates until the sample size is met (Line 15). Then the mean value $m$ is computed for each set of samples, and it is added to the distribution $T$. Support is estimated once the number of sampled nodes meets the default minimum number of samples (Line 18). The mean and standard deviation are then computed for the distribution $T$, which is assumed to follow a normal distribution according to the central limit theorem (Lines 19 and 20). Note that the default minimum number of samples is a user-defined value, manually tuned, to allow the central limit theorem to be applicable on the sampled data. FINISH_SAMPLING (line 21) returns true if the given support is outside of the vague area, or when the maximum sample size $(\text{maxS})$ is reached.

**Algorithm 1: Sampling-based Subgraph Evaluation**

During the approximation phase, ScaleMine collects useful statistics for each candidate subgraph. We show in Section IV how these statistics are utilized to achieve better performance during the exact phase. We show below the information collected during the approximation phase:

**Subgraph estimated support:** for each candidate subgraph, $\text{Supp}(S, G)$ is the value returned from Algorithm 1 which is an estimation of the exact value $\text{Supp}(S, G)$.

**Subgraph evaluation time:** ScaleMine estimates the time required for the exact evaluation of a candidate subgraph as:

$$\sum_{D_i \in D} \frac{\text{time}(D_i) \times |D_i|}{N_i}$$
\( D \) is the set of all domains, \( \text{time}(D_i) \) is the time spent on evaluating the sampled nodes for domain \( D_i \), \( N_i \) is the number of sampled nodes and \( |D_i| \) is the domain size. We utilize this information to guide intra-task parallelism (see Section IV-B).

**Number of valid nodes per \( MNI_{col} \):** besides having the estimated subgraph support, it is also important to know the estimated number of valid nodes per \( MNI_{col} \). This value is calculated in Line 24 of Algorithm 1. We use the estimated number of valid nodes per \( MNI_{col} \) for the early pruning in the exact phase (see Section IV-C). Note that we only store this information for approximated infrequent subgraphs because it only helps with evaluating infrequent subgraphs.

**Expected invalid columns:** an expected invalid column is a column that is predicted to have a number of valid nodes less than \( \tau \). The exact phase utilizes this information to optimize the execution plan. Note that we only store the invalid columns for the approximated infrequent subgraphs only.

### IV. Exact Phase - Parallel FSM

#### A. System Description

ScaleMine employs the master-worker paradigm on a multi-threaded shared-nothing environment. It uses the standard Message Passing Interface (MPI) for communication. Figure 4 shows the system architecture. The master receives the input graph and the user-defined support threshold \( \tau \). The graph is loaded and dispatched to nodes in the cluster by the graph loader. Each node has a single copy of the graph index, which is utilized by the workers (i.e., threads) running on that node; each core is assigned a single thread. Once the graph is loaded, ScaleMine starts its two-phase processing for finding the frequent subgraphs.

The first phase builds the approximate search space, which generates a pool of tasks, denoted by \( P_{App} \). The pool stores both frequent and infrequent predicted subgraphs (i.e., approximations). The second phase handles the exact evaluation of FSM. Alongside \( P_{App} \), ScaleMine uses a second pool of tasks, denoted by \( P_{Ex} \), to store candidate subgraphs generated from exact evaluation. This phase starts by generating a set of tasks consisting of frequent vertices. Once they are evaluated, frequent subgraphs are added to the result set, expanded, and stored in the exact task pool \( P_{Ex} \). The master prioritizes dispatching tasks from \( P_{Ex} \) to available workers until it becomes empty. Statistics of each subgraph are also sent to the corresponding worker. If such statistics are not available for a subgraph, ScaleMine generates them on the fly by approximate evaluation of the subgraph. Dispatched tasks are prioritized by size; smaller subgraphs are processed first.

Due to the nature of FSM, the number of available tasks is small at the beginning and by the end of the evaluation process. Such behavior affects the scalability and the utilization of the available resources. To avoid having idle workers, ScaleMine dispatches tasks to idle workers from the pool of approximated subgraphs \( P_{App} \) whenever \( P_{Ex} \) is empty. These tasks are not random since they are generated by the approximation phase. As such, they are expected to be evaluated in future iterations.

