Big Data Frequent Pattern Mining

David C. Anastasiu and Jeremy Iverson and Shaden Smith and George Karypis
Department of Computer Science and Engineering
University of Minnesota, Twin Cities, MN 55455, U.S.A.
{dragos, jiverson, shaden, karypis}@cs.umn.edu

Abstract

Frequent pattern mining is an essential data mining task, with a goal of discovering knowledge in the form of repeated patterns. Many efficient pattern mining algorithms have been discovered in the last two decades, yet most do not scale to the type of data we are presented with today, the so-called "Big Data". Scalable parallel algorithms hold the key to solving the problem in this context. In this chapter, we review recent advances in parallel frequent pattern mining, analyzing them through the Big Data lens. We identify three areas as challenges to designing parallel frequent pattern mining algorithms: memory scalability, work partitioning, and load balancing. With these challenges as a frame of reference, we extract and describe key algorithmic design patterns from the wealth of research conducted in this domain.

Introduction

As an essential data mining task, frequent pattern mining has applications ranging from intrusion detection and market basket analysis, to credit card fraud prevention and drug discovery. Many efficient pattern mining algorithms have been discovered in the last two decades, yet most do not scale to the type of data we are presented with today, the so-called "Big Data". Web log data from social media sites such as Twitter produce over one hundred terabytes of raw data daily [32]. Giants such as Walmart register billions of yearly transactions [1]. Today's high-throughput gene sequencing platforms are capable of generating terabytes of data in a single experiment [16]. Tools are needed that can effectively mine frequent patterns from these massive data in a timely manner.

Some of today's frequent pattern mining source data may not fit on a single machine's hard drive, let alone in its volatile memory. The exponential nature of the solution search space compounds this problem. Scalable parallel algorithms hold the key to addressing pattern mining in the context of Big Data. In this chapter, we review recent advances in solving the frequent pattern mining problem in parallel. We start by presenting an overview of the frequent pattern mining problem and its specializations in Section 1. In Section 2, we examine advantages of and challenges encountered when parallelizing algorithms, given today's distributed and shared memory systems, centering our discussion in the frequent pattern mining context. We survey existing serial and parallel pattern mining methods in Sections 3 – 5. Finally, Section 6 draws some conclusions about the state-of-the-art and further opportunities in the field.

1 Frequent Pattern Mining: Overview

Since the well-known itemset model was introduced by Agrawal and Srikant [4] in 1994, numerous papers have been published proposing efficient solutions to the problem of discovering frequent patterns in databases. Most follow two well known paradigms, which we briefly describe in this section, after first introducing notation and concepts used throughout the paper.

1.1 Preliminaries

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items. An **itemset** C is a subset of I. We denote by $|C|$ its *length* or *size*, i.e. the number of items in C. Given a list of transactions $\mathcal{T}$, where each transaction $T \in \mathcal{T}$ is an itemset, $|\mathcal{T}|$ denotes the total number of transactions. Transactions are generally identified by a transaction id (*tid*). The **support** of C is the proportion of transactions in $\mathcal{T}$ that contain C, i.e., $\phi(C) = |\{T|T \in \mathcal{T}, C \subseteq T\}|/|\mathcal{T}|$. The *support count*, or *frequency* of C is the number of transactions in $\mathcal{T}$ that contain C. An itemset is said to be a **frequent itemset** if it has a support greater than some user defined minimum support threshold, σ.

The itemset model was extended to handle sequences by Srikant and Agrawal [54]. A **sequence** is defined as an ordered list of itemsets, $s = \langle C_1, C_2, \ldots, C_l \rangle$, where $C_j \subseteq I, 1 \leq j \leq l$. A sequence database $\mathcal{D}$ is a list of $|\mathcal{D}|$ sequences, in which each sequence may be associated with a customer id and elements in the sequence may have an assigned timestamp. Without loss of generality, we assume a lexicographic order of items $i \in C,\ C \subseteq I$. We assume sequence elements are ordered in non-decreasing order based on their timestamps. A sequence $s' = \langle C'_1, C'_2, \ldots, C'_m \rangle$, $m \leq l$ is a **sub-sequence** of s if there exist integers $i_1, i_2, \ldots, i_m$ s.t. $1 \leq i_1 \leq i_2 \leq \ldots \leq i_m \leq l$ and $C'_j \subseteq C_{i_j},\ j = 1, 2, \ldots, m$. In words, itemsets in s' are subsets of those in s and follow the same list order as in s. If s' is a sub-sequence of s, we write that $s' \subseteq s$ and say that s *contains* s' and s is a **super-sequence** of s'. Similar to the itemset support, the **support** of s is defined as the proportion of sequences in $\mathcal{D}$ that contain s, i.e., $\phi(s) = |\{s' | s' \in \mathcal{D}, s \subseteq s'\}| / |\mathcal{D}|$. A sequence is said to be a **frequent sequence** if it has a support greater than σ.

A similar model extension has been proposed for mining structures, or graphs/networks. We are given a set of graphs $\mathcal{G}$ of size $|\mathcal{G}|$. Graphs in $\mathcal{G}$ typically have labelled edges and vertices, though this is not required. $V(G)$ and $E(G)$ represent the vertex and edge sets of a graph G, respectively. The graph $G = (V(G), E(G))$ is said to be a **subgraph** of another graph $H = (V(H), E(H))$ if there is a bijection from $E(G)$ to a subset of $E(H)$. The relation is noted as $G \subseteq H$. The **support** of G is the proportion of graphs in $\mathcal{G}$ that have G as a subgraph, i.e., $\phi(G) = |\{H | H \in \mathcal{G}, G \subseteq H\}| / |\mathcal{G}|$. A graph is said to be a **frequent graph** if it has a support greater than σ.

