Data Mining and Knowledge Discovery manuscript No.
(will be inserted by the editor)

A Distributed Approach for Graph Mining in Massive Networks

Nilothpal Talukder · Mohammed J. Zaki

DOI: 10.1007/s10618-016-0466-x

Abstract We propose a novel distributed algorithm for mining frequent subgraphs from a single, very large, labeled network. Our approach is the first distributed method to mine a massive input graph that is too large to fit in the memory of any individual compute node. The input graph thus has to be partitioned among the nodes, which can lead to potential false negatives. Furthermore, for scalable performance it is crucial to minimize the communication among the compute nodes. Our algorithm, DistGraph, ensures that there are no false negatives, and uses a set of optimizations and efficient collective communication operations to minimize information exchange. To our knowledge DistGraph is the first approach demonstrated to scale to graphs with over a billion vertices and edges. Scalability results on up to 2048 IBM Blue Gene/Q compute nodes, with 16 cores each, show very good speedup.

Keywords Parallel Graph Mining, Distributed Graph Mining, Single Large Graph, Frequent Subgraph Mining, High Performance Computing

1 Introduction

Frequent graph mining is a well-studied problem with numerous applications in areas such as computational chemistry, bioinformatics, and social networks. For instance, protein structures or protein-protein interactions can be modeled by labeled graphs, and a bioinformatics researcher might be interested in finding common sub-structures within these graphs. Likewise, mining frequent subgraphs from a massive social graph or citation network can help find communities which may be of interest to social scientists. Other applications include graph classification, clustering, and indexing.

The frequent subgraph mining (FSM) problem is divided into two broad categories, namely, finding frequent patterns in either i) a *graph database* comprising multiple input graphs (e.g., a set of chemical compounds), or ii) a *single large input graph* setting (e.g., a social network or citation graph). The FSM task is to enumerate all subgraphs with frequency (or support) above some *minimum support* threshold. In the multiple graphs

Rensselaer Polytechnic Institute, Troy, NY 12180, USA
E-mail: nilothpal.talukder@gmail.com, zaki@cs.rpi.edu

case, frequency is simply the number of graphs that contain the pattern. However, defining the notion of support in a single graph is more challenging, since it is not enough to simply state whether a pattern exists or not. Instead, we have to find all the distinct isomorphisms from the pattern to the input graph. However, this violates the *anti-monotonicity* principle that is required for effective pruning of the output pattern space. That is, a subgraph can have fewer isomorphisms (i.e., lower support) than its supergraph (e.g., assume that the input graph has only one vertex labeled A, which is connected to 100 vertices labeled B, then the vertex A occurs only once, but the edge $A-B$ occurs 100 times). Several anti-monotonic support measures have been proposed for use in the single graph setting [3].

In the graph database case there are usually many, moderately sized input graphs, which can be horizontally partitioned among the compute nodes and mined in a distributed manner. Most existing parallel [4, 19, 12, 6] and mapreduce-based approaches [26, 16] adopt this setting. In contrast, our focus is on the more challenging task of mining patterns from a massive and sparse input graph. Existing sequential solutions for this problem include SiGraM [15] and GraMi [5], whereas existing parallel approaches target only shared-memory (SMP) systems [20]. To our knowledge, there is no current distributed FSM algorithm that can handle a massive input graph that is too large to fit in the memory of a single compute node. The input graph thus has to be partitioned among the nodes, which poses several additional mining challenges.

One challenge is the problem of *false negatives*, i.e., frequent patterns that may be missed because some of their occurrences span across partitions. Another challenge is that the global pattern support determination now requires a reduction operation over the partitions, using information from the potentially exponential number of occurrences of a pattern in each partition. Our distributed solution, called DistGraph (for **Dist**ributed **Graph** Mining), guarantees that all frequent patterns that appear in a sequential approach are found. That means it allows no false negatives (frequent patterns that are not discovered) and no false positives (discovered patterns that are actually not frequent). To maintain scalability and efficiency, our approach minimizes the amount of communication to determine the global support via a set of optimizations. Our main contributions are as follows:

- We develop a distributed solution for mining a single massive graph that leverages effective collective communication primitives for scalable performance.
- Unlike previous approaches that assume the input graph fits in the memory, we assume the network is so large that it will not fit in the local memory at each compute node. Our approach is the first one to partition the massive input graph into different segments, which are mined in a distributed manner.
- We propose a hybrid solution for subgraph mining that utilizes both thread-based parallelism within each compute node and distributed computation across multiple compute nodes. Our method is therefore suitable for a variety of environments, such as distributed systems, shared memory systems, and cluster environments over the cloud.
- We show that DistGraph can scale to a massive network with over a billion vertices and four billion edges. To the best of our knowledge this is the largest network considered, to date, for frequent subgraph mining.

2 Background

In this section, we review some definitions. A *graph* is defined as $G = (V, E)$, where V is a set of *vertices* and $E \subseteq V \times V$ is a set of *edges*. In addition, we assume that L is a labeling function for vertices and edges; we denote by $L(v)$ the label for vertex $v \in V$, and by $L(v_1, v_2)$ the label for the edge $(v_1, v_2) \in E$. In the following, we use $G = (V, E)$ for the *single large input graph*, and we use $P = (V_P, E_P)$ for a *pattern graph*, i.e., one of the subgraphs we wish to mine in G.

Subgraph Isomorphism and Embedding: We say that the pattern P is *subgraph isomorphic* to $G = (V, E)$, denoted as $P \subseteq G$, if there exists an injective function, $\phi : V_P \rightarrow V$ such that: 1) $\forall v \in V_P$, $L(v) = L(\phi(v))$, and 2) $\forall (v_i, v_j) \in E_P$, $(\phi(v_i), \phi(v_j)) \in E$ and $L(v_i, v_j) = L(\phi(v_i), \phi(v_j))$. In this case, the isomorphic subgraph in G comprising the vertices $\phi(v_1), \phi(v_2), \cdots, \phi(v_p)$ (where $p = |V_P|$) is also called an *embedding* of the pattern P in the input graph G. We use the terms isomorphism and embedding interchangeably, since given the isomorphism function ϕ, we can uniquely identify the corresponding embedding (which is a subgraph of G).

Support: For a pattern P and input graph G, let $\Sigma(P) = \{\phi_1, \phi_2, \cdots\}$ denote the set of all isomorphisms/embeddings of P in G. The support of a pattern P is defined based on some function of $\Sigma(P)$. For instance, support can be defined as the cardinality of $\Sigma(P)$, i.e., the number of embeddings of P in G. However, this definition violates the *anti-monotonicity* principle, which requires that the support of a pattern should not be greater than the support of its subgraphs. To address this issue a support measure using the maximum independent set of the overlap graph between embeddings was proposed by Kuramochi et al. [15]. However, this is known to be a NP-hard problem.

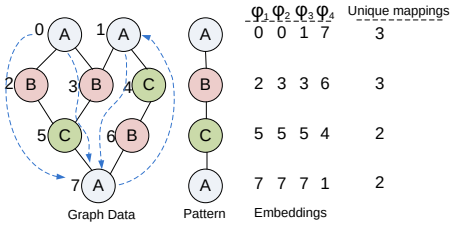


Fig. 1 Support in a single graph

We use the most restricted node (MRN) support [3], also called the minimum image based support, defined as the minimum number of unique vertex mappings over any of the vertices in P. More formally, given the set of embeddings $\Sigma(P) = \{\phi_1, \phi_2, \cdots\}$, the MRN support of P is defined as:

$$\sigma(P) = \min_{v \in V_P} \left\{ |\Phi(v)| \right\} \tag{1}$$

where $\Phi(v)$ denotes the set of unique mappings for a vertex $v \in V_P$, given as

$$\Phi(v) = \bigcup_{i=1}^{|\Sigma(P)|} \phi_i(v)$$

The *relative support* of a pattern, denoted $\sigma_r(P)$, is expressed as a fraction of the number of vertices in the input graph G, i.e., $\sigma_r(P) = \sigma(P)/|V|$.

Fig. 1 shows the embeddings of the pattern $P \equiv A$–B–C–A in the input graph. For instance, one of the embeddings is $\phi_1 = \{0, 2, 5, 7\}$, shown in the first column to the right of the pattern. In total there are four embeddings for P. Therefore, the set of distinct mappings for the first vertex is $\{0, 1, 7\}$. The number of unique mappings over all the pattern vertices are 3, 3, 2 and 2, respectively. Thus, the MRN support of P is $\sigma(P) = \min\{3, 3, 2, 2\} = 2$, or equivalently, its relative support is $\sigma_r(P) = 2/8 = 0.25$ or 25%.

The MRN support also makes intuitive sense, since it avoids over-counting embeddings in the same "locations" in the input graph. For example, consider the pattern $P \equiv A$–B–A. This has two embeddings in the input graph in Fig. 1, namely, $\phi_1 = \{0, 3, 1\}$ and $\phi_2 = \{1, 3, 0\}$. The set of distinct mappings for the first pattern vertex (labeled A) is $\{0, 1\}$, which is also the same for the third vertex (labeled A). On the other hand, the set of distinct mappings for the second vertex (labeled B) is $\{3\}$, resulting in a support value of $\sigma(P) = 1$. Finally, one of the major advantages of using the MRN support is that even if there are an exponential number of embeddings, storing the mappings $\Phi(v)$ across all vertices $v \in V_P$ takes at most polynomial space, namely $O(|V_P| \times |V|)$.

Frequent Single Graph Mining: Given a single input graph G and a user specified parameter called *minimum support threshold*, i.e., $minsup \in \mathbf{Z}$, where $\mathbf{Z}$ denotes the set of positive integers, the task is to discover all subgraph patterns, such that for each pattern P, we have $\sigma(P) \geq minsup$.

3 Related Work

FSM is a well studied problem, in both the graph database [10,14,27] and the single graph [15,5] setting.

Graph Database (multiple graphs): Initial FSM methods used systematic generation of candidate subgraph patterns [10,14] and their level-wise growth via breadth-first search (BFS). Later methods like gSpan [27] and FFSM [9] use canonical ordering of the patterns, and a depth-first (DFS) exploration. Parallel graph mining algorithms targeting SMP systems were proposed in [4] and [19], where they explore different schemes for load balancing among the multiple processors. One of the first distributed methods was [6], which uses a message passing approach for mining molecular graphs. Recently, map-reduce based approaches [26,18,16,2,7] have attracted attention; GPUs have also been exploited [12]. It is important to note that all of these parallel/distributed methods are designed for multiple input graphs, and cannot be adapted for the single graph case due to the underlying algorithmic assumptions.