Instead of waiting for the exact evaluator to produce them, they are evaluated ahead of time to benefit from the available resources. There is a chance that some of these tasks should not be evaluated at all, the approximation phase is accurate enough to minimize such cases.

Once a worker finishes its task, it sends the result back to the master and asks for more tasks. Then, the master updates the task pools (\( P_{Ex} \) and \( P_{App} \)) and sends new tasks to available workers. Updating the task pool involves: (i) removing larger supergraphs from \( P_{App} \) that contains a reported frequent subgraph by the exact phase. (ii) Adding new subgraphs to \( P_{Ex} \) by extending reported frequent subgraphs, and (iii) removing tasks from \( P_{App} \) that match discovered subgraphs to avoid task duplication. ScaleMine explores the search space level by level, from smaller candidates to larger ones, until no more frequent subgraphs are found. ScaleMine incurs minimal communication overhead. Tasks along with their approximate statistical information are sent to workers, which report the computed support values back to the master. No synchronization data or embedding lists are communicated. Since the frequency computation is an expensive task compared to tasks generation and scheduling, the overhead of the master is minimal and insignificant to the overall runtime of ScaleMine.

We discuss in the following sections how ScaleMine exploits the knowledge collected during the approximate phase to provide a scalable FSM solution.

#### B. Subtasking

Reducing the number of idle workers by having enough tasks for all workers is not guaranteed to provide a balanced workload. Evaluating one subgraph can take significantly more time than another subgraph. Therefore, assigning an expensive task to a single worker would introduce a straggler worker which severely affects the load balance and hinders the system scalability. To avoid this scenario, it is important to have coherent tasks; tasks that require almost the same processing time. ScaleMine utilizes the estimated evaluation time for each predicted subgraph to distinguish between expensive and lightweight tasks. Expensive tasks are divided into smaller subtasks which are evaluated by several workers while each lightweight task is assigned to a single worker.
The goal of subtasking is to partition a task into \( n \) subtasks; each subtask is a disjoint partition of the processing space. Partitioning can be either vertical or horizontal. Vertical partitioning assigns a different \( MNI_{\text{col}} \) to different workers. Each worker will be responsible for evaluating its given column. This approach has three limitations: (i) usually the number of columns compared to the available workers is very small. Therefore, the maximum number of workers to be used is limited by the number of columns. (ii) Even if the number of columns is large enough, there is no need to evaluate all columns for candidate subgraphs that are indeed infrequent. As a result, a subset of the workers will end up doing useless work. (iii) Different columns have different execution overheads, which retracts to the first load imbalance problem.

In horizontal partitioning, graph nodes are partitioned among workers. Each worker; which has access to the whole input graph, is responsible for counting valid nodes only in its partition. A hash-based partitioning is a simple yet effective approach for distributing the workload. By opting for this partitioning strategy, enough subtasks are generated, and no extra overhead is required to process unnecessary columns.

We now describe how ScaleMine decides the number of subtasks \( (n) \) that an expensive task should be divided into. A larger number of subtasks utilizes more cores and ensures minimum amount of processing for each core which enhances the load balance. However, having more subtasks increases the processing overhead of each subtask since some pruning optimizations can not be utilized when the task is divided among multiple independent workers. Moreover, subtasking requires more synchronization overhead between the master and the involved workers. ScaleMine tries to find a near optimal value for \( n \) which ensures both good load balance and minimal computation/communication overheads. ScaleMine follows an event-driven simulation-based approach that maximizes the load balance while minimizing the number of subtasks. This approach utilizes the statistics obtained from the approximation phase to simulate the runtime of different partitioning granularities. For each granularity, ScaleMine calculates the imbalance percentage as follows:

\[
\lambda = \frac{L_{\text{max}}}{L} - 1
\]

