Middle-East Journal of Scientific Research 24 (Techniques and Algorithms in Emerging Technologies): 524-531, 2016
ISSN 1990-9233;
© IDOSI Publications, 2016
DOI: 10.5829/idosi.mejsr.2016.24.TAET23634

An Efficient Parallel Iterative Mapreduce Based Frequent Subgraph Mining Algorithm

[1]A. Praveena, [2]B. Anitha and [2]R. Rohini

[1]PG Scholar, Department of CSE,
Vivekanandha College of Engineering for Women, Tamilnadu, India
[2]Assistant Professor, Department of CSE,
Vivekanandha College of Engineering for Women, Tamilnadu, India

Abstract: Mining frequent sub-graphs has attracted a great deal of attention in many areas, such as bioinformatics, the web data mining and social networks. There are many promising main memory-based techniques available in this area, but they lack scalability as the main memory is a bottleneck. Taking the massive data into consideration, traditional database systems like relational databases and object databases fail miserably with respect to on efficiency as frequent subgraph mining is computationally intensive. The proposed algorithm is parallel iterative Map-Reduce based frequent subgraph mining (PSGM), which in this work a frequent subgraph mining algorithm called PSGM which uses iterative Map-Reduce based framework. PSGM is complete as it returns all the frequent subgraphs for a given user-defined support and it is efficient as it applies all the optimizations that the latest FSM algorithms adopt. The experiments with real life and large synthetic datasets validate the effectiveness of PSGM for mining frequent sub-graphs from large graph datasets.

Key words: Hadoop · Map/Reduce · Frequent Mining · Parallel map/reduce · Graph mining

INTRODUCTION

Mining frequent patterns have motivated many researchers since the very first research on finding the association rules in itemset. In many domains graphs are prevalent such as bioinformatics, Semantic Web, cheminformatics and social networks. In graph mining research, frequent subgraph mining is a very well-studied area because of its wide range of applications in the above areas. Frequent patterns can help different functions and relations to understand. For example, in a protein-protein interaction network (PPI), a frequent pattern could uncover unknown functions of a protein and in a social network a friend clique is considered to be a frequent pattern. There are two different aspects of mining frequent subgraphs. 1) Using a single large graph. 2) Using a set of graphs. The difference in the single graph setting and transaction setting is count. The frequency of a substructure is determined by the number of graph transactions containing the pattern is said to be transaction setting and the frequency of a substructure is determined by the number of times the pattern appears in the whole graph is said to be single graph setting. There are a few implementations for finding the frequent patterns in a single graph using Map-Reduce framework and one for finding sub-graphs in transactions graphs in a grid environment. All major frequent subgraph mining algorithms are on the assumption that the graph data fits well in memory. Memory-based algorithms do fairly well on small datasets but as the data size increases, memory becomes a bottleneck. To overcome the problems, a few database approaches was proposed. The issue with the database model is that as the dataset size grows, computation time rises drastically.

The Map-Reduce framework of distributed computing is one of the most successful. It adopts a data-centric approach to distributed computing with the ideology of “moving computation to data”; to improve the IO performance while handling massive data it uses a distributed file system that is particularly optimized.

Corresponding Author: A. Praveena, PG Scholar, Department of CSE,
Vivekanandha College of Engineering for Women, Tamilnadu, India.