Single Input Graph: Sequential single input graph mining methods include SUBDUE [8], SiGraM [15], and more recently GraMi [5]. One of the earliest algorithms, SUBDUE searches for the potentially frequent substructures in the graph that can also compress it well. SiGraM uses the maximum independent set based approach for computing support which can be very expensive. GraMi takes a constraint satisfaction based approach for support computation and optimizes the storage of isomorphisms in memory by detecting automorphism groups. A parallel single graph mining method for SMP systems was proposed in [20]. Distributed map-reduce based graphlet and motif discovery methods have also been proposed [21,17], where the task is to mine

the frequency distribution of all vertex induced subgraphs with up to k vertices, for some small k. In contrast, for FSM, there is no restriction on the size of the patterns to be mined, but the minimum support can be used to prune the search space. Also related is the work on finding all matchings of a query subgraph in a larger graph [23, 1, 22]. However, the key difference is that here the query pattern is fixed (i.e., user provided), and one typically has to report one or all matches of the given query in the input graph. On the other hand, FSM is a much harder problem, since it has to first enumerate patterns from a exponentially large search space, and then it has to perform subgraph isomorphism checks to compute the support for each pattern.

Finally, large distributed graph processing platforms (e.g., `spark.apache.org/GraphX`) have become very popular. However, these systems are mainly optimized for vertex centric graph algorithms (e.g., PageRank). The recent Arabesque framework [24] proposes a general embeddings centric or "think like an embedding" paradigm for graph problems like FSM, motif discovery, and clique finding. However, it assumes that the input graph fits in memory of each compute node. We propose the first distributed subgraph mining method for a partitioned input graph, and one that can handle web-scale networks spanning over a billion vertices and edges.

4 DistGraph: Distributed Mining on a Partitioned Input Graph

Our distributed method allows for a massive input graph, that does not fit in the memory of any individual compute node. We adopt a partitioning approach, where we split the input graph into multiple partitions and perform the mining task in a distributed fashion. Furthermore, we design a hybrid algorithm that leverages multi-core CPUs within each compute node and leverages distributed computation over multiple compute nodes.

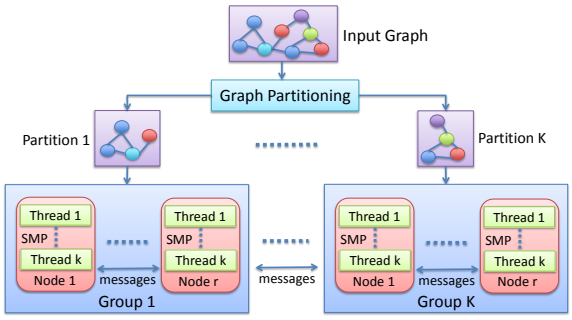


Fig. 2 Distributed Mining Approach

Fig. 2 illustrates our generic distributed mining system framework. Given the massive input graph $G = (V, E)$, we first split it into K partitions, $G_i = (V_i, E_i), 1 \leq i \leq K$. We assume that we have p compute nodes, with each one running a single process with up to k threads. For greater flexibility over the degree of partitioning and parallelism, a group of compute nodes, denoted R_i, handles the same input graph partition G_i. Thus, partition G_i is processed by group R_i. For load balancing, we assume that the number of compute nodes p is a multiple of K, so that each group has an equal number of nodes $r = p/K$. All communication between nodes, either within a group or across groups, is done via message passing, whereas all coordination within a process or compute node is done using shared-memory threads.

Table 1 AllGather and AlltoAll examples

Proc	AllGather		AlltoAll		
	sent	received	sent		received
0	ABC	ABCDEFGHI	ABC		ADG
1	DEF $\Longrightarrow$	ABCDEFGHI	DEF	$\Longrightarrow$	BEH
2	GHI	ABCDEFGHI	GHI		CFI

Collective Communication: Our distributed approach is generic in that it can be instantiated on a tightly-coupled HPC system (e.g., via the message passing interface (MPI) on an IBM Blue Gene supercomputer), or a loosely-coupled platform (e.g., via Hadoop or Spark on a cluster or cloud-based system). We rely only on collective communication operations, described below, that are easy to instantiate on most of the available distributed frameworks (e.g., they are natively available in MPI):

- AllToAll: This is a personalized broadcast operation which rearranges n items of data so that node n gets the nth item. We use a hash function to assign specific data to relevant compute nodes. Also, the sent data can have variable lengths. This is essentially a scatter (or map) operation followed by a gather (or reduce) operation performed by all nodes simultaneously.
- AllGather: Through this operation a compute node can obtain or gather a piece of data from every other node, with all nodes doing this simultaneously. Again, the data can be of variable lengths. The difference between AlltoAll and AllGather is shown in Table 1.
- AllReduce: Finally, this is a reduction operation performed simultaneously by all nodes on a specific operator, such as, SUM. After the operation all nodes will have the same result.

We now outline the challenges posed in distributed mining over a partitioned input graph, and our corresponding solutions, namely: i) eliminating false negative patterns, ii) allowing local pruning, and iii) minimizing communication.

4.1 Eliminating False Negative Patterns via External Neighbors

Recall that the support of a pattern P is given as $\sigma(P) = \min_{v \in V_P}\{|\Phi(v)|\}$, where $\Phi(v)$ denotes the set of unique vertex mappings of $v \in V_P$ in the set of embeddings $\Sigma(P)$ in the input graph G. We also call $\sigma(P)$ the *global support* of P, and say that P is *globally frequent* if $\sigma(P) \geq minsup$.

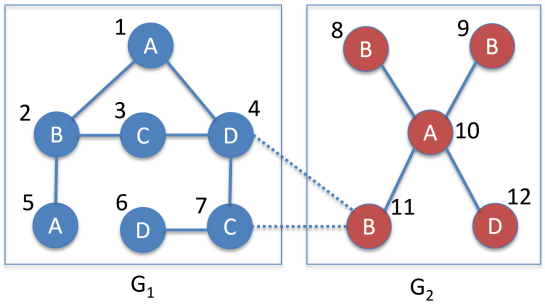


Fig. 3 Partitioning Graph G into G_1 and G_2. Edges that are cut by the partitioning are shown dashed.

The first challenge in mining a partitioned graph is that there can be false negative patterns, i.e., a pattern P that is globally frequent can be missed due to the fact that certain edges involved in subgraph isomorphisms for P span different partitions. For example in Fig. 3, the pattern A–B–C has three embeddings, namely, $(1, 2, 3)$, $(5, 2, 3)$ and $(10, 11, 7)$, and has global support $\sigma(P) = 2$ (since B and C both have two distinct mappings). However, after splitting G into two partitions G_1 and G_2 as shown, the embedding $(10, 11, 7)$ will not be found in G_2 since the edge $(11, 7)$ spans the partitions; it belongs to neither G_1 nor G_2. If we expand G_2 with its 1-hop neighborhood, we can find the embedding. Nevertheless, even with 1-hop overlap, subsequent patterns can still be missed. For example, A–B–C–D has support $\sigma(P) = 2$, but we will not be able to find the embedding $(10, 11, 7, 6)$ or $(10, 11, 7, 4)$, since edges $(7, 6)$ and $(7, 4)$ do not belong to the 1-hop extension of G_2.

To eliminate false negatives, we propose a technique we call *external neighbor expansion* as explained below. Our distributed approach relies on a level-wise or breadth-first approach, where at level l we compute the support of candidate subgraph patterns comprising l edges. The main idea in external neighbor expansion is that before computing patterns at level l, we expand the partitions G_i by requesting from other partitions the neighboring edges (and their end-points) for vertices in the embeddings for each frequent pattern P from the previous level $l - 1$.

More formally, given the input graph G, we first split it into K *initial* partitions, $G_i^0 = (V_i^0, E_i^0)$ for $1 \leq i \leq K$. Vertices $v \in V_i^0$ are called *local vertices*. The graph partition that is available in local memory is always denoted G_i, and initially $G_i \equiv G_i^0$. Now consider level $l = 1$, where we have to compute the support for all the single edge patterns. To avoid false negatives, we have to first extend E_i^0 by adding edges that span across different partitions, such as $(11, 4)$ and $(11, 7)$ in Fig. 3. Thus, we create the extended local partition $G_i^1 = (V_i^1, E_i^1)$, where

$$V_i^1 = V_i^0 \cup \{v |\ (u, v) \in G,\ u \in V_i^0,\ \text{and } v \notin V_i^0\}$$
$$E_i^1 = E_i^0 \cup \{(u, v) |\ u \in V_i^0, v \in V_i^1 \setminus V_i^0\}$$

The extended graph partition that is now available in local memory is $G_i \equiv G_i^1$. Due to external neighbor expansion all relevant edges that span partitions are available in local memory, which ensures that there will be no false negatives when determining the frequent single edge patterns.

Let $\mathcal{G}^1$ denote the set of all frequent single edge patterns. For any frequent pattern $P \in \mathcal{G}^1$, the set of mappings in partition G_i^1 for each vertex $v \in V_P$ is given as: $\Phi_i(v) = \cup_{\phi \in \Sigma_i(P)} \phi(v)$. Before we begin level $l = 2$, we perform external neighbor expansion, by creating expanded local partitions as follows:

$$V_i^2 = V_i^1 \cup \{x |\ (x, y) \in G,\ \text{such that } y \in \Phi_i(v), v \in V_P, P \in \mathcal{G}^1 \text{ and } x \notin G_i^1\}$$
$$E_i^2 = E_i^1 \cup \{(x, y) |\ y \in V_i^1 \text{ and } x \in V_i^2 \setminus V_i^1\}$$

In other words, we make sure that for any input graph vertex x that does not belong to G_i^1, but is a neighbor of some vertex y that belongs to the set of mappings for a frequent single edge pattern P, we request the external edge (x, y) via neighbor expansion. This way, the expanded local partition $G_i^2 = (V_i^2, E_i^2)$ will have all relevant edges for $l = 2$, and the support of possible frequent patterns with two edges can be computed without false negatives.