where \( L_{\text{max}} \) is the maximum predicted workload on any worker and \( L \) is the average predicted workload over all workers. A smaller value of \( \lambda \) translates to a more coherent distribution of the subtasks. Task partitioning is based on a maximum subtask time \( \theta \), if a task has a predicted time \( T_p \) more than \( \theta \), it is partitioned into \( n = T_p/\theta \) subtasks. ScaleMine starts evaluating \( \lambda \) without any task partitioning \( (\theta=\text{maximum predicted task time}) \), and keeps generating finer workloads by decreasing \( \theta \), 10% after each iteration, until \( \lambda \) becomes less than a given threshold. Since this step is based on an event-driven simulation, its overhead is insignificant.

C. Optimizations

ScaleMine also utilizes the subgraphs statistics collected during the approximation phase to optimize the performance of the exact phase. We highlight below each one of these optimizations. Notice that these optimizations are only applied for subgraphs predicted as infrequent.

**Early Pruning:** When evaluating an \( MNI_{\text{col}} \), let \( nV \) denote the number of the already found valid nodes and let \( nR \) denote the number of remaining nodes. An important optimization is to stop evaluating this \( MNI_{\text{col}} \) when the stopping condition: \( nV + nR < \tau \) is met. In other words, the number of remaining nodes plus the valid ones is not enough to satisfy \( \tau \). It is easy to detect this case in a single-threaded solution [11] when the whole column is evaluated by a single task. However, this is quite challenging when the task is divided among multiple workers.

For subgraphs expected to be infrequent, ScaleMine employs a heuristic approach to decide when to stop evaluating a column. For a subtask \( i \), let \( nV_i, nR_i \) and \( \beta_i \) be the number of already found valid nodes, the number of nodes to be evaluated and the percentage of work assigned to worker \( i \), respectively. The stopping condition can be re-written for each subtask \( i \) as: \( nV_i + nR_i < \tau \beta_i \). Interestingly, the predicted number of valid nodes, \( pV_i \), can be utilized to allow earlier break by modifying the stopping condition to: \( nR_i < (\tau - pV_i) \beta_i \).

That is, assuming a consistent distribution of valid nodes among subtasks, a break happens earlier when the difference between \( \tau \) and \( pV_i \) becomes larger. In reality, subtasks tend to differ in the distribution of valid nodes. To accommodate this, the following is added to the left-hand side of the stopping condition: \( nV_i - (pV_i \cdot \beta_i \cdot \alpha_i) \), where \( \alpha_i \) is the percentage of the so far progress of subtask \( i \). Once this condition is met, the remaining unprocessed nodes are treated as being valid nodes, giving an upper bound on the number of valid nodes. If this upper bound does not exceed \( \tau \), then for sure the exact value cannot exceed \( \tau \) too, and the column is reported as invalid and the whole subgraph as infrequent. For the case where the upper bound exceeds \( \tau \), the evaluation needs to be repeated after turning this optimization off. To minimize the chances of such case, we introduce \( m \); a constant value between 0 and 1 (inclusive). Having a lower \( m \) delays the application of this condition. The final version of the stopping condition is:

\[
nV_i - (pV_i \cdot \beta_i \cdot \alpha_i) + nR_i < (\tau - pV_i) \cdot \beta_i \cdot m
\]

Pruning expensive nodes: Some graph nodes, especially in large dense graphs, are excessively expensive to evaluate. Similar to GraMi [11], ScaleMine avoids evaluating these nodes in favor of other lightweight nodes. GraMi relies on a user-given threshold to identify expensive nodes. Instead, ScaleMine exploits the knowledge gained during the approximation phase to identify these nodes. During the approximate phase, for each \( MNI_{\text{col}} \) in each subgraph, ScaleMine maintains the average computation steps needed to evaluate the graph nodes. Then, during the exact phase, ScaleMine identifies expensive nodes as the nodes that need significantly more computation steps than their corresponding averages. Similar to the previous optimization, the costly nodes are assumed to be valid for support computation. Then, if the calculated
support is found to be larger than \( \tau \), full re-evaluation is required but with this optimization turned off.