The main reason for the framework is to gain attention to higher level of abstraction that it provides, it keeps many system level details hidden from the programmers and allows them to concentrate more problems specific on computational logic analyses are limited to estimating global statistics (such as diameter), spectral analysis, or vertex-centrality analysis. Efforts for mining sub-structure are not that common, except a few works for counting triangles. Specifically, frequent subgraph mining on Map-Reduce has received the least attention. Given the growth of applications of frequent subgraph mining in various disciplines including social networks, bioinformatics, cheminformatics and semantic web, a scalable method for frequent subgraph mining on Map-Reduce is of high demand. Solving the task of frequent subgraph mining (FSM) on a distributed platform like Map-Reduce is challenging for various reasons. First, an FSM method computes the support of a candidate subgraph pattern over the entire set of input graph dataset. In a distributed platform, the input graphs are partitioned over different worker nodes, the local support of a sub-graph at a worker node is not much useful for deciding whether the given subgraph is frequent or not. Also, local support of a subgraph in various nodes cannot be aggregated in a global data structure because Map-Reduce programming model does not provide any built-in mechanism for communicating. Also, the support computation cannot be delayed, as following Apriori principle; future candidate frequent patterns [1] can be generated only from a frequent pattern. A parallel method to extract the significant patterns from a set of labeled graphs using Map-Reduce is proposed. It works for both directed and undirected graphs. Given a graph dataset D = $\{G_1,G_2,\ldots..,G_n\}$, the support of a subgraph S(g) indicates the total number of times the subgraph g appears in the whole dataset D. A subgraph g is called frequent if S(g) at least satisfies the user provided support. It constructs and retains all the patterns that have a non-zero support in the map phase of the mining and then in the reduce phase, in a different computing nodes it decides whether a pattern is frequent by aggregating its support is used to ensure completeness. To overcome the dependency of a mining process, PSGM runs in an iterative fashion, where the output from the reducers of iteration i-1 can be used as an input for the mappers in the iteration i. The mappers of iteration i generate candidate subgraphs of size i (number of edges) and also compute the local support of the candidate pattern. The reducers of iteration i then find the true frequent sub-graphs (of size i) by aggregating their local supports.

Related Work: There exist many algorithms for solving the in-memory version of frequent subgraph mining task most notable among them are AGM [1], FSG [2], gSpan [3], Gaston [4] and DMTL [4]. These methods assume that the dataset is small and the mining task finishes in a reasonable amount of time using an in-memory method. To consider the large datasets, a few traditional database based graph mining algorithms, such as DB-Subdue [5] and DB-FSG [6] and OOFSG [7] are also proposed.

For large-scale graph mining tasks, researchers considered shared memory parallel algorithms for frequent subgraph mining. Cook *et al*. presented a parallel version of their frequent subgraph mining algorithm Subdue [8]. Wang *et al.* developed a parallel toolkit [9] for their Motif-Miner [10] algorithm. Meinl *et al*. created a software named Parmol [11] which includes parallel implementation of Mofa [12], gSpan [17], FFSG [13] and Gaston. Par SeMis [14] is another such tool provides the parallel implementation of a gSpan algorithm. The problem caused by the size of input graphs is scalability; there are a couple of notable works, PartMiner [15] and PartGraphMining [16], which can be based on the idea of partitioning the graph data.

There also exists a work [17] on adaptive parallel graph mining for CMP Architectures. Map-Reduce framework has been used to mine frequent patterns where the transactions in the input database are simpler combinatorial objects such as a set [18], [19], [20], [21], or a sequence [22]. In [23], the authors consider frequent subgraph mining on Map-Reduce; however, it is inefficient due to various shortcomings. The most notable method is that do not adopt any mechanism to avoid generating duplicate patterns. Due to the size of the candidate sub-graph space the size gets increased; furthermore, the output set contains the duplicate copy of the same graph patterns that are hard to unify as the user has to provide a sub-graph isomorphism routine for this method. Another problem with the method is that it requires the number of MapReduce iterations can be specified by the user. Authors did not mention how the total iteration counts are determined so that the algorithm can find all frequent patterns for a given support. One feasible way to set the iteration count to be the edge count of the largest transaction, but that will be an overkill of the resources. FSM-H does not suffer any limitations from the above method. During the revision phase of this journal, it became aware of another work [23] of frequent subgraph mining on MapReduce. It is a non-iterative method which runs on each partition of the graph database.

Proposed Work: Map-Reduce is a programming model that enables distributed computation over massive data. The model provides two abstract functions: map and reduce. Map corresponds to the "map" function and Reduce corresponds to the "fold" function in functional programming. A worker node in Map-Reduce is called a mapper or a reducer. A mapper and reducer takes a collection of (key, value) pairs and applies the map function on each of the pairs to generate an arbitrary number of (key, value) pairs as intermediate output. The reducer aggregates all the values that have the same key in a sorted list and applies the reduce function on that list. The output writes to the output file. The files (input and output) of Map-Reduce are managed by a distributed file system. Hadoop is an open-source implementation of Map-Reduce programming model written in Java language.