In general, before we begin mining patterns at any level $l \geq 2$, we make sure that all relevant external neighbors are available locally, where "relevant" means that

these pertain to extensions of some frequent pattern from the previous level. Given the frequent patterns $\mathcal{G}^{l-1}$ from level $l-1$, and the set of mappings $\Phi_i(v)$ in G_i^{l-1} for vertices v in some frequent pattern P, the expanded partitions at level l are obtained as follows:

$$V_i^l = V_i^{l-1} \cup \{x|\ (x,y) \in G,\ \text{such that } y \in \Phi_i(v), v \in V_P, P \in \mathcal{G}^{l-1} \text{ and } x \notin G_i^{l-1}\}$$
$$E_i^l = E_i^1 \cup \{(x,y)|\ y \in V_i^{l-1} \text{ and } x \in V_i^l \setminus V_i^{l-1}\}$$

The expanded partition is denoted as $G_i \equiv G_i^l = (V_i^l, E_i^l)$. The vertices in the set V_i^0 (corresponding to the initial partitioning), are called *local vertices*. The vertices in the set $V_i^l \setminus V_i^0$ are called *external vertices*, and likewise, the edges in the set $E_i^1 \setminus E_i^0$ are called *external edges*. As before, for any pattern P, its embeddings in the expanded partition G_i are denoted as $\Sigma_i(P)$, and the set of mappings in partition G_i for each vertex $v \in V_P$ is then given as $\Phi_i(v) = \cup_{\phi \in \Sigma_i(P)} \phi(v)$. Since the expanded partition comprises both local and external vertices, we can correspondingly divide the set of mappings $\Phi_i(v)$ into *local mappings*, denoted $\Phi_i^l(v)$ and *external mappings*, denoted $\Phi_i^e(v)$, as follows:

$$\text{Local Mappings: } \Phi_i^l(v) = \{u \in G_i|\ \exists \phi \in \Sigma_i(P),\ u = \phi(v), u \in V_i^0\}$$
$$\text{External Mappings: } \Phi_i^e(v) = \{u \in G_i|\ \exists \phi \in \Sigma_i(P),\ u = \phi(v), u \in V_i^l \setminus V_i^0\}$$

Theorem 1 *At any level l, let $G_i \equiv G_i^l$ for $1 \le i \le K$ denote the set of expanded partitions. Then $\Sigma(P) = \bigcup_{i=1}^K \Sigma_i(P)$ for any pattern P.*

Proof: Let P be a pattern with l edges, and let $\phi \in \Sigma(P)$. We prove by induction that $\phi \in \Sigma_i(P)$ for some i. Let P be obtained by adding the pattern edge (u,v) to a globally frequent P' from the previous level, i.e., with $l-1$ edges. Let $(x,y) = (\phi(u), \phi(v))$, where $x, y \in G$. Either (x,y) was already in G_i^{l-1} or it was an external edge that was requested by external neighbor expansion before mining level l. In either case, the edge (x,y) must belong to some expanded partition G_i^l. Thus $\phi \in \Sigma_i(P)$ for some i. ■

Thus, external neighbor expansion ensures the set of embeddings for any pattern P at level l can be found from the embeddings of P in each expanded partition $G_i \equiv G_i^l$. This will guarantee that there are no false negatives.

4.2 Enabling Local Pruning via Local Support

The second challenge we now face is that since the MRN support of a pattern P depends on the minimum number of unique mappings of its vertices, at first glance, it appears that we will not be able to prune any pattern locally. To see this, for a globally frequent pattern P, let $v^* \in V_P$ be the vertex with the minimum number of mappings, i.e., $v^* = \arg\min_{v \in V_P}\{|\Phi(v)|\}$. That is $\sigma(P) = |\Phi(v^*)|$. Now, for each partition G_i, let $\Sigma_i(P)$ be the set of embeddings, and let $\Phi_i(v)$ be the corresponding set of mappings for any $v \in V_P$. Just like the global support in Eq.(1) we could define the local support of P in partition G_i as $\min_{v \in V_P}\{|\Phi_i(v)|\}$. Further, let the minimum cardinality vertex in partition G_i be $v_i^* = \arg\min_{v \in V_P}\{|\Phi_i(v)|\}$. The problem is that different partitions may have different minimum cardinality vertices v_i^*, and they need not match v^*, and we thus we cannot make any guarantees about the global support of P using information from $\Phi_i(v_i^*)$. For example, in Fig. 3, consider the pattern $P \equiv A$–B. The

set of embeddings for P is given as $\Sigma(P) = \{(1,2),(5,2),(10,8),(10,9),(10,11)\}$, and thus the set of mappings for the pattern vertex labeled A is $\{1,5,10\}$, and for B it is $\{2,8,9,11\}$. Thus, the minimum cardinality vertex is A and the global support is $\sigma(P) = 3$. However, in partition G_1^0 the minimum cardinality vertex is B with mappings $\{2\}$, and in G_2^0 the minimum cardinality vertex is A with mappings $\{10\}$. As such there is no relationship between the cardinalities of $|\Phi(v^*)|$ and $|\Phi_i(v_i^*)|$. So at first glance, it appears that our only recourse is to compute the embeddings for all possible patterns in each partition G_i, without any local pruning, followed by a reduction step to compute the global support. We now show that we can do better.

Given the set of embeddings $\Sigma_i(P)$ for (expanded) partition G_i, let $\Phi_i(v)$ be the corresponding set of mappings for any $v \in V_P$. We define the *local support* of a pattern P as:

$$\sigma_i(P) = \max_{v \in V_P} \{|\Phi_i(v)|\}$$

and we say that pattern P is *locally frequent* if

$$\sigma_i(P) \geq \frac{minsup}{K} \tag{2}$$

Theorem 2 *Let $\Sigma(P) = \cup_{i=1}^{K} \Sigma_i(P)$, and let $\sigma(P) \geq minsup$. Then there exists some partition such that $\sigma_i(P) \geq \frac{minsup}{K}$.*

Proof: Assume that for all partitions we have $\sigma_i(P) < \frac{minsup}{K}$. Summing over all partitions we have

$$\sum_{i=1}^{K} \frac{minsup}{K} > \sum_{i=1}^{K} \sigma_i(P), \text{ which implies}$$

$$minsup > \sum_{i=1}^{K} \sigma_i(P) = \sum_{i=1}^{K} \max_{v \in V_P} \{|\Phi_i(v)|\}$$

However, note that

$$\sum_{i=1}^{K} \max_{v \in V_P} \{|\Phi_i(v)|\} \geq \max_{v \in V_P} \left\{ \sum_{i=1}^{K} |\Phi_i(v)| \right\} \geq \min_{v \in V_P} \left\{ \sum_{i=1}^{K} |\Phi_i(v)| \right\} \geq \min_{v \in V_P} \{|\Phi(v)|\}$$

Thus, we conclude that $minsup > \min_{v \in V_P} \{|\Phi(v)|\} = \sigma(P)$, which contradicts the fact that $\sigma(P) \geq minsup$. This means our assumption was false, and there must exist at least one partition such that $\sigma_i(P) \geq \frac{minsup}{K}$. ■

Thus, we need to compute the global support only for those patterns P that are locally frequent in at least one partition. That is, each compute node can find the embeddings in its local (expanded) partition G_i, and we can prune patterns that are not locally frequent in at least one partition. This can cut down on a lot of unnecessary computation to determine the global support.

4.3 Minimizing Communication via Support Bounding

The third challenge we face is how to cut down on the cost of communicating the embeddings $\Sigma_i(P)$ across the distributed system so as to estimate the global set of embeddings $\Sigma(P)$ for each pattern P. As such, a pattern can have an exponential number of embeddings, and likewise there can be an exponential number of embeddings. However, observe that we do not need to communicate $\Sigma_i(P)$ to all other nodes to correctly determine the global support of a pattern P; all we need is to send over the mappings $\Phi_i(v)$ for each vertex in the pattern, since $\Phi(v) = \cup_{i=1}^{K} \Phi_i(v)$, and thus we can determine the global support $\sigma(P) = \min_v |\Phi(v)|$. Nevertheless, note that the size of communicating Φ_i is $O(|V_P| \times |V|)$, which is still too expensive for graphs with billions of vertices. Furthermore, we have to communicate this information not just for one pattern but for very many candidate patterns during the mining process.

Instead of communicating the entire set of mappings for each partition and each vertex, i.e., $\Phi_i(v)$, we communicate only the cardinality values, i.e., we exchange $\{|\Phi_i(v)|\}$, for all $v \in V_P$. This drastically cuts down on the communication cost to only $O(|V_P|)$ per pattern, which is independent of the input graph size, and is orders of magnitude lower than communicating $\Phi_i(v)$, especially in very large graphs. Unfortunately, in this case we can no longer compute the exact global support based on only the cardinality information. We solve this problem via a two-step support bounding technique, described below.

As mentioned earlier, we divide the mappings $\Phi_i(v)$ for each vertex $v \in P$, into local mappings, denoted $\Phi_i^l(v)$ and external mappings, denoted $\Phi_i^e(v)$. The local mappings only involve vertices in the initial vertex partition V_i^0 (i.e., without the neighborhood expansion step). The external mappings for v involve a vertex in some other partition V_j^0 such that $j \neq i$. Note that the following property holds:

$$|\Phi_i(v)| = |\Phi_i^l(v)| + |\Phi_i^e(v)|$$

Now, instead of communicating $|\Phi_i(v)|$, we communicate $|\Phi_i^l(v)|$ and $|\Phi_i^e(v)|$.

Next, we derive a lower-bound and upper-bound on the support of P. Define the *lower-support estimate* of P as follows:

$$\widehat{\sigma}_l(P) = min_{v \in V_P} \left\{ \sum_{i=1}^{K} |\Phi_i^l(v)| \right\} \tag{3}$$

The theorem below shows that $\widehat{\sigma}_l(P)$ is indeed a lower-bound for the true global support $\sigma(P)$.

Theorem 3 *It holds that $\widehat{\sigma}_l(P) \leq \sigma(P)$. Thus, if $\widehat{\sigma}_l(P) \geq minsup$, then $\sigma(P) \geq minsup$.*

Proof: For any vertex $v \in V_P$, we have

$$\bigcup_{i=1}^{K} \Phi_i^l(v) \subseteq \bigcup_{i=1}^{K} \Phi_i(v) = \Phi(v)$$

However, since $\Phi^l(v)$ are all disjoint by definition, this implies $\sum_{i=1}^{K} \left|\Phi_i^l(v)\right| \leq |\Phi(v)|$. Therefore, $\widehat{\sigma}_l(P) = min_{v \in V_P} \left\{ \sum_{i=1}^{K} |\Phi_i^l(v)| \right\} \leq min_{v \in V_P} \{|\Phi(v)|\} = \sigma(P)$. Hence, $\widehat{\sigma}_l(P) \geq minsup$ implies $\sigma(P) \geq minsup$. ■

Finally, we derive an upper-bound on the support of P. Define the *upper-support estimate* of P as follows:

$$\widehat{\sigma}_u(P) = min_{v \in V_P} \left\{ \sum_{i=1}^{K} |\Phi_i(v)| \right\} = min_{v \in V_P} \left\{ \sum_{i=1}^{K} \left(|\Phi_i^l(v)| + |\Phi_i^e(v)| \right) \right\} \quad (4)$$

The theorem below shows that $\widehat{\sigma}_u(P)$ is an upper-bound for the true global support $\sigma(P)$.