The problem of frequent pattern mining (FPM) is formally defined as follows. Its specialization for the frequent itemset mining (FIM), frequent sequence mining (FSM), and frequent graph mining (FGM) is straight-forward.

Definition 1 *Given a pattern container $\mathcal{P}$ and a user-specified parameter σ $(0 \leq \sigma \leq 1)$, find all sub-patterns each of which is supported by at least $\lceil \sigma |\mathcal{P}| \rceil$ patterns in $\mathcal{P}$.*

At times, we may wish to restrict the search to only **maximal** or **closed** patterns. A maximal pattern m is not a sub-pattern of any other frequent pattern in the database, whereas a closed pattern c has no proper super-pattern in the database with the same support.

A number of variations of the frequent sequence and frequent graph problems have been proposed. In some domains, the elements in a sequence are symbols from an alphabet $\mathcal{A}$, e.g., $\mathcal{A} = \{A, C, G, T\}$ and $s = \langle TGGTGAGT \rangle$. We call these sequences *symbol sequences*. The symbol sequence model is equivalent to the general itemset sequence model where $|C| = 1$ for all $C \in s,\ s \in \mathcal{D}$. Another interesting problem, *sequence motif mining*, looks to find frequent sub-sequences within one (or a few) very long sequences. In this case, the support threshold is given as a support count, the minimum number of occurrences of the sub-sequence, rather than a value $0 \leq \sigma \leq 1$, and additional constraints may be specified, such as minimum/maximum sub-sequence length. A similar problem is defined for graphs, unfortunately also called *frequent graph mining* in the literature, where the support of G is the number of *edge-disjoint* subgraphs in a large graph $\mathcal{G}$ that are isomorphic to G. Two subgraphs are edge-disjoint if they do not share any edges. We call each appearance of G in $\mathcal{G}$ an *embedding*. Two graphs G and H are isomorphic if there exists a bijection between their vertex sets, $f\colon V(G) \to V(H)$, s.t. any two vertices $u, v \in V(G)$ are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.

1.2 Basic Mining Methodologies

Many sophisticated frequent itemset mining methods have been developed over the years. Two core methodologies emerge from these methods for reducing computational cost. The first aims to prune the candidate frequent itemset search space, while the second focuses on reducing the number of comparisons required to determine itemset support. While we center our discussion on frequent itemsets, the methodologies noted in this section have also been used in designing FSM and FGM algorithms, which we describe in Sections 4 and 5, respectively.

1.2.1 Candidate Generation

A brute-force approach to determine frequent itemsets in a set of transactions is to compute the support for every possible candidate itemset. Given the set of items I and a partial order with respect to the subset operator, one can denote all possible candidate itemsets by an *itemset lattice*, in which nodes represent itemsets and edges correspond to the subset relation. Figure 1 shows the itemset lattice containing candidate itemsets for example transactions denoted in Table 1. The brute-force approach would compare each candidate itemset with every transaction $C \in \mathcal{T}$ to check for containment. An approach like this would require $O(|\mathcal{T}| \cdot L \cdot |I|)$ item comparisons, where the number of non-empty itemsets in the lattice is $L = 2^{|I|} - 1$. This type of computation becomes prohibitively expensive for all but the smallest sets of items and transaction sets.

One way to reduce computational complexity is to reduce the number of candidate itemsets tested for support. To do this, algorithms rely on the observation that every candidate itemset of size k is the union of two candidate itemsets of size $(k-1)$, and on the converse of the following lemma.

tid	items
1	a, b, c
2	a, b, c
3	a, b, d
4	a, b
5	a, c
6	a, c, d
7	c, d
8	b, c, d
9	a, b, c, d
10	d

Table 1:
Example transactions with items from the set $I = \{a, b, c, d\}$.

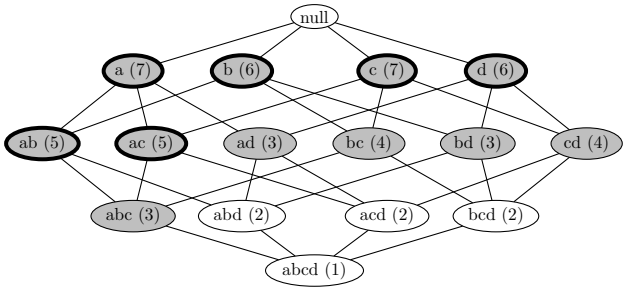

Figure 1: An itemset lattice for the set of items $I = \{a, b, c, d\}$. Each node is a candidate itemset with respect to transactions in Table 1. For convenience, we include each itemset frequency. Given $\sigma = 0.5$, tested itemsets are shaded gray and frequent ones have bold borders.

Lemma 1.1 (Downward Closure) *The subsets of a frequent itemset must be frequent.*

Conversely, the supersets of an infrequent itemset must be infrequent. Thus, given a sufficiently high minimum support, there are large portions of the itemset lattice that do not need to be explored. None of the white nodes in Figure 1 must be tested, as they do not have at least two frequent parent nodes. This technique is often referred to as *support-based pruning* and was first introduced in the Apriori algorithm by Agrawal and Srikant [4].

Algorithm 1 Frequent itemset discovery with Apriori.