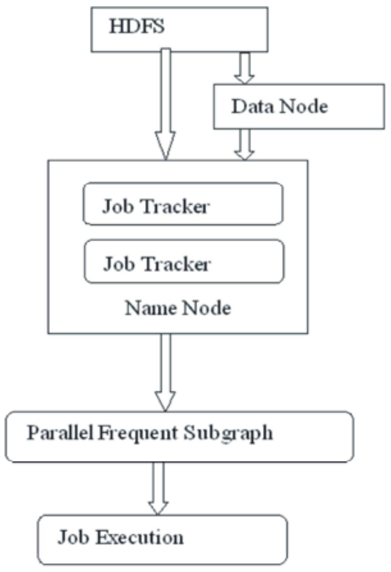

Fig. 1: Parallel Frequent Subgraph Mining Architecture

Hadoop Distributed File System: HDFS is a technology that it provides data distribution, replication and automatic recovery in a user-space file system that is relatively easy to configure and conceptually, easy to understand. However, the utility comes to light when map/reduce jobs can be executed on data stored in HDFS. As the name implies, map/reduce jobs can be principally comprised of two steps: the map step and the reduce step. The overall workflow generally looks something like this: The idea underpinnings map/reduce bringing compute to the data instead of the opposite should be like a very simple solution to the I/O bottleneck in the traditional parallelism. However, implementing a framework where a single large file is transparently diced up and distributed across multiple physical computing elements (all while appearing to remain a single file to the user) is not trivial.

Hadoop, the most widely used map/reduce framework, accomplishes using HDFS, the Hadoop Distributed File System. HDFS is fundamental to Hadoop because it provides the data chunking and distribution across computing elements necessary for map/reduce applications to be efficient. Since it talking about an actual map/reduce implementation and not an abstract concept, let's refer to the abstract compute elements now as compute nodes.

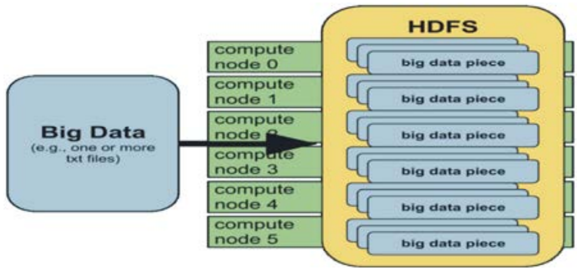

Fig. 2: Hadoop distributed file system

When a file is copied into HDFS as depicted above, that file is transparently split into "chunks" and replicated three times for reliability. Each of these chunks is distributed to compute nodes in the Hadoop cluster so that a given chunk exists on three different nodes. Although physically chunked up and distributed in triplicate, all of the interactions with the file on HDFS still appeared as the same single file where it is copied into HDFS initially. Thus, HDFS handles the entire computing nodes, distributing and recombining the data.

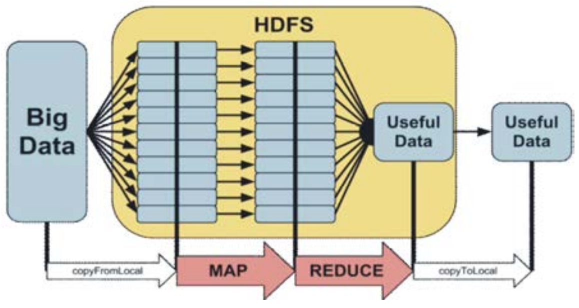

Fig. 3: Map/Reduce Jobs

Map/reduce Programming Model: The purpose to deal with the massive data distributed and stored in the server cluster, such as crawled documents, web request logs, etc., the programmer writes program in order to get the results from different data such as inverted indices, web document and different views and so on. Map/Reduce programming model, by map and reduce function realize the Mapper and Reducer interfaces. They form the core of task.

Mapper Phase: Map function requires the user to handle the input of a pair of < key, value> and produces a group of intermediate key and value pairs. <key,value> consists of two parts, key stands for the "group number " of the value. Value stands for the data related to the task. It combines the intermediate values with same key using sorting and then sends the output to the reduce function. Map algorithm process is described as follows:

- Step1. MapReduce framework produces a map task for each input and each graph input is generated by the Name Node job. Each <Key,Value> corresponds to a map task.
- Step2. Execute Map task, process the input <key,value> to produce a new <key,value>. This process is called "divide into groups". That is, make the same values correspond to the same key words. A given input value pair can be mapped into 0 or more output pairs. Output key value pairs that do not required the same type of the input key value pairs.
- Step3. Mapper's output is sorted using the corresponding key value pairs and to be allocated to each Reducer.The total number of blocks and the number of job reduce tasks is same.