**Starting with invalid columns**: For a subgraph that contains a predicted invalid column, ScaleMine starts the evaluation by inspecting the predicted invalid column first, hoping that this column is indeed invalid and the whole subgraph being infrequent. As such, this optimization results in a lower cost execution plan by avoiding the overhead of evaluating other valid columns whenever an invalid column exists.

### D. Correctness

ScaleMine is a two-phase solution; approximate assessment followed by exact evaluation. ScaleMine starts the approximation phase to generate enough tasks and statistics that guide/help the exact evaluation phase. Notice that the exact phase creates its own search space by starting from the frequent edges and extending subgraphs until no more frequent subgraphs are found. The final results of ScaleMine are produced by the exact phase. For a frequent subgraph which is not discovered by the approximate phase, ScaleMine will find it as it builds its own search space and exactly evaluates it. Sometimes, ScaleMine borrows some tasks from the pool of the approximation phase and gives it to idle workers. For a borrowed infrequent subgraph that was mistakenly considered as frequent by the approximation phase, ScaleMine exactly evaluates it. Consequently, ScaleMine is guaranteed to produce the exact results correctly in all cases.

### V. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of ScaleMine using large graphs. Specifically, we show the following: (i) ScaleMine significantly outperforms state-of-the-art FSM systems (Section V-B). (ii) The effect of the proposed optimizations is significant (Section V-C). (iii) The search space generated by the approximation phase achieves both high quality and minimal runtime overhead (Section V-D). Finally, (iv) ScaleMine scales to thousands of computing cores on graphs with up to one billion edges (Section V-E). In all experiments, we only report the total processing time for ScaleMine and its competitors which do not include the time needed for loading the graph into memory. As ScaleMine uses a randomization-based component (approximation phase), we run each experiment multiple times and report their average.

#### A. Experimental setup

**Datasets**: We evaluated ScaleMine using four real graphs. Table II shows the statistics of each graph. Mico models the Microsoft co-authorship graph; nodes represent authors and labeled with their fields of interest. An edge represents a collaboration between two authors. To populate Mico, we crawled the computer science collaboration graph from academic.microsoft.com. Weibo [23] represents a microblogging network crawled from sina.weibo.com, which allows users to follow each other. Each node represents a user and labeled with the city he lives in, while each edge represents a follower-followee relationship. Twitter\(^2\) models the Twitter social network. Each node represents a user while edges represent a follower-followee relationship. The original graph does not have node labels, so we randomly assign node labels from a pool of 25 distinct labels using Gaussian distribution. Patents [24] models the U.S. patents’ citations network and includes all citations of patents granted between 1975 and 1999. Each node represents a patent and is labeled with the patent class whereas each edge represents a citation relationship. In our experiments, we set the minimum number of samples to 90. We also employ the Shapiro-Wilk normality test [25] to verify that, given our minimum number of samples, the resulting distribution of the input samples are normal.

**Implementation**: ScaleMine is implemented in C++ and uses Message Passing Protocol (MPI) for communication between the master and workers.

**Hardware Setup**: We used two hardware settings for our experiments. The first one is a local cluster of 16 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs, each with 12 cores. The machines run a 64-bit 3.2.0-38 Linux Kernel and are interconnected by a 10Gbps Ethernet switch. We use this cluster for the comparison with existing FSM solutions. We also use Shaheen II, a Cray XC40 supercomputer which has 6,174 dual sockets compute nodes based on 16 cores Intel processors running at 2.3GHz with 128GB of RAM. In this supercomputer, a hard limit of one day is set as a maximum processing time for any task, jobs that exceed this limit are killed.

#### B. Comparison with Existing Systems

In this experiment, we compare ScaleMine to GraMi [11] and Arabesque [17], the state-of-the-art FSM systems. GraMi is a single-threaded system, and subsequently, it under-utilizes the current multi-core architectures available in nowadays hardware. Arabesque is a generic distributed graph mining system that runs on a cluster of machines. This experiment is conducted on our local cluster of 16 machines.