1: $k = 1$
2: $F_k = \{i | i \in I, \phi(\{i\}) \geq \sigma\}$
3: **while** $F_k \neq \emptyset$ **do**
4: $\quad k = k + 1$
5: $\quad F_k = \{C | C \in F_{k-1} \times F_{k-1}, |C| = k, \phi(C) \geq \sigma\}$
6: **end while**
7: $Answer = \bigcup F_k$

Algorithm 1 shows the pseudo-code for Apriori-based frequent itemset discovery. Starting with each item as an itemset, the support for each itemset is computed, and itemsets that do not meet the minimum support threshold σ are removed. This results in the set $F_1 = \{i | i \in I, \phi(\{i\}) \geq \sigma\}$ (line 2). From F_1, all candidate itemsets of size two can be generated by joining frequent itemsets of size one, $F_2 = \{C | C \in F_1 \times F_1, |C| = 2, \phi(C) \geq \sigma\}$. In order to avoid re-evaluating itemsets of size one, only those sets in the Cartesian product which have size two are checked. This process can be generalized for all $F_k, 2 \leq k \leq |I|$ (line 5). When $F_k = \emptyset$, all frequent itemsets have been discovered and can be expressed as the union of all frequent itemsets of size no more than k, $F_1 \cup F_2 \cup \cdots \cup F_k$ (line 7).

In practice, the candidate generation and support computation step (line 5) can be made efficient with the use of a *subset function* and a hash tree. Instead of computing the Cartesian product, $F_{k-1} \times F_{k-1}$, we consider all subsets of size k within all transactions in $\mathcal{T}$. A subset function takes as input a transaction and returns all its subsets of size k, which become candidate itemsets. A hash tree data structure can be used to efficiently keep track of the number of times each candidate itemset is encountered in the database, i.e. its support count. Details for the construction of the hash tree can be found in the work of Agrawal and Srikant [4].

1.2.2 Pattern Growth

Apriori-based algorithms process candidates in a breath-first search manner, decomposing the itemset lattice into level-wise itemset-size based equivalence classes: k-itemsets must be processed before $(k+1)$-itemsets. Assuming a lexicographic ordering of itemset items, the search space can also be decomposed into prefix-based and suffix-based equivalence classes. Figures 2 and 3 show equivalence classes for 1-length itemset prefixes and 1-length itemset suffixes, respectively, for our test database. Once frequent 1-itemsets are discovered, their equivalence classes can be mined independently. Patterns are *grown* by appending (prepending) appropriate items that follow (precede) the parent's last (first) item in lexicographic order.

Zaki [63] was the first to suggest prefix-based equivalence classes as a means of independent sub-lattice mining in his algorithm, Equivalence CLAss Transformation (ECLAT). In order to improve candidate support counting, Zaki transforms the transactions into a *vertical database* format. In essence, he creates an inverted index, storing,

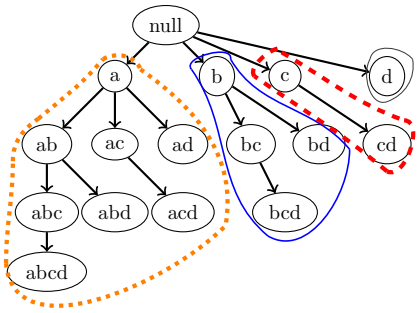

Figure 2: Prefix tree showing prefix-based 1-length equivalence classes in the itemset lattice for $I = \{a, b, c, d\}$.

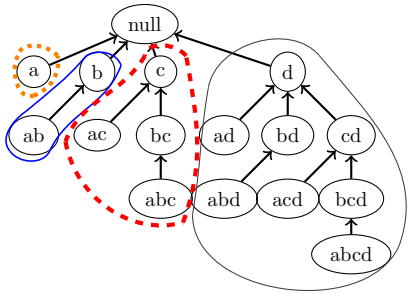

Figure 3: Suffix tree showing suffix-based 1-length equivalence classes in the itemset lattice for $I = \{a, b, c, d\}$.

for each itemset, a list of tids it can be found in. Frequent 1-itemsets are then those with at least $\lceil \sigma |\mathcal{T}| \rceil$ listed tids. He uses lattice theory to prove that if two itemsets C_1 and C_2 are frequent, so will their intersection set $C_1 \cap C_2$ be. After creating the vertical database, each equivalence class can be processed independently, in either breath-first or depth-first order, by recursive intersections of candidate itemset tid-lists, while still taking advantage of the downward closure property. For example, assuming $\{b\}$ is infrequent, we can find all frequent itemsets having prefix a by intersecting tid-lists of $\{a\}$ and $\{c\}$ to find support for $\{ac\}$, then tid-lists of $\{ac\}$ and $\{d\}$ to find support for $\{acd\}$, and finally tid-lists of $\{a\}$ and $\{d\}$ to find support for $\{ad\}$. Note that the $\{ab\}$-rooted subtree is not considered, as $\{b\}$ is infrequent and will thus not be joined with $\{a\}$.

A similar divide-and-conquer approach is employed by Han et al. [22] in FP-growth, which decomposes the search space based on length-1 suffixes. Additionally, they reduce database scans during the search by leveraging a compressed representation of the transaction database, via a data structure called an FP-tree. The FP-tree is a specialization of a prefix-tree, storing an item at each node, along with the support count of the itemset denoted by the path from the root to that node. Each database transaction is mapped onto a path in the tree. The FP-tree also keeps pointers between nodes containing the same item, which helps identify all itemsets ending in a given item. Figure 4 shows an FP-tree constructed for our example database. Dashed lines show item-specific inter-node pointers in the tree.

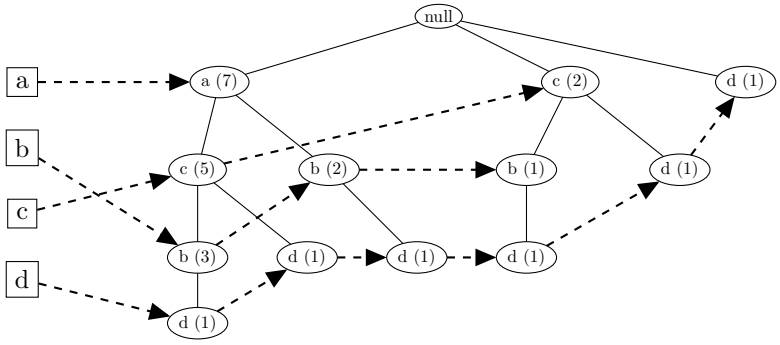

Figure 4: The FP-tree built from the transaction set in Table 1.