Reduce Phase: Reduce function is also provided by the user and it always process after the map function which handles the intermediate key pairs. Reduce function mergers the values to get a small set of values the process is called "merge ". Reducer makes a group of intermediate values set that associated with the same key. In MapReduce framework <key,value> is the communication interface for the programmer in Map Reduce model.

For example,<key,value> can be seen as a "letter", key is the letter’s posting address, value is the letter’s content. With the same address letters will be delivered to the same place <key,value>, MapReduce framework can automatically and accurately cluster the values with the same key together. Reducer algorithm process is described as follows:

- Step1. Shuffle. The output from the sorted Mapper is used to give the input to the reducer. In this stage, MapReduce will assign related block for each Reducer.
- Step 2. Sorting. In this stage, the input of reducer is grouped or sorted according to the key (because the output of different mapper may have the same key). The two stages of Shuffle and Sort are synchronized;
- Step 3. Secondary Sorting. If the key grouping rule in the intermediate process is different from its rule before reduce. It can provide a new method is defined i.e., Comparator. The comparator is used to group the intermediate keys for the second time.

Map tasks and Reduce task is a whole process, it cannot be separated. It should be used together in the program. It calls a MapReduce the process as an MR process. In an MR process, Map tasks run are improved and Reduce tasks run are improved, Map and Reduce tasks can run serially. An MR process and the next MR process can run in a sequential manner, synchronization between these operations is guaranteed by the MR system. Let, G = $\{G_1, G_2, \ldots, G_n\}$ be a graph database, where each $G_i \in G$, $\forall i = \{1\ldots n\}$ represents a labeled, undirected and connected graph. For a graph g, its size is defined as the number of edges it contains. Now, t(g) = $\{G_i: g \subseteq G_i \in G\}$, $\forall i = \{1\ldots n\}$, is the *support-set* of the graph g (here the subset symbol denotes a subgraph relation). Thus, t(g) contains all the graphs in G that has a subgraph isomorphic to g. The cardinality of the *support-set* is called the *support* of g. g is called frequent if support = min, where min is predefined/user-specified *minimum support (minsup)* threshold. The set of frequent patterns are represented by F. Based on the size (number of edges) of a frequent pattern, it can partition F into a several disjoint sets, Fi such that each of the Fi contains frequent patterns of size i only.

Map/reduce Subgraph: Parallel frequent subgraph mining (FSM) task, partitions the graph dataset G = $\{G_i\}$ i=1...n into k disjoint partitions, such that each partition contains roughly equal number of graphs; thus it mainly distributes the support counting subroutine of a frequent pattern mining algorithm. Conceptually, each node of PSGM runs an independent FSM task over a graph dataset which is 1/k th of the size of |G|. The FSM algorithm is that PSGM implements are an adaptation of the parallel algorithm.

Candidate Generation: Given a frequent pattern of size k, it adjoins a frequent edge F1 with c to obtain a candidate pattern d of size k + 1. If d can add the additional vertex then the added edge is called a forward edge, otherwise it is called a back edge; it connects two existing vertices of c. Add the vertex of a forward edge is given an integer id, which is the largest integer id following the ids of the existing vertices of c; thus the vertex-id stands for the order in which the forward edges are adjoined while building a candidate pattern. In graph mining, c is called the parent of d and d is a child of c, parent-child relationship it can arrange the set of candidate patterns of a mining task in a candidate generation tree.

Duplicate Graph Checking: From multiple generation paths a candidate pattern can be generated, but only one such path is explored during the candidate generation step and the remaining unwanted paths are identified and subsequently ignored. A graph mining algorithm needs to solve the graph isomorphism task, as the duplicate copies of a candidate patterns are isomorphic to each other is used to identify invalid candidate generation paths. A well-known method for identifying graph common sharing is to use canonical coding scheme, which serializes the edges of a graph using a prescribed order and generates a string such that all isomorphic graphs will generate the same string. There are many different canonical coding schemes; one of the main methods is min-dfs-code. According to the scheme, the generation path of a pattern in which the insertion order of the edges matches with the edge ordering in the min-dfs-code is considered as the valid generation path and the remaining generation paths are considered as duplicate and hence ignored.