Theorem 4 *It holds that $\widehat{\sigma}_u(P) \geq \sigma(P)$. Thus, if $\sigma(P) \geq minsup$, then $\widehat{\sigma}_u(P) \geq minsup$.*

Proof: For any vertex $v \in V_P$, we have $\bigcup_{i=1}^{K} \Phi_i(v) = \Phi(v)$, which in turn implies $\sum_{i=1}^{K} |\Phi_i(v)| \geq |\Phi(v)|$. Therefore, $min_{v \in V_P} \left\{ \sum_{i=1}^{K} |\Phi_i(v)| \right\} \geq min_{v \in V_P} \{|\Phi(v)|\} = \sigma(P)$. Thus, $\sigma(P) \geq minsup$ implies $\widehat{\sigma}_u(P) \geq minsup$. ■

Then, by the counter-positive of Thm. 4, it must be true that if $\widehat{\sigma}_u(P) < minsup$, then $\sigma(P) < minsup$.

The two-step support bounding technique is then used to prune patterns as follows. First, we check whether for a pattern P it holds that $\widehat{\sigma}_l(P) \geq minsup$. In this case, by Thm. 3, P must be globally frequent, and thus we can simply output the pattern. On the other hand, if $\widehat{\sigma}_l(P) < minsup$, then we cannot be sure whether P is globally frequent or not. Second, we check whether $\widehat{\sigma}_u(P) < minsup$. In this case, by the counter-positive of Thm. 4, P cannot possibly be globally frequent, and we can thus prune it from further consideration. On the other hand, if $\widehat{\sigma}_u(P) \geq minsup$, we call such patterns *potentially frequent patterns*, since $\widehat{\sigma}_u(P) \geq minsup$, does not imply that $\sigma(P) \geq minsup$. For correctness, we have to determine their exact global support and eliminate the infrequent ones. To summarize, the basic idea behind two-step support bounding approach to minimize communication is to determine patterns that are definitely globally frequent, and to eliminate patterns that cannot possibly be globally frequent. For the remaining patterns we determine their exact global support and retain only the frequent ones. This approach is very effective in reducing the amount of communication in the distributed system.

5 DistGraph: Algorithm Details

Alg. 1 outlines the DistGraph approach, which is executed by a group of compute nodes R_i working on the graph partition $G_i = (V_i, E_i)$ as illustrated in Fig. 2. Starting from the single edge patterns, the candidates are extended in a level-by-level or breadth-first (BFS) fashion. This choice is dictated by the fact that in the partitioned approach, each compute node can only obtain the local support of a pattern. If we perform a DFS exploration, there will be an overwhelming number of potentially frequent patterns. Instead, a BFS exploration allows one to eliminate the globally infrequent patterns at the current level before proceeding to the next level.

Let $\mathcal{P}_i$ be the set of globally frequent patterns from the previous level; initially $\mathcal{P}_i$ contains the empty graph. In Step 4, the patterns $P \in \mathcal{P}_i$ are extended by one edge and each valid candidate pattern is checked to see whether it is locally frequent (w.r.t. partition G_i) or not, using Eq. (2). Locally frequent candidates are added to the set $\mathcal{P}_{lf}$, and those not locally frequent are added to the set $\mathcal{P}_{nlf}$. This step is

Algorithm 1 Distributed Single Graph Mining

DistGraph (Graph partition G_i, *minsup*, threads k)
1: $\mathcal{P}_i \leftarrow \{\emptyset\}$
2: **repeat**
3: Initialize locally frequent and non-frequent lists, $\mathcal{P}_{lf} \leftarrow \emptyset$, $\mathcal{P}_{nlf} \leftarrow \emptyset$
4: Compute-Local-Support (G_i, $\mathcal{P}_i$, $\mathcal{P}_{lf}$, $\mathcal{P}_{nlf}$)
5: ($\mathcal{P}_i$,$\mathcal{C}_i$) $\leftarrow$ Support-bound-Pruning (*minsup*, P_{lf}, P_{nlf})
6: $\mathcal{P}_i \leftarrow$Exact-Global-Support(G_i,$\mathcal{C}_i$,$\mathcal{P}_i$)
7: Distribute patterns $\mathcal{P}_i$ (by AllGather) among the compute groups
8: External-Neighbor-Expansion(G_i,$\mathcal{P}_i$)
9: **until** there is no globally frequent pattern in $\mathcal{P}_i$, determined by AllReduce($|\mathcal{P}_i|$) = 0

performed across all compute nodes, using k threads each. Next, in Step 5, we use two-step support-bounding to prune patterns that are definitely globally frequent and infrequent. The patterns that are guaranteed to be globally frequent based on the local-only support (via Eq. (3)) are stored in a new set $\mathcal{P}_i$. On the other hand, the patterns that are potentially globally frequent (via Eq. (4)) are stored in $\mathcal{C}_i$. For patterns in $\mathcal{C}_i$, we determine their exact global support in Step 6, and then remove the globally infrequent ones from $\mathcal{C}_i$. After taking a union with $\mathcal{C}_i$, the set $\mathcal{P}_i$ contains patterns that are globally frequent (i.e., have $\sigma(P) \geq minsup$), and these are now exchanged with all groups in Step 7, so that each group has the full set of globally frequent patterns (across the entire system) before the next level begins. However, each group retains only those patterns that are relevant to its partition G_i. Finally, to eliminate false negatives, in Step 8 we fetch relevant external neighbors from other partitions to create the expanded local partitions G_i. The level-wise approach is then repeated until no more globally frequent patterns are found.

We now give detailed description of our approach for partitioning the input graph, computing local support, estimating the global support, support-bound based pruning to minimize communication, determining exact global support, and fetching external neighbors.

5.1 Input Graph Partitioning

Let $G = (V, E)$ denote the massive input graph, and let $\{V_1^0, V_2^0, \cdots V_K^0\}$ denote a K-way partitioning of the vertex set, such that partitions are pair-wise disjoint (i.e., $V_i^0 \cap V_j^0 = \emptyset$) and together encompass all the vertices (i.e., $\cup_i V_i^0 = V$). We denote the i-th initial partition as $G_i^0 = (V_i^0, E_i^0)$, where the edge set E_i^0 includes all edges in E both of whose end-points are in V_i^0, i.e., $E_i^0 = \{(u, v) | (u, v) \in E \text{ and } u, v \in V_i^0\}$. While any partitioning scheme, even a random one, can be used for graph partitioning, for good performance we need the partitions to be balanced and to minimize the edge cut (the number of vertices with an endpoint in another partition). Graph partitioning is a well-studied problem and solutions are generally derived using heuristics and approximation algorithms, since the balanced partitioning problem is NP-complete. We use the efficient PARMETIS [11] tool, which results in a good quality partitioning.

Furthermore, before the mining begins, we include the 1-hop overlap of the bordering edges between two neighboring partitions. That is, for any two partitions $G_i^0 = (V_i^0, E_i^0)$ and $G_j^0 = (V_j^0, E_j^0)$, if there exists an edge (u, v) with $u \in V_i^0$ and $v \in V_j^0$, then we add the edge (u, v) to both E_i^0 and E_j^0. Thus, given the initial K-way partitioning of the vertices $\{V_1^0, \cdots, V_k^0\}$, the expended graph partitions $G_i^1 = (V_i^1, E_i^1)$

are defined as follows:

$$E_i^1 = \{(u, v) \in E \mid u \in V_i^0 \text{ and } v \in V\}$$
$$V_i^1 = V_i^0 \cup \{v \mid (u, v) \in E, u \in V_i \text{ and } v \notin V_i\}$$

After 1-hop expansion, the expended partitions are given as $G_i \equiv G_i^1$. Note that in the discussion below we avoid using the superscripts for the expanded partitions to reduce clutter in the notation; G_i always refers to the expanded local partitions before the next level begins.

5.2 Computing Local Support

Given the current (expanded) local partition G_i, and the set of globally frequent patterns $\mathcal{P}_i$ from the previous level, Alg. 2 shows the steps to extend each pattern $P \in \mathcal{P}_i$ and compute its local support. This function is carried out in parallel by all the k threads within each compute node, as well as collectively by all the compute nodes in each group R_i. That is, the patterns in $\mathcal{P}_i$ are divided up equally among all available nodes/threads.

Algorithm 2 Compute Local Support

COMPUTE-LOCAL-SUPPORT(G_i, $\mathcal{P}_i$, $\mathcal{P}_{lf}$, $\mathcal{P}_{nlf}$)

```
1: for each pattern P ∈ P_i do
2:    Create Embeddings Σ_i(P)
3:    E_i(P) ← GET-EXTENSIONS(G_i, P, Σ_i)
4:    Compute local support for all extensions in E_i(P)
5:    for each e ∈ E_i(P) do
6:       Extend P by e to obtain P'
7:       if P' is canonical then
8:          Add P' to P_lf if P' is locally frequent, else add P' to P_nlf
9:       end if
10:   end for
11:   for each v ∈ V_P do
12:      Compute and store cardinalities |Φ_i^l(v)| and |Φ_i^e(v)|
13:   end for
14:   Discard Σ_i(P)
15: end for
```

For a specific pattern P, each processing element first computes its embeddings $\Sigma_i(P)$ in G_i. Next, from each embedding, the set of all edge extensions are collected in $\mathcal{E}_i(P)$. For systematic and non-redundant graph enumeration, we use the DFScode for canonicality checking and rightmost path edge extensions [27], as detailed below in Alg. 3. Given the set of all extensions from each embedding of P, we can compute the local support of each extension. The pattern P is extended by each of the new edges $e \in \mathcal{E}_i(P)$ to obtain the pattern P', which is retained only if it is canonical. Finally, P' is either added to $\mathcal{P}_{lf}$ if it is locally frequent, or else it is added to $\mathcal{P}_{nlf}$. It is important to note that we discard all of the embeddings $\Sigma_i(P)$ after computing the cardinalities of the vertex mappings $|\Phi_i^l(v)|$ and $|\Phi_i^e(v)|$ for each vertex $v \in V_P$, otherwise the amount of memory required to retain all embeddings can easily exceed available resources.