Since the ordering of items within a transaction will affect the size of the FP-tree, a heuristic attempt to control the tree size is to insert items into the tree in non-increasing frequency order, ignoring infrequent items. Once the FP-tree has been generated, no further passes over the transaction set are necessary. The frequent itemsets can be mined directly from the FP-tree by exploring the tree from the bottom-up, in a depth-first fashion.

A concept related to that of equivalence class decomposition of the itemset lattice is that of *projected databases*. After identifying a region of the lattice that can be mined independently, a subset of $\mathcal{T}$ can be retrieved that only contains itemsets represented in that region. This subset, which may be much smaller than the original database, is then used to mine patterns in the lattice region. For example, mining patterns in the suffix-based equivalence class of $\{b\}$ only requires data from tids 1, 2, 3, 4, 8, and 9, which contain $\{b\}$ or $\{ab\}$ as prefixes.

2 Paradigms for Big Data Computation

The challenges of working with Big Data are two-fold. First, dataset sizes have increased much faster than the available memory of a workstation. The second challenge is the computation time required to find a solution. Computational parallelism is an essential tool for managing the massive scale of today's data. It not only allows one to operate on more data than could fit on a single machine, but also gives speedup opportunities for computationally intensive applications. In the remainder of this section we briefly discuss the principles of parallel algorithm design, outlining some challenges specific to the frequent pattern mining problem, and then detail general approaches for addressing these challenges in shared and distributed memory systems.

2.1 Principles of Parallel Algorithms

Designing a parallel algorithm is not an easy prospect. In addition to all of the challenges associated with serial algorithm design, there are a host of issues specific to parallel computation that must be considered. We will briefly discuss the topics of memory scalability, work partitioning, and load balancing. For a more comprehensive look at parallel algorithm design, we refer the reader to Grama et al. [17].

As one might imagine, extending the serial FIM methods to parallel systems need not be difficult. For example, a serial candidate generation based algorithm can be made parallel by replicating the list of transactions $\mathcal{T}$ at each process, and having each process compute the support for a subset of candidate itemsets in a globally accessible hash tree. These "direct" extensions however, rely on assumptions like unlimited process memory and concurrent read / concurrent write architecture, which ignore the three challenges outlined at the outset of this section.

One of the key factors in choosing to use parallel algorithms in lieu of their serial counterparts is data that is too large to fit in memory on a single workstation. Even while the input dataset may fit in memory, intermediary data such as candidate patterns and their counts, or data structures used during frequent pattern mining, may not. *Memory scalability* is essential when working with Big Data as it allows an application to cope with very large datasets by increasing parallelism. We call an algorithm memory scalable if the required memory per process is a function of $\Theta(\frac{n}{p}) + O(p)$, where n is the size of the input data and p is the number of processes executed in parallel. As the number of processes grows, the required amount of memory per process for a memory scalable algorithm decreases. A challenge in designing parallel FPM algorithms is thus finding ways to split both the input and intermediary data across all processes in such a way that no process has more data than it can fit in memory.

A second important challenge in designing a successful parallel algorithm is to decompose the problem into a set of *tasks*, where each task represents a unit of work, s.t. tasks are independent and can be executed concurrently, in parallel. Given these independent tasks, one must devise a *work partitioning*, or *static load balancing* strategy, to assign work to each process. A good work partitioning attempts to assign equal amounts of work to all processes, s.t. all processes can finish their computation at the same time. For example, given an $n \times n$ matrix, an $n \times 1$ vector, and p processes, a good work partitioning for the dense matrix-vector multiplication problem would be to assign each process n/p elements of the output vector. This assignment achieves the desired goal of equal loads for all processes. Unlike this problem, FPM is composed of inherently irregular tasks. FPM tasks depend on the type and size of objects in the database, as well as the chosen minimum support threshold σ. An important challenge is then to correctly gauge the amount of time individual tasks are likely to take in order to properly divide tasks among processes.

A parallel application is only as fast as its slowest process. When the amount of work assigned to a process cannot be correctly estimated, work partitioning can lead to a load imbalance. *Dynamic load balancing* attempts to minimize the time that processes are idle by *actively* distributing work among processes. Given their irregular tasks, FPM algorithms are prime targets for dynamic load balancing. The challenge of achieving good load balance becomes that of identifying points in the algorithm execution when work can be re-balanced with little or no penalty.

2.2 Shared Memory Systems

When designing parallel algorithms, one must be cognizant of the memory model they intend to operate under. The choice of memory model determines how data will be stored and accessed, which in turn plays a direct role in the design and performance of a parallel algorithm. Understanding the characteristics of each model and their associated challenges is key in developing scalable algorithms.

Shared memory systems are parallel machines in which processes share a single memory address space. Programming for shared memory systems has become steadily more popular in recent years due to the now ubiquitous multi-core workstations. A major advantage of working with shared memory is the ease of communication between

processes. Any cooperation that needs to take place can be accomplished by simply changing values in memory. Global data structures may be constructed and accessed with ease by all processes.

Designing algorithms for shared memory systems comes with its own set of challenges. A consequence of shared memory programming is the need to acknowledge hardware details such as cache size and concurrent memory accesses. If two processes attempt to write to the same memory address at the same time, one may overwrite the other, resulting in a *race condition*. Moreover, a particular memory address may not necessarily be located on the same physical machine that is running the process, incurring large network delays.