Figure 5 shows how ScaleMine compares to the other systems using two datasets; Patents and Twitter. For medium sized graphs like Patents and Twitter, finding the frequent subgraphs is challenging which render the single machine approach; GraMi, unsuitable. In figure 5(a), both ScaleMine and Arabesque significantly outperform GraMi as they make better utilization of available resources. Moreover, ScaleMine outperforms Arabesque because it avoids storing and communicating intermediate results. Figure 5(b) shows the results on the Twitter dataset. ScaleMine achieves up to two orders of magnitude speedup compared to GraMi.

---

\(^2\)http://socialcomputing.asu.edu/datasets/Twitter
of magnitude better performance than GraMi. For $\tau=160k$, GraMi could not find the frequent subgraphs within two days. Also, Arabesque crashes for all support thresholds due to the storage and communication overheads associated with the excessive number of embeddings.

C. Optimizations

In this experiment, we measure the effect of each of the proposed optimizations using 512 computing cores on Shaheen II. We start from the baseline approach (Baseline), this approach applies exact evaluation, available tasks are dispatched to idle workers whenever possible. Then, we gradually apply each optimization. The first optimization is to build an approximate search space then utilize its pool of tasks to keep workers busy (Approx. Space). The second optimization is to divide expensive tasks, first by using the average predicted time (Task Division (Avg)), second by using our simulation-based task partitioning (Task Division (Sim)). Finally, we apply the early pruning optimization (Early pruning).

Figure 6 shows the results on Twitter and Weibo datasets. Y-Axis shows the elapsed time in hours. Each bar represents the time spent after adding the corresponding optimization (the exact runtime is on top of each bar). A column marked with red "XX" indicates that the experiment did not finish within 24 hours. For Twitter on $\tau=155k$, the baseline does not finish within one day due to the workload imbalance. We noticed in this experiment that a few cores were highly overloaded while the others were idle. When we introduce the search space approximation phase, the system finishes in 11 hours because it generates a pool of tasks to improve the utilization of workers. In other words, the system was able to complete the whole mining process significantly faster than the baseline by just utilizing the information collected during the approximation phase. With Task Division optimization, the processing time for each candidate is estimated; hence expensive tasks are identified and divided among workers. With our workload simulation technique, we achieve better performance by saving 700 seconds compared to using the average as a threshold. Dividing the tasks based on the average value did not correctly represent the distribution of the tasks execution times. Therefore, it resulted in dividing the wrong subtasks and increasing their overheads. On the other hand, ScaleMine’s simulation approach was able to correctly capture the variability among the subtasks; allowing ScaleMine to only divide the expensive ones to achieve a better balanced runtime. Finally, the early pruning strategy of ScaleMine improves system efficiency by finishing in 300 seconds less time.

A similar behavior is noticed in Weibo dataset, when using $\tau=490k$ and limiting subgraph size to 5 edges, where dividing the tasks using the average time didn’t allow the system to finish within one day. On the other hand, utilizing the proposed simulation-based task division makes it practically feasible. Overall, the conducted experiments show the importance of all the optimizations proposed by ScaleMine.

D. Approximation Phase Performance

In this experiment, we measure the accuracy of the approximation phase by comparing the reported frequent patterns from the approximation phase to the exact frequent patterns reported at the end of the mining process. If a pattern is reported as frequent by the approximation phase and it is indeed frequent, we count this pattern as a true positive. If the approximation phase missed an actual frequent pattern, then we count it as a false negative. Finally, patterns found as frequent by the approximation phase but are not actually frequent are counted as false positives. We utilize the known F-measure metric for assessing the quality of our approximation phase.