Algorithm 3 Get Extensions

GET-EXTENSIONS(Graph partition $G_i = (V_i, E_i)$, P, Σ_i)

```
1: for each φ ∈ Σ_i do
2:   for each u ∈ right-most path of φ do
3:     Get valid forward and backward extensions E_f(u) and E_b(u), resp.
4:     for each (u, v) ∈ E_f(u) such that v ∉ V_i do
5:       j ← partition id for vertex v
6:       Ext_i[P] ← Ext_i[P] ∪ {(v, j)}
7:     end for
8:   end for
9: end for
```

We now discuss Alg. 3 which identifies the valid extensions, and also determines which external vertices are be fetched later to create the expanded partitions G_i. For a pattern P, Alg. 3 finds all valid backward and forward extensions of its embeddings in Σ_i. Backward extensions are those that introduce cycles in an existing patterns, whereas forward extensions introduce a new vertex with the corresponding edge. Note that extensions happen only along the right-most path; see [27] for more details. Note that only forward extensions can introduce an external vertex. That is, if (u, v) is a forward extension where $u \in V_i$ but $v \notin V_i$ we would need to fetch this vertex for the next extension if P turns out to be globally frequent. So in steps 4-6, we collect such external neighbors for P and store them in the set $Ext_i[P]$ for use later in the external neighbor expansion step. So we add to $Ext_i[P]$ the pair (v, j) where v is the external vertex id, and j is the partition id for v. Then, in Alg. 6 the actual requests are made to the corresponding partitions only if P is found to be globally frequent.

5.3 Support-Bound Based Pruning

Alg. 4 outlines the steps in computing the lower and upper support estimates for all locally frequent patterns P, so that we can determine the globally frequent patterns. We distribute the task of support-bound estimation among all compute nodes.

First, only those patterns that are locally frequent in some partition are considered. To achieve this, in Step 1, each pattern in $\mathcal{P}_{lf}$ is assigned to a designated compute node via hashing on its unique DFScode. Each compute node distributes the patterns and their vertex mapping cardinalities $|\Phi_i^l(v)|$ and $|\Phi_i^e(v)|, \forall v \in V_P$, to the designated compute node via ALLTOALL. All compute nodes accumulate the received vertex mapping cardinalities for their assigned patterns and determine their lower-bound and upper-bound supports, $\widehat{\sigma}_l(P)$ and $\widehat{\sigma}_u(P)$, respectively. If pattern P satisfies $\widehat{\sigma}_l(P) \geq minsup$, then it is definitely globally frequent, and is added to the set $\mathcal{F}$. Otherwise, if P satisfies $\widehat{\sigma}_u(P) \geq minsup$, then we add it to the set of potentially frequent patterns $\mathcal{G}$. Now, the patterns with $\widehat{\sigma}_u(P) < minsup$ comprise the set $\mathcal{Q}$; such patterns can be infrequent either because they are globally infrequent, or it may be because they were locally infrequent in one or more partitions, and thus their vertex mapping cardinalities were not communicated in Step 1. To eliminate this possibility, we first communicate the patterns in $\mathcal{Q}$ to all compute nodes via ALLGATHER (Step 12). The unified set of such patterns is denoted $\mathcal{Q}^G$. Next, each compute node looks up the vertex mapping cardinalities for these patterns in $\mathcal{P}_{nlf}$, and sends them over to the designated compute node via ALLTOALL in Step 13. Note that we do not consider patterns in $\mathcal{P}_{lf}$, since they were sent earlier. Given the mapping cardinalities, each compute node can again

Algorithm 4 Support-bound Pruning

SUPPORT-BOUND-PRUNING ($minsup$, $\mathcal{P}_{lf}$, $\mathcal{P}_{nlf}$)

1: Distribute $P \in \mathcal{P}_{lf}$ to all compute nodes via ALLTOALL using DFS code for P; also send $|\Phi_i^l(v)|$ and $|\Phi_i^e(v)|$, for all $v \in V_P$
2: Initialize sets $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{Q}$ to $\emptyset$
3: **for** each P mapped to self **do**
4: **if** $\widehat{\sigma}_l(P) \geq minsup$ **then**
5: $\mathcal{F} \leftarrow \mathcal{F} \cup \{P\}$
6: **else if** $\widehat{\sigma}_u(P) < minsup$ **then**
7: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{P\}$
8: **else**
9: $\mathcal{G} \leftarrow \mathcal{G} \cup \{P\}$
10: **end if**
11: **end for**
12: Perform ALLGATHER for all patterns $P \in \mathcal{Q}$ into $\mathcal{Q}^G$
13: Perform ALLTOALL for $|\Phi_i^l(v)|$ and $|\Phi_i^e(v)|$ for all hashed $P \in \mathcal{Q}^G$ to corresponding compute nodes, if $P \in \mathcal{P}_{nlf}$
14: **for** each $P \in \mathcal{Q}^G$ **do**
15: **if** $\widehat{\sigma}_l(P) \geq minsup$ **then**
16: $\mathcal{F} \leftarrow \mathcal{F} \cup \{P\}$
17: **else if** $\widehat{\sigma}_u(P) \geq minsup$ **then**
18: $\mathcal{G} \leftarrow \mathcal{G} \cup \{P\}$
19: **end if**
20: **end for**
21: ALLGATHER to get all globally frequent patterns $\mathcal{F}$ and store in $\mathcal{P}_i$ if they are in either $\mathcal{P}_{lf}$ or $\mathcal{P}_{nlf}$; Similarly, store all potentially globally frequent patterns $\mathcal{G}$ in $\mathcal{C}_i$
22: Return $(\mathcal{P}_i, \mathcal{C}_i)$

compute the estimates $\widehat{\sigma}_l(P)$ and $\widehat{\sigma}_u(P)$ for patterns in $\mathcal{Q}^G$, and add them to either $\mathcal{F}$ or $\mathcal{G}$, accordingly.

It is important to stress that DISTGRAPH does not communicate the vertex mapping cardinalities for all patterns in $\mathcal{P}_{lf}$ and $\mathcal{P}_{nlf}$. The global support is estimated only for those patterns that are locally frequent in at least one partition. Next, only for these patterns, we communicate the mapping counts in $\mathcal{P}_{nlf}$, if we determine that the estimated global support is below *minsup*. Our approach thus performs only the necessary amount of communication. Also, whereas compute nodes can be selective in not requesting the mapping counts from the nodes where a pattern P is already locally frequent (and thus we will have access to its mapping counts from Step 1), such cases are typically too few to outperform the efficient collective communication done in Step 12.

5.4 Computing Exact Global Support

To determine the exact global support for a pattern P we need to have access to all its vertex mappings $\Phi_i(v)$ in each partition $G_i = (V_i, E_i)$ for each $v \in V_P$. However, after the local support computation phase, we have access to only the mapping cardinalities $|\Phi_i^l(v)|$ and $|\Phi_i^e(v)|$ for each vertex, but not the mappings $\Phi_i(v)$ or the embeddings $\Sigma_i(P)$. The vertex mappings $\Phi_i(v)$ can be obtained in several ways: i) *memory-based:* keep the mappings $\Phi_i(v)$ in memory for all possible patterns, but communicate the external mappings only if $P \in \mathcal{C}_i$. This approach consumes too much memory, since it is not possible to retain the mappings for all patterns (both locally infrequent or

frequent) for all except the smallest of graphs with few embeddings per pattern; ii) *file-based:* we can store the set of mappings Φ_i to local disk for all (infrequent and frequent) patterns. For each pattern $P \in \mathcal{C}_i$, we can then read the mappings from disk; iii) *regenerate embeddings:* This is our default approach, where we regenerate or recompute the embeddings for each pattern in a distributed manner. We experimentally evaluate these approaches in Sec. 6.

Algorithm 5 Compute Exact Global Support

EXACT-GLOBAL-SUPPORT(G_i, $\mathcal{C}_i$, $\mathcal{P}_i$)

1: AllGather $\mathcal{C}_i$, i.e., the candidates for exact global support check, into $\mathcal{C}^G$
2: **for** each $P \in \mathcal{C}^G$ **do**
3: If $P \in \mathcal{P}_{lf} \cup \mathcal{P}_{nlf}$, regenerate vertex mappings $\Phi_i^l(v)$ and $\Phi_i^e(v), \forall v \in V_P$ (Optional: for memory/file based versions read from memory/file, respectively)
4: Perform AllToAll to send the external mappings $\Phi_{ij}^e(v)$ to group R_j
5: Update local mappings $\Phi_i^l(v)$ by including mappings received from other groups, i.e., $\Phi_i^l(v) \leftarrow \Phi_i^l(v) \cup \Phi_{ji}^e(v)$
6: Compute Global support $\sigma(P) = \min_{v \in V_P} \sum_{i=1}^{K} |\Phi_i^l(v)|$ by AllReduce
7: If $\sigma(P) < minsup$ remove P from $\mathcal{C}_i$
8: **end for**
9: $\mathcal{P}_i \leftarrow \mathcal{P}_i \cup \mathcal{C}_i$
10: Return $\mathcal{P}_i$

Alg. 5 describes how we determine the exact global support for all patterns $P \in \mathcal{C}_i$. We divide up the set $\mathcal{C}_i$ among the compute nodes in the group R_i to carry out the exact support determination in a distributed manner. Further, all the k threads are also involved in regenerating the mappings $\Phi_i(v)$ in parallel for all $v \in V_P$ in Step 3. The external mappings for v involve a vertex in some other partition V_j ($j \neq i$). Therefore, $\Phi_i^e(v) = \bigcup_{j \neq i}^{K} \Phi_{ij}^e(v)$, where $\Phi_{ij}^e(v)$ denotes the set of external mappings in partition V_i that come from partition V_j. Also, we ignore $\Phi_{ii}^e(v)$, since it is essentially $\Phi_i^l(v)$. The local mappings do not have to be communicated to any other group. However, we use the AllToAll operation to communicate to group R_j the external mappings $\Phi_{ij}^e(v)$ involving initial partition V_j (Step 4). At the end of this exchange, each group R_j has all the local vertex mappings for each vertex v in pattern P, which it uses to update $\Phi_i^l(v)$ (Step 5). Formally, we can write $\Phi_i^l(v) = \Phi_i^l(v) \cup \{\cup_{j \neq i}^{K} \Phi_{ji}^e(v)\}$. Since a graph vertex can belong to at most one partition, we can then compute the true global support of P as $\sigma(P) = \min_{v \in V_P} \sum_{i=1}^{K} |\Phi_i^l(v)|$, which is accomplished by performing an AllReduce operation on the local vertex mappings and considering the minimum value (Step 6). Finally, the pattern is marked as infrequent if the support is below *minsup*, and each group removes such patterns from $\mathcal{C}_i$. Finally, the remaining patterns in $\mathcal{C}_i$ are globally frequent, and are thus added to the set $\mathcal{P}_i$.