With an appropriately sized dataset or a sufficiently large shared memory system, FPM input and intermediary data can be made to fit in memory. In this case, the challenge is that of structuring mining tasks in such a way as to minimize processes contention for system resources. To achieve this, processes could cooperate in counting support by assigning each process a distinct subset of the database and having them compute support for all candidate itemsets with respect to their portion of the database. This, however, can lead to race conditions when multiple processes increment counts for the same candidate itemset. A better approach decomposes the candidate itemset lattice into disjoint regions based on equivalence classes, as described in Section 1, and assigns distinct regions of the lattice to processes. In this way, no two processes will be responsible for updating the count of the same itemset, eliminating the possibility of race conditions.

2.3 Distributed Memory Systems

In a distributed (*share-nothing*) memory system, processes only have access to a local private memory address space. Sharing input data and task information must be done explicitly through inter-process communication. While processes most often communicate through network transmissions, they can also exchange information by writing and reading files on a shared file system. There are two programming paradigms commonly used for distributed memory systems. *Message passing* is a classic model that has been used very successfully in the scientific computing community for several decades. *MapReduce* is a recent paradigm, developed by Dean and Ghemawat [13] and designed specifically for working with Big Data applications.

2.3.1 Message Passing

The message passing paradigm implements the *ACTOR model of computation* [23], which is characterized by inherent computation concurrency within and among dynamically spawned actors. In the message passing paradigm, processes are actors, and they interact only through direct message passing. Messages are typically sent over a network connection and their transmission is thus affected by available inter-process network bandwidth. An advantage of the message passing paradigm is that the developer has total control over the size and frequency of process communication. Since all communication must be invoked through sending and receiving a message, the application must attempt to minimize network traffic, which is significantly slower than accessing local memory.

Parallel FPM algorithms designed for message passing based systems typically partition data such that each process holds an equal share of the input in local memory. As a consequence, subsets of the local transactions must be shared with other processes that need them. For example, in candidate generation approaches, one may choose to broadcast the local input among all processes, during each iteration of the algorithm, in a round robin fashion. The amount of communication necessary in this scenario may be a detriment to the overall execution. Moreover, the set of candidate itemsets must also be partitioned across processes, incurring additional communication overhead. An alternative approach may use a pattern growth based algorithm that mines distinct projected databases associated with an equivalence class. Once a projected database has been extracted by communicating with the other processes that store its transactions, it can be mined by a process without further communication overhead. However, care must be taken to choose small enough equivalence classes s.t. their projected databases and count data structures fit in the local process memory.

2.3.2 MapReduce

MapReduce [13] is a recent programming model for distributed memory systems that has become very popular for data-intensive computing. By using a restricted program model, MapReduce offers a simple method of writing parallel programs. While originally a proprietary software package at Google, several successful MapReduce open-source implementations have been developed, of which Hadoop [59] is currently the most popular.

Computation in MapReduce consists of supplying two routines, Map and Reduce. The problem input is specified as a set of *key-value pairs*. Each key is processed by an invocation of Map, which emits another (possibly different) key-value pair. Emitted key-value pairs are grouped by key by the MapReduce framework, and then the list of

grouped values in each group is processed individually by a REDUCE invocation. REDUCE in turn emits either the final program output or new key-value pairs that can be processed with MAP for another iteration.

The canonical example of a MapReduce program is word frequency counting in a set of documents. Program input is organized into key-value pairs of the form `<ID, text>`. A MAP process is assigned to each `ID`. MAP iterates over each word w_i in its assigned document and emits the pair $\langle w_i, 1 \rangle$. Pairs from all MAP processes are grouped and passed to a single REDUCE process. Finally, REDUCE counts the appearances of each w_i and outputs the final counts.

Individual MAP and REDUCE tasks are independent and can be executed in parallel. Unlike the message passing paradigm, parallelism in MapReduce is *implicit*, and the program developer is not required to consider low-level details such as data placement or communication across memory address spaces. The MapReduce framework manages all of the necessary communication details and typically implements key-value pair transmissions via a networked file system such as GFS [15] or HDFS [7]. Certain MapReduce implementations provide some means to make global read-only data easily accessible by all MAP processes. In Hadoop, this is achieved through the use of a *Distributed Cache*, which is an efficient file caching system built into the Hadoop runtime.

A criticism of the model comes from its heavy reliance on disk access between the MAP and REDUCE stages when key-value pairs must be grouped and partitioned across REDUCE processes. If the computation is iterative in nature, a naïve MapReduce implementation could require shuffling the data between physical disk drives in each iteration.

Pattern growth methods are popular in the MapReduce environment, due to their ability to decompose the problem into portions that can be independently solved. After computing the support of 1-length patterns in a similar manner as the word frequency counting example previously described, equivalence classes can be mapped to different processes via MAP. A serial algorithm is used to complete the local task, then a REDUCE job gathers the output from all processes.

3 Frequent Itemset Mining

Much like its serial counterpart, there are two main approaches to solving the parallel frequent itemset mining (PFIM) problem, namely, candidate generation and pattern growth. In this section, we discuss a number of proposed PFIM algorithms, paying special attention to the algorithmic details addressing the three challenges introduced in Section 2. For easy reference, Table 2 lists the serial and parallel methods described in this section.

3.1 Memory Scalability

Serial FIM algorithms can be easily parallelized if memory constraints are ignored. Among the first to address the memory constraint issues related to PFIM were Agrawal and Shafer [3], who proposed the Count Distribution (CD) algorithm. In their method, the list of transactions $\mathcal{T}$ is distributed among the processes s.t. each process is responsible for computing the support of all candidate itemsets with respect to its local transactions. Instead of a globally accessible hash tree, each process builds a local hash tree which includes all candidate itemsets. Then, with a single pass over the local transactions, the local support for each candidate itemset can be computed. The global support for each candidate itemset can then be computed as the sum of each process' local support for the given candidate itemset, using a global reduction operation. At the same time that Agrawal and Shafer introduced the CD algorithm, Shintani and Kitsuregawa [52] introduced an identical algorithm which they called Non-Partitioned Apriori (NPA).