Support Value: Support value of a graph pattern g is important because it is used to determine whether g is frequent or not. To count g's support it need to find the database graphs in which g is embedded. This mechanism requires solving a sub-graph isomorphism problem, which is NP-complete. One feasible way to compute the support of a pattern without explicitly performing the sub-graph isomorphism test across all database graphs is to maintain the occurrence-list (OL) of a pattern; such a list stores the embedding of the pattern (in terms of vertex id) in each of the database graphs where the pattern exists. When a pattern is extended to obtain a child pattern in the candidate generation step, the embedding of the child pattern must include the embedding of the parent pattern, thus the occurrence-list of the child pattern can be generated efficiently from the occurrence list of its parent. Then the support of a child pattern can be obtained trivially from its occurrence-list.

Frequent Subgraph Mining: The proposed parallel data parallel, preparation phase and mining phase. In data partition phase PSGM creates the partitions of input data the infrequent edges from the input graphs are removed. Preparation and mining phase performs the actual mining task.

Data Parallel: The parallel input process splits the input graph dataset (G) into many partitions. One straightforward partition scheme is to distribute the graphs so that each partition contains the same number of graphs from G. It works well for most of the datasets. However, for datasets where the size (edge count) of the graphs in a dataset varies substantially, PSGM offers another splitting option in which the total number of edges aggregated over the graphs in a partition, are close to each other. In experiment section, it shows that the latter partition scheme it improves the load-balancing factor of a Map-Reduce job for improving the performance. For PGSM, an important tuning parameter is said to be the number of partition. In experiment section, it also show that for achieve the optimal performance, the number of partitions for PSGM should be substantially larger than the number of partition in a typical Map-Reduce task.

Data Allocation: The mappers in this phase prepare some partition specific data structures such that for each partition there is a distinct copy of these data structures. The first of such data structure is called edge-extension-map, which is used for any candidate generation that happens over the entire mining session. They are static for a partition in the sense that they are same for all patterns generated from a partition. The mappers in the preparation phase also start the mining task by emitting the frequent single edge pattern as the key-value pair. It stores the possible extension from a vertex considering the edges that exists in the graphs of a partition. Note that, all the single edges can exist in any graph of any partition is frequent since the partition phase has filtered out all the infrequent edges. As mentioned earlier, the key of a pattern is its min-dfs-code and the value is the pattern object. Each pattern object has four essential attributes: (a) Occurrence List (OL) that stores the embedding of the pattern in each graph in the partition, (b) Right-Most-Path (c) VSET that stores the embedding of the Right Most Path in each graph in the partition and (d) support value. Mappers in the preparation phase compute the min-dfs-code and create the pattern object for each single-edge patterns.

Mining Algorithm: In this phase, mining process discovers all possible frequent subgraphs through iteration. Data allocation phase populates all frequent subgraphs of size one and writes it in the distributed file system. Iterative job starts by reading these from HDFS. The map function of mining phase reconstructs all the static data structures that are required to generate candidate patterns from the current pattern.

In the preparation phase each of the mappers of an ongoing iteration is responsible for performing the mining task over a particular chunk of the data written in HDFS. The mappers reconstruct the pattern object of size k along with the static data structures and generate the candidates from the current pattern. Use Java to write the baseline mining algorithm as well as the map and the reduce function in the preparation and the mining phase. It overrides the default input reader and writes a custom input reader for the preparation phase. To improve the execution time of MapReduce job, it compresses the data while writing them in HDFS. We used global counter provided by Hadoop to track the stopping point of the iterative mining.