5.5 External Neighbor Expansion

To avoid false negatives, the external neighbor expansion in Alg. 6 ensures that all edges needed for computing the embeddings are available locally. For each pattern P that is globally frequent, we have to collect the external neighbor requests for any vertex in $\Phi_i(v)$ for $v \in V_P$. We collect this list for each pattern, denoted $Ext_i[P]$, when

we compute the local support, as detailed in Alg. 3. Recall that $Ext_i[P]$ comprises pairs of the form (v, j), where $v \in V_j$, with $j \neq i$. The first step is to do an AllToAll to request vertex v from partition G_j. Next, another AllToAll is used to respond back with the degree and neighbor list. Finally, each partition G_i extends its edge and vertex sets. The external neighbor requests are carried out simultaneously by dividing up the requests among nodes in a group R_i. It is worth remarking that the neighborhood extension does not significantly expand the memory footprint. This is because we request the external neighbors only for the embeddings of frequent patterns, and as computation progresses, fewer and fewer of these are found. It is possible to further reduce the local memory by actually deleting from G_j vertices and edges that do not participate in the mappings for any frequent pattern.

Algorithm 6 External Neighbor Expansion

External-Neighbor-Expansion(G_i, $\mathcal{P}_i$)

1: Perform AlltoAll for $(v, j) \in Ext_i[P], \forall P \in \mathcal{P}_i$ sending global vertex ids v to the corresponding partitions j
2: Reply with AlltoAll with the degree and the neighbor lists of the global ids
3: Extend the partition G_i with the neighbors, and mark all the requested global vertex id v's as own vertex, i.e. $V_i \leftarrow V_i \cup \{v\}$

5.6 DistGraph Correctness

Our method guarantees correctness, i.e., it produces no false positive or false negative patterns, as proved in the theorem below.

Theorem 5 *Alg. 1 is correct, i.e., it outputs a pattern P if and only if $\sigma(P) \geq minsup$.*

Proof: We prove the correctness by induction. Let $\mathcal{P}'_i$ be the set of globally frequent patterns from the previous level comprising patterns with $r-1$ edges.

Let P be a pattern with r edges, and let it be globally frequent, i.e., $\sigma(P) \geq minsup$. Monotonicity of $\sigma(P)$ guarantees that there exists a pattern in $\mathcal{P}'_i$ from which P can be obtained by an edge extension. In the Compute-Local-Support *step, we try all extensions of all patterns $P' \in \mathcal{P}'_i$, and therefore P will be one of the candidate extensions. Given the invariant $\Sigma(P) = \cup\Sigma_i(P)$, Thm. 2 guarantees that P will be locally frequent in at least one partition, so that $P \in \mathcal{P}_{lf}$ for some partition. In* Support-bound-Pruning *either $\widehat{\sigma}_l(P) \geq minsup$ and thus it will be added to the new set of globally frequent patterns $\mathcal{P}_i$, or $\widehat{\sigma}_u(P) \geq minsup$ and thus it will be added to the set of potentially frequent patterns $\mathcal{C}_i$. In the latter case, after exact global support computation in* Exact-Global-Support*, P will be found to be globally frequent and thus will be added to $\mathcal{P}_i$, the set of globally frequent patterns at the current level r.*

On the other hand, assume that P is not globally frequent, i.e., $\sigma(P) < minsup$. In Compute-Local-Support *if P is not locally frequent in at least one partition, we are done. Otherwise, if P is locally frequent in some partition, it will be added to the corresponding set $\mathcal{P}_{lf}$. Next in* Support-bound-Pruning *if $\widehat{\sigma}_u(P) < minsup$, then P will be pruned and we are done. Otherwise, P will be added to the set $\mathcal{C}_i$, but will be eliminated in* Exact-Global-Support *when we determine its exact global support.*

Thus, a pattern P is added to $\mathcal{P}_i$ if and only if $\sigma(P) \geq minsup$. To complete the proof, we have to show that the base case holds. Note that initially, when $r = 0$, we start with the empty graph $\emptyset$ which is trivially frequent. For $r = 1$, this empty graph will be extended by all possible edges in each partition, and thus the set of all frequent patterns with a single edge will be correctly found. Finally, the algorithm will terminate when each $\mathcal{P}_i$ is empty, which implies there are no new frequent patterns to be found. ■

The worse-case time complexity for mining all frequent subgraphs is clearly exponential. To see this, consider an input graph G with $|V| = n$ vertices and $|E| = m$ edges. Assume that the largest frequent pattern has l edges, then the number of possible frequent patterns are: $\sum_{i=0}^{l} \binom{m}{l} = \frac{m^{l+1}-1}{m-1} = O(m^l)$. Next, for any pattern P with l edges, we have to compute its support, which can take $O\left(\binom{m}{l}\right) = O(m^l)$ time. Thus, the worst case time complexity for frequent subgraph mining is $O(m^l \times m^l) = O(m^{2l})$. Our distributed approach reduces the computational time to $O(m^{2l}/p)$ in the ideal case, where p is the number of compute elements. For further results on the complexity of subgraph mining problems see [28, 13].

6 Experiments

All our distributed experiments are performed on up to 2048 IBM Blue Gene/Q (BG/Q) nodes. Each node consists of a 16-core 1.6 GHz A2 processor, with 16 GB of DDR3 memory. DistGraph is implemented in C++, compiled using g++ (v. 4.4.7) and -O3 optimization flag. We use the OpenMP (v. 3.0) library for thread-based parallelism within compute nodes, and the portable, open-source and freely available MPI implementation MPICH2 (v. 1.5) for distributed computation across nodes. We compared our sequential implementation with GraMi [5], which is written in Java, and is compiled using JDK v. 1.7. Our DistGraph code is available for download at `https://github.com/zakimjz/DistGraph`.

Table 2 Datasets and their Properties: Number of vertices $|V|$, edges $|E|$, labels $|L|$, avg. degree AD, clustering coefficient CC

dataset	$\|V\|$	$\|E\|$	$\|L\|$	AD	CC
PDB1 (small)	20,226	83,356	22	8.2	0.57
PDB2 (medium)	2,020,188	7,905,260	22	3.9	0.55
PDB3 (large)	17,400,398	68,599,675	22	7.9	0.55
PDB4 (16x)	278,406,368	1,097,594,800	22	7.9	0.55
PDB5 (32x)	556,812,736	2,195,189,600	22	7.9	0.55
PDB6 (64x)	1,113,625,472	4,390,379,200	22	7.9	0.55
Patent	2,942,159	14,275,931	37	9.7	0.07
Youtube	4,584,572	23,236,009	80	5.8	0.19

6.1 Datasets

We used three main datasets in our experiments, namely *PDB*, *Patent* and *Youtube*. The properties of the datasets are shown in Table 2. The PDB datasets (PDB1–6) were

Table 3 Partitions and External Neighbors

dataset	max comp.	num partitions (avg. ext neighbors)
PDB1	4,477	4 (688), 8 (764)
PDB2	7,856	16 (57,246), 32 (38,625), 64 (23,801)
PDB3	11,568	32 (350,144), 64 (213,725), 128 (109,570)
Patent	2,934,409	32 (128,431), 64 (75,694), 128 (43,597)
Youtube	4,583,987	128 (71,606), 256 (38,628)

created from the 3D distance matrices using different number of protein 3D structures from the RCSB Protein Data Bank (`www.rcsb.org/pdb`). The datasets PDB4, PDB5 and PDB6 are actually 16x, 32x and 64x replications of PDB3, respectively, used for scalability experiments. Our largest graph PDB6 consists of more than one billion vertices and 4 billion edges. The effective diameter for the PDB graphs (for 90% of the node pairs) is between 24 and 32 hops.

The Patent dataset (`www.nber.org/patents`) consists of pairwise citation data of the US patents granted between Jan 1963 and Dec 1999. We considered the "grant year" of the patents as the vertex labels. The effective diameter (90%) for Patent is 9.5. The Patent dataset has some hub vertices with very high degrees. We use the hub-duplication technique [25] to deal with such high degree nodes. If any node v has degree more than 30, we make several duplicates of that node, say $v_1, v_2, \cdots$, and redistribute the neighbors of v among the duplicates v_i, so that no node has degree more than 30. Finally, we connect v and all v_i in a clique. Note that no interactions are lost, and this technique allows us to avoid the potential explosion in the number of embeddings for hub nodes.

The Youtube dataset (`netsg.cs.sfu.ca/youtubedata`) lists the crawled video ids and related videos for each video posted from Feb 2007 to Jul 2008. We restricted the maximum vertex degree to 15. We created 10 buckets for rating (scale 0 to 5) and 8 buckets for length of the video. Combining these two we generated 80 different vertex labels. For all the datasets the edges are unlabeled. The effective diameter (90%) for Youtube it is 7.7.

Both Patent and Youtube have a giant component that comprises over 99.6% of the vertices, and thus can easily be split into K parts. However, the PDB datasets comprise many components (each being a separate protein structure), and straightforward K-way split results in partitions with no external edges. Since we desire external neighbors, we split each protein into K parts provided it has at least 64 vertices, which results in many external edges, and the K parts of a single protein are then assigned to different partitions. Table 3 shows the average number of external edges for different number of partitions, respv., for the different datasets. For instance, Youtube has one giant component comprising 4,583,987 vertices, and it has on avg. 71,606 external neighbors with 128 partitions.

6.2 Performance results

We study the performance of our distributed algorithm by varying one of the three different parameters, namely the number of compute nodes (p), threads (k), and partitions (K), while keeping the other two fixed. All our performance results are shown as barplots. In the plots the left Y axis shows the total time, and the bars show the time for different methods with different minimum support values. *A value s on the*

X axis indicates minsup of s% with respect to $|V|$, i.e., we are using relative support for each pattern. So for a pattern to be frequent it must satisfy the condition $\sigma(P) \geq |V| \times s/100$. In most cases the right Y axis records the speedup with respect to a baseline method in the same plot.