Since each process can build its hash tree and compute its local support for candidate itemsets independently, the only inter-process communication required is during the global reduction operation. However, since the hash tree data structure is built serially by each process, for all candidate itemsets, a bottleneck is realized when the process count becomes sufficiently high. Also, since each process will have a local copy of the hash tree corresponding to all candidate itemsets, if the number of candidate itemsets becomes too large, the hash tree will likely not fit in the main memory of each process. In this case, the hash tree will have to be partitioned on disk and the set of transactions scanned once for each partition to compute candidate itemset supports, a computationally prohibitive exercise in the context of Big Data. Mueller developed a serial algorithm for FIM, named SEAR [40], which is identical to Apriori with the exception of using a trie in place of the hash tree data structure. Furthermore, Mueller extended SEAR along the lines of the CD algorithm, which he called PEAR [40].

Another PFIM algorithm based on the CD algorithm is the CD TreeProjection algorithm developed by Agarwal et al. [2]. Their algorithm is a parallel extension of their TreeProjection algorithm. In the CD TreeProjection algorithm, identical lexicographic trees are built on each process, in place of the hash trees of the original CD algorithm. From the lexicographic trees, the support counts are computed and globally reduced. This method shares scalability characteristics with other CD based algorithms.

Table 2: Serial and parallel frequent itemset mining algorithms.

Type	Acronym	Name	Cite
ser	Apriori	Apriori	[4]
ser	ECLAT	Equivalence CLAss Transformation	[63]
ser	FP-growth	Frequent Pattern Growth	[22]
ser	Partition	Partition	[51]
ser	SEAR	Sequential Efficient Association Rules	[40]
ser	TreeProjection	TreeProjection	[2]
par	BigFIM	Frequent Itemset Mining for Big Data	[39]
par	CCPD	Common Candidate Partitioned Database	[66]
par	CD	Count Distribution	[3]
ser	CD TreeProjection	Count Distributed TreeProjection	[2]
par	DD	Data Distribution	[3]
par	Dist-Eclat	Distributed Eclat	[39]
par	DPC	Dynamic Passes Combined-counting	[33]
par	FPC	Fixed Passes Combined-counting	[33]
par	HD	Hybrid Distribution	[19]
par	HPA	Hash Partitioned Apriori	[52]
par	HPA-ELD	HPA with Extremely Large Itemset Duplication	[52]
par	IDD	Intelligent Data Distribution	[19]
ser	IDD TreeProjection	Intelligent Data Distribution TreeProjection	[2]
par	NPA	Non-Partitioned Apriori	[52]
par	ParEclat	Parallel Eclat	[67]
par	Par-FP	Parallel FP-growth with Sampling	[9]
par	PCCD	Partitioned Candidate Common Database	[66]
par	PEAR	Parallel Efficient Association Rules	[40]
par	PPAR	Parallel PARTITION	[40]
par	SPC	Single Pass Counting	[33]

Lin et al. [33] introduced three variations of the CD algorithm for the MapReduce framework, namely Single Pass Counting (SPC), Fixed Passes Combined-counting (FPC), and Dynamic Passes Combined-counting (DPC). In SPC, the responsibility of the Map function is to read the list of all candidate itemsets from the Distributed Cache and count the frequency of each of them with respect to a local partition of $\mathcal{T}$. Then, the Reduce function computes a global summation of local support counts and outputs to the distributed cache the new set of candidate itemsets for the next iteration. SPC is a direct extension of CD to MapReduce. The FPC and DPC algorithms are optimizations of SPC which combine iterations of the classic Apriori algorithm into a single pass.

The PPAR algorithm of Mueller [40], which is the natural parallel extension of the algorithm Partition, takes a slightly different approach to address the memory scalability challenge. In Partition, Savasere et al. [51] partition $\mathcal{T}$ horizontally and each partition is processed independently. The union of the resulting frequent itemsets, which are locally frequent in at least one partition, are then processed with respect to each partition in order to obtain global support counts. Mueller splits $\mathcal{T}$ into p partitions and assigns one to each of the p processes. Once all processes have finished identifying the locally frequent itemsets, a global exchange is performed to get the union of all locally frequent itemsets. Then, each process gathers the counts for the global candidate itemsets. Finally, the counts are globally reduced and the final set of frequent itemsets is identified. Like CD, PPAR has the advantage that the entire list of transactions does not need to fit in the main memory of each process. However, the set union of locally frequent itemsets may be much larger than the largest set of candidate itemsets in CD, and may not fit in the local memory available to each process.

Agrawal and Shafer [3] proposed a second method, which they called Data Distribution (DD), to address the memory scalability limitations of CD. To this end, they suggest that the candidate itemsets, as well as $\mathcal{T}$, should be distributed, using a round-robin scheme. Since processes have access to only a subset of the candidate itemsets, each is responsible for computing the global support for its local candidate itemsets. In order to compute global support for the local candidate itemsets, each process needs to count the occurrences of each of its candidate itemsets with respect to the entire list of transactions. With each process given access to only a portion of $\mathcal{T}$, this requires processes to share their local portions of $\mathcal{T}$ with each other. This type of communication pattern, all-to-all, can be expensive depending on the capabilities of the framework/architecture on which it is implemented. In a distributed system,

using a message passing framework, the exchange of local partitions of $\mathcal{T}$ can be done efficiently using a ring-based all-to-all broadcast, as implemented by Han et al. [19] in their Intelligent Data Distribution (IDD) algorithm.