Figure 7 shows the calculated F-measure for Patents and Twitter datasets. The X-axis represents different support values. For each dataset, the used support thresholds are the same thresholds used in Figure 5. For Twitter, ScaleMine maintains an F-measure = 1 for all the different support values. As for the patents dataset, ScaleMine achieves an F-measure of more than 0.97. Achieving this high accuracy comes at a low computation cost compared to the total mining time. Figure 8 shows the time of the approximation phase alongside the time of the exact phase. In this experiment, we show the total time for each dataset using its lowest support threshold; i.e. we pick the most time-consuming mining process for each dataset. As shown in Figure 8, the approximation phase takes between 3% and 21% of the total execution time. Specifically, for larger graphs, with more expensive tasks, the proportion of approximation time becomes lower. This shows that our approximation phase can be used to support fast and accurate results as a standalone approximate FSM solution.
Different approaches have been proposed to solve this problem. The join-based approach [26] constructs new candidate subgraphs by joining smaller frequent ones. This approach suffers from the costly joins and the overhead of pruning false positives. The extension-based approach [10] extends subgraphs directly from one subgraph instead of joining two other subgraphs. Other methods focus on particular subsets of frequent subgraphs, such as maximal [27], closed [28] and significant [29, 30] frequent subgraphs. All these systems do not work when the input is a single large graph.

**Single graph mining.** In this setting, a single large graph is the input for the FSM algorithm. It is not straightforward to define an appropriate support metric in this setting, hence, several works proposed defining an appropriate anti-monotone support metric such as MIS [9] and MNI [20]. StiGRAM [9] uses the MIS metric to discover frequent subgraphs in a single sparse graph. StiGRAM needs to store intermediate results in order to evaluate frequencies. Since the number of intermediate embeddings is huge, StiGRAM becomes very expensive in practice and cannot scale to large dense graphs. Grami [11] overcomes such limitations by finding only a minimal number of embeddings to satisfy the support threshold. Both approaches are limited by the capabilities of a single machine and cannot support large-scale graphs.

**Approximate mining.** A set of approximate FSM systems were proposed to have an efficient execution of the mining task. GreW [31] proposes a heuristic approach that prunes large parts of the search space. Unlike the approximate phase of ScaleMine where the produced result includes most of the patterns, GreW discovers only a small subset of the answers. SEUS [32] is an approximate method that constructs a compact summary of the input graph. This facilitates pruning many infrequent candidates. However, it is only useful when the input graph contains few and very frequent subgraphs.

GAPPROX [33] employs an approximate version of the MIS metric. It mainly relies on enumerating all intermediate isomorphisms but allows approximate matches. SUBDUE [34] is a branch-and-bound technique that mines approximate subgraph patterns that can be used to compress the original graph. Finally, Khan et al. [35] propose proximity patterns, which relax the connectivity constraint of subgraphs. Allowing approximate matches, like in [33, 34, 35], may result in producing patterns that do not exist in the input graph. Relying on these approaches for our approximate phase may generate a lot of useless tasks.

**Parallel FSM** Several research efforts [12, 13] proposed FSM systems that utilize multi-core architectures. In [13], the authors propose a parallel extension to SUBDUE, which outputs approximate results. The work proposed in [36] parallelize StiGRAM. This system inherits all of the limitations associated with StiGRAM, i.e., it only supports small sparse graphs. Moreover, the discussed experiments show that the system cannot benefit from more than 30 cores. Although these systems can utilize the capabilities of the underlying hardware, they are still limited by the capabilities of a single machine.

### VI. RELATED WORK

#### A. Single Machine Approaches

**Transactional mining.** This setting is concerned with mining frequent subgraphs on a dataset of many, usually small graphs. We do not show the performance of ScaleMine using a lower support thresholds are the same thresholds used in Figure 5.