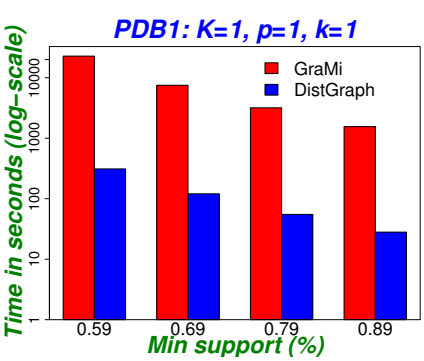


Fig. 4 PDB1: Sequential Comparison of DistGraph with GraMi

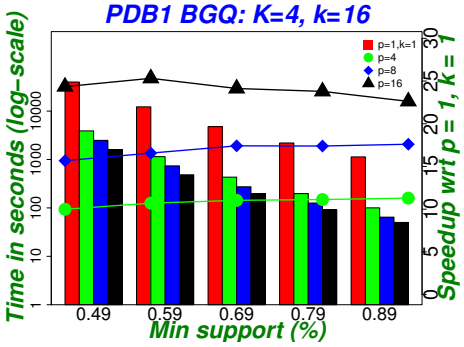


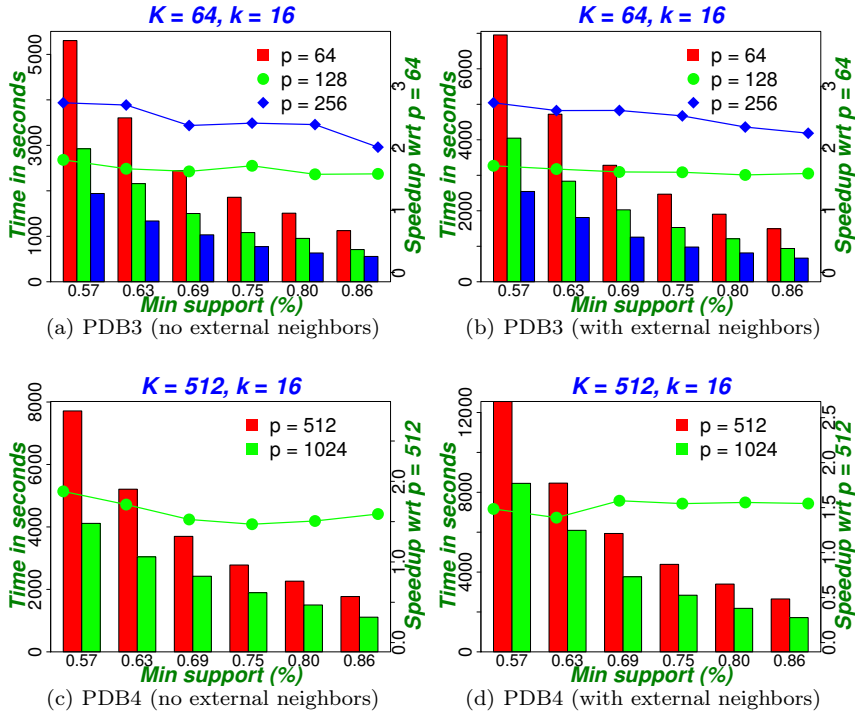
Fig. 5 DistGraph on PDB1: Sequential ($p = 1, k = 1$) and Distributed ($p = 4, 8, 16$)

Sequential Comparison: We compared a sequential run of DistGraph with the state-of-art sequential single graph mining method GraMi [5] on the small PDB1 dataset. Since the Java code cannot be run on the Blue Gene/Q system, we test on a 16-core, 2.9Ghz AMD Opteron 6272 processor with 256GB memory. Both methods were run using a single core. The results are shown in Fig. 4. We can see that sequential DistGraph is an order of magnitude faster (the y-axis is in log-scale). For $minsup = 0.59\%$, GraMi took 22787s versus 311s for DistGraph (i.e., 73 times faster). Although the comparison is not entirely fair, since GraMi is in Java and our code is in C++, we report these timings to show that we have a very efficient implementation. That is, when reporting the distributed performance below, we are not using a slow sequential method that can artificially boost speedup numbers.

Varying Compute Nodes: We show the effect of varying the number of compute nodes. Fig. 5 compares the sequential run of DistGraph (with $p = 1, k = 1$), against using 4, 8, and 16 compute nodes on BG/Q for PDB1. We obtain speedups of 10 with $p = 4$, speedup of 16 with $p = 8$, and a speedup of 25 with $p = 16$ compute nodes. While the speedups are good, they are less than ideal, since PDB1 is a relatively small graph with limited work load.

Fig. 6 shows the effect of varying the compute nodes, and the effect of external neighbors on PDB3 and PDB4. For these datasets, we were unable to run the sequential version of DistGraph (with $p = 1$) for the lower support value, and therefore the speedups are reported with respect to the baseline number of compute nodes shown on the right y-axis.

We can see that the speedup is generally good, indicating relatively good load balancing across compute nodes. For instance, for PDB3 we get a speedup close to 3 for the lowest support, when comparing $p = 128$ with $p = 32$, with the ideal speedup being 4. Likewise, for PDB4 (which contains more than 1 billion edges), we get a

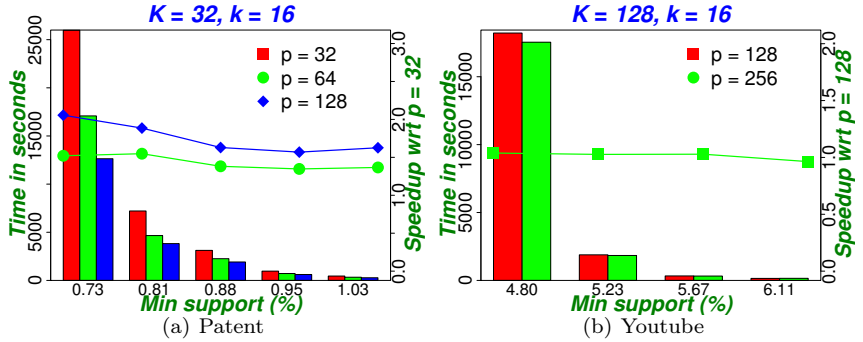


(a) PDB3 (no external neighbors) (b) PDB3 (with external neighbors)

(c) PDB4 (no external neighbors) (d) PDB4 (with external neighbors)

Fig. 6 PDB3 and PDB4: varying compute nodes; effect of external neighbors

speedup of 1.6 (ideal being 2), when we go from $p = 512$ to $p = 1024$ compute nodes. Finally, whereas we get similar speedups without and with external neighbors, the run times with external neighbor requests are about twice as high, at the lowest *minsup* value. This is mainly due to the cost of determining the exact global support, which is not required when there are no external neighbors.

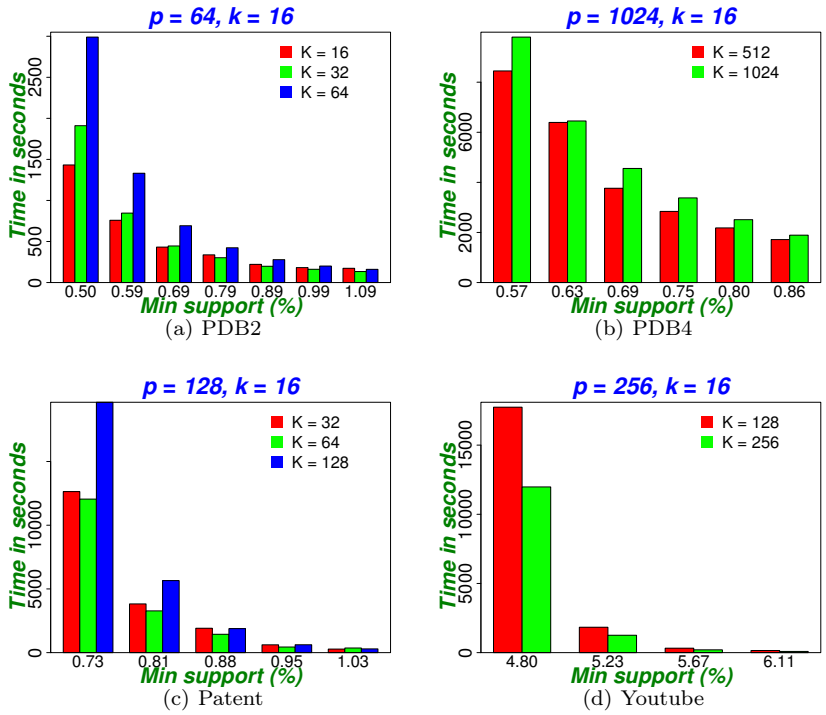


(a) Patent (b) Youtube

Fig. 7 Patent and Youtube: varying number of compute nodes

Fig. 7 shows results of varying compute nodes on the Patent and Youtube datasets. Even though these graphs are relatively large, they do not have many frequent pat-

terns. Therefore, the speedups are relatively low for Youtube, whereas Patent has more patterns and has better speedups (close to 2 with $p = 128$ wrt $p = 32$). On the other hand, the PDB datasets have many more "structural motifs" or frequent patterns and thus there is enough work to achieve good distributed performance.



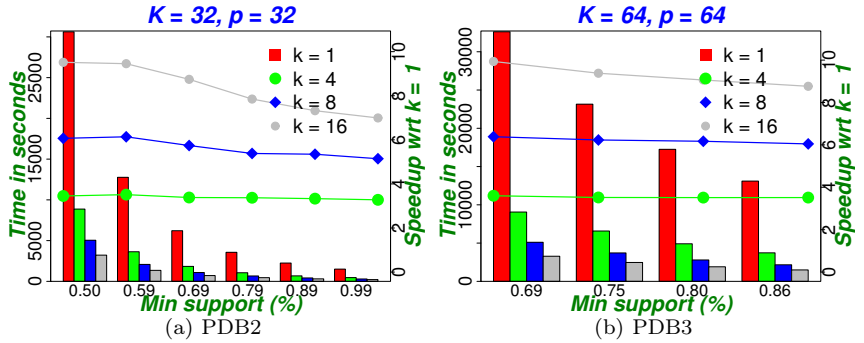
(a) PDB2 (b) PDB4

(c) Patent (d) Youtube

Fig. 8 Effect of varying partitions

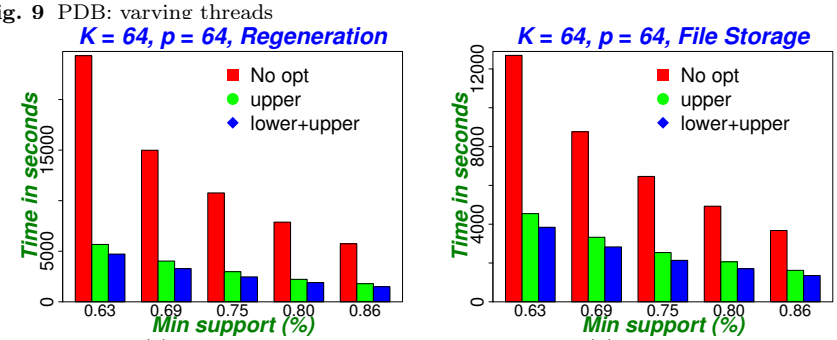
Varying Partitions: In Fig. 8 we show the effect of varying the number of partitions K. All results on PDB are *with* external neighbors. Consider the PDB2 dataset. With $K = 16$ partitions of the input graph, since we have $p = 64$ compute nodes, this means that there are 4 compute nodes in each group, which can work in a distributed manner to process the same partition. On the other hand, with $K = 64$ partitions of the input graph, each partition is processed by only one compute node. Although there are fewer external neighbors with $K = 64$ partitions compared to $K = 16$, the latter has the advantage of distributing the work load for a partition over 4 nodes. Across the datasets, the results show that for a fixed number of compute nodes, generally speaking, having fewer input partitions (with multiple compute nodes per group) results in better performance. However, Youtube is an exception. With fewer frequent patterns, having more compute nodes in a group does not help.