Each of the methods described above address the candidate itemset count explosion, inherent in candidate generation based algorithms. Zaki et al. [67] proposed a parallel extension of their serial algorithm, ECLAT, which they called ParEclat. The parallelization strategy in ParEclat is to identify a level of the itemset lattice with a large enough number of equivalence classes s.t. each process can be assigned some subset of classes. At this point, each process can proceed to mine the classes it was assigned, independently of other processes. In the end, a reduction is performed to gather all of the frequent itemsets as the result. In order to mine the subset of classes assigned to it, each process needs access to all of the tid-lists belonging to items in its assigned classes. Because the equivalence classes of ECLAT can be processed independently, no communication is needed after the initial class distribution. In many cases, this vertical exploration of the itemset lattice, as opposed to the horizontal exploration of Apriori based approaches, leads to much smaller memory requirements. However, ParEclat makes an implicit assumption that the tid-lists associated with the subset of classes being explored fits in the main memory of a single process, which may not always be the case. Consider the case that some subset of items appears in a significant portion of $\mathcal{T}$, then the corresponding tid-lists for these items will be a large fraction of the original list of transactions. Based on our original assumption that $\mathcal{T}$ does not fit in the memory of a single process, it can be concluded that ParEclat is memory scalable for only certain classes of input.

Another method similar to ParEclat was introduced by Cong et al. [9]. Their method, called Par-FP, follows the same parallelization strategy as ParEclat, but uses standard FP-Growth as its underlying algorithm instead of ECLAT. Like ParEclat, Par-FP is also a pattern generation method and is thus formulated within similar memory assumptions. Moens et al. [39] proposed Dist-Eclat and BigFIM in the MapReduce context. Dist-Eclat follows the same approach as ParEclat, adapted to the MapReduce framework. BigFIM is a hybrid approach that uses a MapReduce equivalent of CD to compute candidate itemsets level-wise until each process' assigned set of candidate itemsets no longer fits in its memory, at which point the algorithm transitions to a pattern growth approach.

Zaki et al. [66] proposed shared memory implementations of both the CD algorithm and the DD algorithm. In their implementation of the CD algorithm, which is referred to as Common Candidate Partitioned Database (CCPD), each process computes support for all candidates with respect to a distinct partition of the list of transactions. The support counts are aggregated in a shared candidate hash tree, whose access is controlled with a locking mechanism at the leaf of each node. The Partitioned Candidate Common Database (PCCD) algorithm is the shared memory implementation of the DD algorithm. Zaki et al. observed that, although there is no locking required, the increased disk contention from all processes scanning a shared list of transactions leads to a slowdown when using more than one process, due to processes operating on disjoint candidate hash trees.

3.2 Work Partitioning

A limitation of the DD algorithm is that it leads to redundant work when compared with CD, which performs the same amount of work as its serial counterpart. Let the cost of incrementing the count for a candidate itemset that exists in a process' hash tree be a function $f(X)$ of the number of unique candidate itemsets stored in the hash tree X. Then, in the case of the CD algorithm, the function becomes $f(M_k)$, where M_k is the total number of candidate itemsets of a given size k. In contrast, for the DD algorithm, the cost is $f(M_k/p)$. Furthermore, in the CD algorithm, each process is required to process $|\mathcal{T}|/p$ transactions, while in DD, each process is responsible for $|\mathcal{T}|$. Since each transaction processed will generate the same number of k-size candidate itemsets, the computational cost for each algorithm at iteration k can be modeled as follows:

$$\texttt{Cost}_{CD} = \frac{|T|}{p} \times f(M_k), \quad \text{and}$$
$$\texttt{Cost}_{DD} = |T| \times f\left(\frac{M_k}{p}\right).$$

In order for the DD algorithm to be as computationally efficient as the CD algorithm, $f(M_k/p)$ must be less than $f(M_k)$ by a factor of p. However, Han et al. [19] showed that $f(M_k/p) > f(M_k) \times 1/p$, thus the DD algorithm introduces redundant computation.

The authors proposed a few optimizations to the distribution and processing of transactions which can appropriately address these issues. When applied to the DD algorithm, these optimizations are referred to as Intelligent Data Distribution (IDD). In IDD, candidate itemsets are distributed to processes based on their prefix, following a lexicographic ordering or items. Each process is assigned a subset of I as prefixes they are responsible for. A process can quickly check to see if a transaction will generate any candidate itemsets which start with items from its subset

of I and skip non-matching transactions. Therefore, a process will only traverse its hash tree for those candidate itemsets it is responsible for.

IDD uses a bin-packing algorithm [41] to ensure that each process receives approximately the same number of candidate itemsets and all candidate itemsets assigned to a particular process begin with the items of its subset of I. During each iteration, the number of candidate itemsets starting with each frequent item are counted. Then, the candidate itemsets are partitioned into p different bins, such that the sum of the numbers of candidate itemsets starting with the items in each bin are roughly equal. Longer prefixes can be considered if initial partitioning does not lead to evenly packed bins, until the bins are evenly packed. The authors theoretically and experimentally show that IDD addresses the memory bottleneck issues of the CD algorithm, as well as the redundant work introduced by the DD algorithm. In Intelligent Data Distributed TreeProjection Algoirthm, Agarwal et al. [2] distributed the lexicographic tree by assigning to each process, subtrees associated with specific first items. The lexicographic tree is distributed s.t. the sum of the supports of the subtrees assigned to a process is as close to balanced as possible.

Shintani and Kitsuregawa [52] also recognized the redundant work inherent to the DD algorithm. Unlike the approach in IDD, the Hash Partitioned Apriori (HPA) algorithm uses a hash function to determine the process responsible for a subset of candidate itemsets. In other words, each process scans its portion of the transaction set and generates k-size candidate itemsets. The algorithm hashes the itemsets and determines the process responsible for computing their support. If an itemset is hashed to its own process id, then the process increases support count for the itemset in its own hash tree. Otherwise, the candidate itemset is sent to the process to which it was hashed, so that its support count can be incremented there.