Algorithm: Data Allocation
Allocation data(D):

```
1.  create a data directory in distributed file system
2.  partition = create partition()
3.  while data available in D
4.  Allocate each split in different mapper
5.  write partition to file allocation in data directory
```

Mining Algorithm

```
    key = offset
    value = location of partition file in data directory
    Mapper Data Allocation( key, value):
    1. Generate Level one OL(value)
    2. P = get single length patterns()
    3. Generate Level one MAP(value)
    4. forall Pi in P:
    5. intermediate value = serialize(Pi)
    6. intermediate key = min-dfs-code(Pi )
    7. emit(intermediate key,intermediate value)
    key = min-dfs-code
 values = List of Byte-stream of a pattern object in all
partitions Reducer preparation(Text key, BytesWritable h
values i):
    1. for all value in values:
    2. write to file(key,value)
```

RESULTS AND DISCUSSION

The parallel frequent subgraph mining algorithm task is on the large graph datasets. As input, it used real-world graph datasets which are taken from an online and also synthetic datasets using a tool called Graph-gen. The number of graphs in these datasets range from 100K to 1000K and each graph contains on average 25-30 edges. It conducts all experiments in a 10-nodes Hadoop cluster, where one of the node is set to be a master nodes and the remaining two nodes are set to serve as data node. Each machine possesses a 3.1 GHz quad -core Intel processor with 16GB memory and 1 TB of storage.

Table 1: Compare runtime vs Support value for YEAST Dataset

	Support value					
Algorithm	30	50	70	80	90	100
FSM-H	84	75	65	56	45	42
PSGM	76	63	57	44	32	20

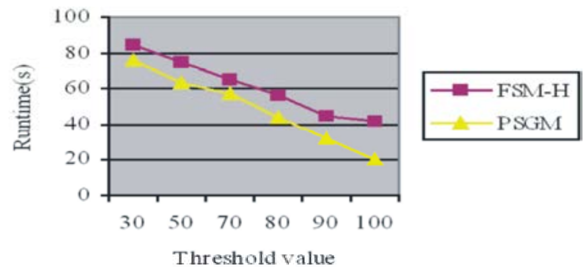

Fig. 3: Performance comparison of the traditional FSM-H and PSGM.

Table 1: Compare no of reducer vs Support value for YEAST Dataset

		Support value					
Algorithm		2	4	6	8	10	12
FSM-H	120	104	98	84	75	65	
PSGM	97	83	79	65	57	49	

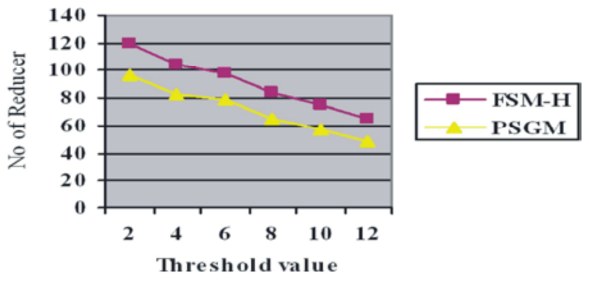

Fig. 4: Performance comparison of no of reducer the traditional FSM-H and PSGM.

CONCLUSION AND FUTURE WORK

The proposed system uses parallel iterative MapReduce -based parallel frequent subgraph mining algorithm, called PSGM. It shows the performance of PSGM over real life and large synthetic datasets for various system and input configurations. It also compares the execution time of PSGM with an existing method, which shows that PSGM is significantly better than the existing method. Although it shows a novel approach to subgraph mining that has promising results, due to limited facilities, further work will be needed to fully understand the scalability of the method.