Varying Threads: Fig. 9 shows the effect of varying the number of threads k per compute node. The PDB datasets are *with* external neighbors. For both PDB2 and PDB3 we get speedup around 10 wrt $k = 1$ at the lowest support considered when using $k = 16$ threads (ideal speedup is 16).



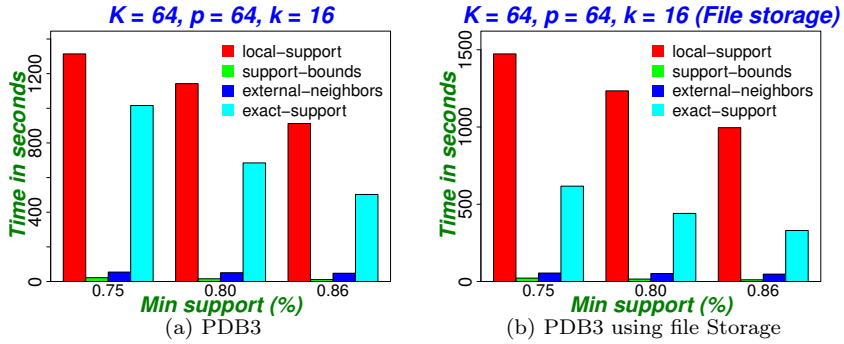
(a) PDB2 (b) PDB3

Fig. 9 PDB: varying threads



(a) Regeneration (b) File-based

Fig. 10 PDB3: Effect of Two-step Support Bounding



(a) PDB3 (b) PDB3 using file Storage

Fig. 11 Timing profile

Effect of support bounding optimization: In Fig. 10 we compare the effect of using the lower and upper support estimates to prune patterns. We also analyze the effect of regenerating the embeddings versus using the file-based approach to compute the global support. Here, "No opt" refers to using no optimization, i.e., considering all locally frequent patterns as the candidates for exact support computation. We can immediately see that the pruning based on support bounds is extremely effective in cutting down the communication cost, and thus the run times. The upper-bound effectively prunes patterns that cannot possibly be globally frequent. The lower-bound further determines the patterns that are truly globally frequent and prevents costly exact support computation.

For the remaining patterns, we have to determine the global support by either keeping the embeddings in memory, in a file on disk, or via regeneration. We show the time for the regeneration and file-based case in Fig. 10 (a) and (b), respv., on PDB3. As expected, we were not able to run the memory-based version for this graph. We can see that the file-based time is about 2 times faster than regeneration. However, with both lower and upper support bounding optimizations turned on, the times are more or less comparable. For instance, for the lowest support, the regeneration time is 4721.3s, whereas the file-based time is 3831.8s.

We also profiled the time spent in the various phases of DistGraph, as shown in Fig. 11(a) for PDB3 for both the regeneration and file-based approaches using $minsup = 0.75\%$. Both approaches show that the time for support-bound based pruning ($< 0.5\%$ of total time), and expanding the partitions via external neighbor requests ($< 2\%$ of total time) is negligible compared to the time it takes for computing the local support ($> 58\%$ of total time) and the time to determine the exact global support (41% and 28%, respectively, for regeneration and file-based approaches). Obviously in regeneration, we have to recompute the set of embeddings to determine actual support, which can be avoided by storing the embeddings on disk. Nevertheless, the regeneration approach is more general, since it can be applied to massive networks and in situations where the I/O cost for storing and retrieving potentially millions of embeddings can be a bottleneck.

Scalability: Finally, in Fig. 12 we show the scalability of our distributed mining algorithm. We show the performance on PDB3, PDB4, PDB5, and PDB6 (all *with* external neighbors). PDB6 has over a billion vertices and four billion edges, and is the largest graph to be used for frequent subgraph mining experiments. We use the following number of compute nodes/partitions ($p = K$): PDB3 (32), PDB4 (512), PDB5 (1024), PDB6 (2048). That is, to mine PDB6, we use $p = 2048$ compute nodes and $K = 2048$ partitions. We can see that DistGraph can scale to billion vertex graphs. For instance, for $minsup = 0.69$, the time on PDB6 (64x) is 7313s compared to 5491s for PDB3 (1x), which is only a factor of 1.33 higher (ideally the factor should be 1, since we are using 2048 = 64 x 32 compute nodes), which shows very good distributed performance, considering that PDB6 has many more partitions and thus many more external neighbors (64x in absolute terms) compared to PDB3.

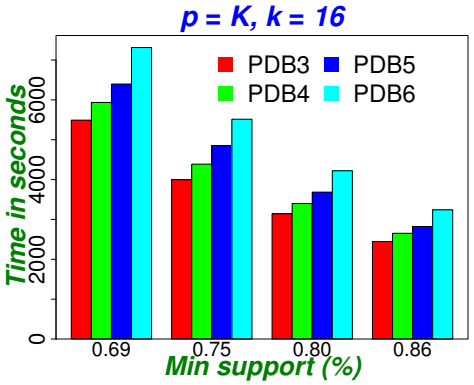


Fig. 12 DistGraph: Scalability on Billion Vertex Graph

7 Conclusions

We presented a novel distributed approach for mining frequent subgraph patterns from a single large graph. We partition the input graph to distribute the workload across the system. We minimize the amount of communication by sending compact mapping

counts instead of the pattern embeddings/mappings. Our solution is also very flexible, since it relies on collective communication primitives that are available on almost all distributed frameworks. Our hybrid approach also leverages both shared memory and distributed systems. Further, based on the resource availability, one can consider different number of partitions, compute nodes and threads, for scalability.

Our work is scalable, however, there are also some limitations that can be improved. One limitation of our current implementation of external neighbors expansion is that we do not prune any of the irrelevant vertices. While the increase is usually gradual, in pathological cases, the partitions may eventually become too large, especially if there are very large patterns. In such cases, we can *prune* from each partition any vertex that does not belong to the mappings for any frequent pattern. We can also use a fixed size memory-based cache for the local graph partition, swapping out least recently used vertices/edges to disk. Another approach is to move the computation instead of the data, i.e., migrate patterns to the partitions that own the external vertex, as opposed to requesting the neighbors. This is likely incur more communication overhead, but we plan to explore these alternatives in the future. Another limitation is dealing with hub nodes with very high degrees, which can lead to exponentially many embeddings. We outlined one approach where we duplicate the hubs nodes, creating a clique, and redistributing the neighbors among them, but more effective solutions are desired, especially since this is a problem for any graph mining algorithm. Finally, the techniques proposed in this paper can be used within the Arabesque [24] distributed graph mining framework for FSM over massive partitioned graphs. We plan to explore this in the future, and also plan to compare with a Spark implementation of DistGraph.

Acknowledgements This work was supported by NSF Award IIS-1302231. We thank Chris Carothers and Bulent Yener for several discussions on the practical and theoretical aspects of our distributed algorithm.

References

1. F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *IEEE Int'l Conference on Data Engineering*, 2013.
2. M. Bhuiyan and M. Al Hasan. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):608–620, March 2015.
3. B. Bringmann and S. Nijssen. What is frequent in a single graph? In *Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, 2008.
4. G. Buehrer, S. Parthasarathy, and Y.-K. Chen. Adaptive parallel graph mining for cmp architectures. In *IEEE Int'l Conference on Data Mining*, 2006.
5. M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7:517–528, 2014.
6. G. D. Fatta and M. R. Berthold. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):773–785, 2006.
7. S. Hill, B. Srichandan, and R. Sunderraman. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, 2012.
8. L. B. Holder and D. J. Cook. Discovery of inexact concepts from structural data. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):992–994, 1993.
9. J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *IEEE International Conference on Data Mining*, 2003.

10. A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery, LNCS Vol. 1910, Springer*, pages 13–23, 2000.
11. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, 1998.
12. R. Kessl, N. Talukder, P. Anchuri, and M. J. Zaki. Parallel graph mining with GPUs. *Proceedings of the BigMine Workshop (ACM SIGKDD), Journal of Machine Learning Research: Conference and Workshop Proceedings*, 36:1–16, 2014.
13. B. Kimelfeld and P. G. Kolaitis. The complexity of mining maximal frequent subgraphs. *ACM Transactions on Database Systems (TODS)*, 39(4):32, 2014.
14. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *IEEE International Conference on Data Mining*, 2001.
15. M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.
16. W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in mapreduce. In *IEEE International Conference on Data Engineering*, 2014.
17. Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In *Advanced Parallel Processing Technologies, LNCS Vol. 5737, Springer*, pages 341–355, 2009.
18. W. Lu, G. Chen, A. Tung, and F. Zhao. Efficiently extracting frequent subgraphs using mapreduce. In *IEEE International Conference on Big Data*, 2013.
19. T. Meinl, M. Wörlein, I. Fischer, and M. Philippsen. Mining molecular datasets on symmetric multiprocessor systems. In *IEEE International Conference on Systems, Man and Cybernetics, Vol. 2*, 2006.
20. S. Reinhardt and G. Karypis. A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph. In *IEEE International Parallel and Distributed Processing Symposium*, 2007.
21. S. Shahrivari and S. Jalili. Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing*, 97(11):1101–1120, 2015.
22. Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *ACM SIGMOD International Conference on Management of Data*, 2014.
23. Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of VLDB Endowment*, 5(9):788–799, 2012.
24. C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph pattern mining. In *25th ACM Symposium on Operating Systems Principles*, 2015.
25. D. Ucar, S. Asur, U. Catalyurek, and S. Parthasarathy. Improving functional modularity in protein-protein interactions graphs using hub-induced subgraphs. In *Knowledge Discovery in Databases: PKDD 2006*, pages 371–382. Springer, 2006.
26. B. Wu and Y. Bai. An efficient distributed subgraph mining algorithm in extreme large graphs. In *International Conference on Artificial Intelligence and Computational Intelligence: Part I*, 2010.
27. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *IEEE International Conference on Data Mining*, 2002.
28. G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 344–353. ACM, 2004.