#### E. Scalability

We show in Figure 9 the scalability and speedup efficiency of ScaleMine using the four real graphs; Patents, Twitter, Weibo and Mico with $\tau = 15k, 155k, 460k$ and $8m$, respectively. For both Weibo and Mico, we set the maximum allowed frequent subgraph size to 5 edges. Figures 9(a) and 9(b) show the scalability and speedup efficiency for Patents and Twitter datasets when the number of workers (cores) ranges from 32 to 1024. ScaleMine achieves good speedup efficiency up to 512 cores for both datasets; 87% for Patents and 69% for the Twitter dataset. When the number of workers increases to 1024, the total time spent by ScaleMine decreased but it does not achieve good speedup efficiency; around 67% for Patents and 59% for Twitter. Such decrease in speedup efficiency is expected since subtasks become smaller and the parallelization overhead becomes relatively expensive. Figures 9(c) and 9(d) show the scalability and speedup efficiency for the largest two datasets; Weibo and Mico, starting from 512 to 8192 workers. We do not show the performance of ScaleMine using a lower number of cores as it takes significant time to finish. Similar to the last experiment, ScaleMine achieves good scalability and speedup efficiency for both datasets up to 4096 cores. After that, adding more workers does not significantly improve the performance which resulted in lower speedup efficiency.

![Fig. 7. Approximation phase accuracy for Patents and Twitter datasets. The used support thresholds are the same thresholds used in Figure 5](image)

![Fig. 8. Approximation phase time w.r.t the exact time](image)
and cannot scale to handle nowadays large-scale graphs.

B. Distributed FSM Systems

A number of distributed FSM systems [6, 7, 37, 8, 38] have been devoted for the transactional setting. These systems cannot handle the single graph setting. Other efforts [14, 18, 17, 15] proposed distributed systems for the single graph setting. MRPF [14] and MRSUB [15] rely on the MapReduce framework [16]. They both require a user-given parameter for the maximum size of the frequent subgraphs, such requirement limits their applicability. Furthermore, since these systems are based on MapReduce, they inherit its limitations. Since FSM is an iterative process, these systems incur a significant disk access cost due to flushing/reading the intermediate data between the subsequent MapReduce jobs.

Both Abraeccue [17] and Pregi [18] leverage the vertex-centric programming model to provide a solution for FSM on large graphs. Abraeccue supports FSM through a set of internal abstractions; these abstractions facilitate retrieval, processing, and extension of subgraph matches. Pregi, on the other hand, utilizes a combination of coarse-grain and fine-grain processing, where the master node controls the flow of the mining process and individual vertices are in charge of embedding discovery. Pregi utilizes aggregators to synchronize the flow of information between the workers and the master node. The main drawback of both systems is that they require to find all subgraph embeddings for computing the support as well as for subgraph extension. The number of embeddings is known to increase exponentially with the graph size, which significantly increases the communication volume between machines and hinders the scalability of the system. Moreover, since both systems rely on a fine-grain programming model, only vertices related to frequent subgraphs will be computationally active while others will be inactive. As a result, these systems may suffer from load imbalance since they cannot equally distribute active vertices across the workers.

C. Workload Prediction

APlug [39] is a framework that allows users to maximize the CPU utilization for distributed bag-of-tasks applications. It takes as input the set of user constraints; such as budget, time or minimum speedup efficiency and suggests the best combinations of CPU cores as well as the level of task decomposition that meets the user requirements. APlug is suitable for applications where the search space follows a tree structure. This tree decomposition may result in distributed subtrees that overlap with each other. Such overlap results in redundant subgraph evaluation which can be prohibitively expensive for costly tasks like FSM. PREDIcT [40] is a methodology for predicting the runtime of iterative algorithms. It estimates the number of iterations required for an algorithm to converge and the runtime of each iteration. Then, it uses an extrapolator and a cost model to predict the runtime of the algorithm. PREDIcT is meant for iterative algorithms which are known to have a uniform work per iteration. Unlike these algorithms, the runtime of each FSM iteration significantly differs from the other iterations due to the unpredictable search space and the complexity of subgraph isomorphism problem.

VII. Conclusion

In this paper, we proposed ScaleMine, an exact scalable frequent subgraph mining system for a single large graph. ScaleMine is a dual-phase system that starts with generating a sample-based approximation of the search space. Then, it runs an exact evaluation phase that utilizes the approximation phase to achieve better load balance and efficient evaluation of the candidate subgraphs. Our results show that ScaleMine is at least an order of magnitude faster than state-of-the-art systems. Furthermore, ScaleMine is capable of scaling to thousands of cores (i.e., 12x more than the competitors) and graphs with up to one billion edges (i.e., 10x more than the competitors).