REFERENCES

1. Nijssen, S. and J. Kok, 2004. "A quickstart in frequent structure mining can make a difference," in Proc. of ACM SIGKDD, 2004.
2. Chakravarthy, S., R. Beera and R. Balachandran, 2004. "Db-subdue: Database approach to graph mining," in Advances in Knowledge Discovery and Data Mining.
3. Chakravarthy, S. and S. Pradhan, 0000. "Db-fsg: An sql-based approach for frequent subgraph mining," in Proceedings of the 19th international conference on Database and Expert Systems Applications, ser. DEXA '08.
4. Cook, D.J., L.B. Holder, G. Galal and R. Maglothin, 2001. "Approaches to parallel graph-based knowledge discovery," Journal of Parallel and Distributed Computing, pp: 61.
5. Wang, C. and S. Parthasarathy, 2004. "Parallel algorithms for mining frequent structural motifs in scientific data," in Proc. of the 18th annual intl. conference on Supercomputing.
6. Parthasarathy, S. and M. Coatney, 2002. "Efficient discovery of common substructures in macromolecules," in In Proc. of IEEE International Conference on Data Mining.
7. Meinl, T., M. Worlein, O. Urzova, I. Fischer and M. Philippsen, 2006. "The parmol package for frequent subgraph mining." ECEASST.
8. Borgelt, C. and M. Berthold, 2002. "Mining molecular fragments: finding relevant substructures of molecules," in Proc.of IEEE International Conference on Data Mining.
9. Huan, J., W. Wang and J. Prins, 2003. "Efficient mining of frequent subgraphs in the presence of isomorphism," in ICDM.
10. Philippsen, M., M. Worlein, A. Dreweke and T. Werth, 2011. "Parsemis -the parallel and sequential mining suite," [Online]. Available: https://www2.cs.fau.de/EN/research/ParSeMiS/index.html.
11. Wang, J., W. Hsu, M.L. Lee and C. Sheng, 2006. "A partition-based approach to graph mining," in Proc. of the 22nd International Conference on Data.
12. Nguyen, S.N., M.E. Orlowska and X. Li, 2008. "Graph mining based on a data partitioning approach," in Nineteenth Australasian Database Conference (ADC 2008).
13. Chen, G.P., Y.B. Yang and Y. Zhang, 2012. "Mapreduce-based balanced mining for closed frequent itemset," in IEEE 19th International Conference on Web Service, 2012, pp: 652-653.
14. Wang, S.Q., Y.B. Yang, Y. Gao, G.P. Chen and Y. Zhang, 2012. "Mapreducebased closed frequent itemset mining with efficient redundancy filtering," 2012 IEEE 12th International Conference on Data Mining Workshops.
15. Zhou, L., Z. Zhong, J. Chang, J. Li, J. Huang and S. Feng, 2010. "Balanced parallel fp-growth with mapreduce," in Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on.
16. Li, H., Y. Wang, D. Zhang, M. Zhang and E.Y. Chang, 2008. "Pfp: parallel fp-growth for query recommendation," in Proceedings of the 2008 ACM conference on Recommender systems.
17. Jeong, B.S., H.J. Choi, M.A. Hossain, M.M. Rashid and M.R. Karim, 2012. "A MapReduce Framework for Mining Maximal Contiguous Frequent Patterns in Large DNA Sequence Datasets," IETE Tech. Review, 29: 162-168.
18. Hill, S., B. Srichandan and R. Sunderraman, 2012. "An iterative mapreduce approach to frequent subgraph mining in biological datasets," in Proc. of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine.
19. Wu, B. and Y. Bai, 2010. "An efficient distributed subgraph mining algorithm in extreme large graphs," in Proceedings of the 2010 international conference on Artificial intelligence and computational intelligence: Part I.
20. Liu, Y., X. Jiang, H. Chen, J. Ma and X. Zhang, 2009. "Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network," in Proc. of the 8th Intl. Symp. on Advanced Parallel Processing Technologies.
21. Di Fatta, G. and M. Berthold, 2006. "Dynamic load balancing for the distributed mining of molecular structures," Parallel and Distributed Systems, IEEE Transactions on.
22. Lin, J. and C. Dyer, 2010. Data-Intensive Text Processing with MapReduce.
23. Cheng, J., Y. Ke, W. Ng and A. Lu, 2007. "Fg-index: towards verification-free query processing on graph databases," in SIGMOD, pp: 857-872.
24. Bhuvaneswari, M., R. Rohini and B. Preetha, 2013. "A Survey on privacy preserving public auditing for secure data storage", International Journal of Engineering Research and Technology, Nov 2013.

25. Praveena, A., B. Anitha and R. Rohini, 2015. "Study of Iterative Mapreduce Techniques on Frequent Subgraph Mining", International Journal of Advanced Research in Computer Science and Software Engineering, Volume 5, November 2015, ISSN: 2277 128X.

26. Jayakumari, A., R. Rohini and B. Anitha, 2015. "Study of Privacy Preserving Frequent Itemset Mining Via Smart Splitting", International Journal of Advanced Research in Computer Science and Software Engineering,Volume 5, November 2015, ISSN: 2277 128X.