The current prototype of ScaleMine assumes that each machine has an in-memory copy of the input graph. Our prototype was built to demonstrate the benefits of the two-phase approach. However, the current implementation limits the ability of ScaleMine to handle graphs larger than the capacity of a single machine. In the future, we intend to investigate graph partitioning techniques to assign a smaller partition of the input graph to each machine. Typically, graph partitioning increases the communication overhead of the system due to the exchange of data and intermediate results among machines. We will investigate adaptive partitioning techniques [41] to possibility reduce this overhead. By doing so, we expect to scale to at least an order of magnitude larger graphs (i.e., 10B edges) and more workers (i.e., 100,000 cores).

Acknowledgements

For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.
REFERENCES


ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph

Ehab Abdelhamid  Ibrahim Abdelaziz  Panos Kalnis  Zuhair Khayyat  Fuad Jamour
Extreme Computing Research Center
Computer, Electrical and Mathematical Sciences and Engineering Division
King Abdullah University of Science and Technology (KAUST), Saudi Arabia
{firstName.lastName}@kaust.edu.sa

APPENDIX

A. Abstract

This artifact contains the source codes of ScaleMine as well as the scripts required for compiling and running it. ScaleMine is a parallel frequent subgraph mining system that is capable of running on a single machine, a cluster of machines or a supercomputer. It assumes that the input graph is formatted using the widely used Labeled Graph (LG) format. ScaleMine implements all the algorithms described in the paper: ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph, SC16. To validate the results, compile the source code, run the test scripts and check the results in the corresponding output files.

B. Description

1) Check-list (artifact meta information):
   - Algorithm: Frequent Subgraph Mining, Subgraph Isomorphism
   - Program: MPI and C++ code
   - Compilation: mpic++ with the -O3 flag
   - Binary: MPI executable
   - Data set: Graph files in .lg format. Patent-citation graph is included in the public package as an example.
   - Run-time environment: Linux environment with MPI and Boost C++ libraries
   - Hardware: A single machine with multiple processors, a cluster of compute nodes or a supercomputer
   - Output: The list of frequent subgraphs, data loading time and the total execution time.
   - Experiment workflow: Download the package, compile the source code, run the application and check the output file
   - Publicly available?: Yes

2) How delivered: ScaleMine is an open source library under GNU Lesser General Public License (LGPL) (http://www.gnu.org/licenses/lgpl.html) and is hosted with source code, build and running instructions at: https://github.com/ehab-abdelhamid/ScaleMine.

3) Software dependencies: MPI and Boost libraries. ScaleMine has been tested on a 64-bit 3.2.0-38 Linux Kernel and is expected to run correctly under other Linux distributions.

4) Datasets: A sample dataset (patent-citation) is included with the ScaleMine package. All the other used datasets are publicly available. ScaleMine assumes that the input graphs are in lg format.

C. Installation

MPI as well as Boost libraries need to be installed on the target machine. Then, download ScaleMine from its github repository and run the script compile.sh to build it.

D. Experiment workflow

After downloading the package from the GitHub URL, install it using the above instruction. For evaluation, run the tool using the following command:

```
mpirun -n <N> pfsm -file <GRAPH> -freq <F> -threads <T>
```

Where:
- N: the number of MPI computation nodes (machines). Please make sure that N is at lease two; one for the master and one for a worker. Best practice is to have one computation node at each separate machine, then for each machine set a number of parallel threads.
- GRAPH: the input graph file
- F: the user-specified support threshold
- T: the number of threads per compute node

Output: ScaleMine outputs a list of frequent subgraphs followed by the elapsed time of the mining process as well as the graph loading time.

E. Evaluation and expected result

The expected results are the list of frequent subgraphs. Moreover, the algorithm execution time should follow that listed in the paper.

F. Notes

To know more about our library, send feedback, or file issues, please visit our GitHub page: https://github.com/ehab-abdelhamid/ScaleMine