In ParEclat, Zaki et al. [67] employed a greedy algorithm that distributes work evenly among processes. First, each equivalence class is given a weight based on its cardinality. This weight is then used to distribute classes so that all processes are assigned subsets of classes with even weight.

A more advanced work distribution scheme for pattern growth methods was proposed by Cong et al in [9], for their Par-FP algorithm. The authors suggested the use of *selective sampling* as a preprocessing step to estimate the cost for FP-Growth execution rooted at a particular item. In selective sampling, the t most frequent items, where t is an application parameter, are discarded from all transactions, along with any infrequent items. An FP-Tree is built from the resulting set of items and mined by a single process, recording the execution time of mining the projected database of each sampled item. After the sampling step, the work of mining each item $i \in I$ is partitioned in the following way. First, of the sampled items, any identified as *large* are further split into subtasks which are assigned to processes in a round-robin fashion. A large item is defined as any item whose sample execution time is significantly higher than the expected execution time given an equal distribution of work. Then, all other sampled items are assigned using a binning algorithm based on their sample execution time. Finally, those frequent items which were not sampled are assigned to processes in a round-robin fashion.

3.3 Dynamic Load Balancing

The estimation of work execution cost in the context of FIM can be done in a reasonably accurate way on homogeneous systems through work partitioning, as a preprocessing step to the algorithm execution. MapReduce systems are often heterogeneous, often composed of nodes with diverse compute power or resource availability. Lin et al. [33] noted that the number of candidate itemsets is typically small during later iterations of Apriori in his SPC algorithm, leading to increased overhead due to startup and scheduling of MapReduce jobs. Therefore, the authors proposed two dynamic load balancing improvements to SPC, the FPC and DPC algorithms. FPC combines a fixed number of Apriori iterations into a single pass, which means less MapReduce overhead as well as less scans over the list of transactions, but can also lead to many false positive results. The DPC algorithm dynamically chooses how many iterations to combine during each Map phase, via two heuristics. First, in a single Map phase, the DPC algorithm will combine as many iterations as it can without exceeding a specified maximum number of candidate itemsets. Since MapReduce systems can be heterogeneous, it is possible that the appropriate threshold for one compute node leads to a significantly longer or shorter execution time on a different node for the same iteration. The second heuristic of DPC dynamically adjusts the threshold value based on the execution times from previous iterations. In this way, the DPC algorithm can accommodate systems with dynamically changing resource availability.

3.4 Further Considerations

As is the case with most things in parallel computing, there is no silver bullet. Han et al. [19] noted that, as the process count to number of candidate itemsets ratio increases, the number of candidate itemsets assigned to each process in the IDD algorithm is reduced, shrinking the size of each process' hash tree and the amount of computation work per transaction. For a sufficiently high number of processes or sufficiently low number of candidate itemsets,

Table 3: Serial and parallel frequent sequence mining algorithms.

Type	Acronym	Name	Cite
ser	AprioriAll	AprioriAll	[5]
ser	BIDE	BI-Directional Extension	[57]
ser	CloSpan	Closed Sequential Pattern Mining	[61]
ser	FreeSpan	Frequent Pattern-projected Sequential Pattern Mining	[21]
ser	GSP	General Sequential Patterns	[54]
ser	PrefixScan	Prefix-projected Sequential Pattern Mining	[42]
ser	SPADE	Sequential PAttern Discovery using Equivalence classes	[62]
ser	WAP-miner	Web Access Pattern Miner	[43]
par	ACME	Advanced Parallel Motif Extractor	[50]
par	DGSP	Distributed GSP	[45]
par	DPF	Data Parallel Formulation	[18]
par	EVE	Event Distribution	[27]
par	EVECAN	Event and Candidate Distribution	[27]
par	HPSPM	Hash Partitioned Sequential Pattern Mining	[53]
par	MG-FSM	Large-Scale Frequent Sequence Mining	[38]
par	NPSPM	Non-Partitioned Sequential Pattern Mining	[53]
par	Par-ASP	Parallel PrefixSpan with Sampling	[9]
par	Par-CSP	Parallel CloSpan with Sampling	[10]
par	PLUTE	Parallel Sequential Patterns Mining	[44]
par	pSPADE	Parallel SPADE	[64]

this decrease in work per transaction will mean that, in distributed systems, the communication of the transactions between processes will become the limiting component of the IDD algorithm. Therefore, the authors introduce the Hybrid Distribution algorithm, which uses both the CD and IDD algorithms. To do this, the processes are split into g equally sized groups. Each process group is responsible for computing the support for all candidate itemsets with respect to the $|\mathcal{T}|/g$ transactions assigned to it. This can be viewed conceptually as executing the CD algorithm on g pseudo-processes. Within each group, the local support for all candidate itemsets is computed by executing the IDD algorithm on the p/g processes in the group. Each process in a group computes the support of the candidate itemsets assigned to it with respect to the $|\mathcal{T}|/g$ assigned to its process group. The choice of g can be determined automatically at runtime. When the number of candidate itemsets is not sufficiently high to ensure that each process in the system is assigned enough candidate itemsets to offset the cost of communicating the transactions within each process group, then g is set to p, meaning the CD algorithm is executed with each process as its own group. Otherwise, g is chosen small enough to ensure each process gets an adequate number of candidate itemsets so that work outweighs communication.

Shintani and Kitsuregawa [52] noted that the HPA algorithm suffers a similar drawback as the IDD algorithm. Namely, that if the process count to number of candidate itemsets ratio becomes sufficiently high, then the system can become under utilized. For this reason, they also introduced the HPA with Extremely Large Itemset Duplication (HPA-ELD) algorithm. In the case of the HPA algorithm, the under utilization comes from the possibility that the number of candidate itemsets will be small enough so that the memory available to each process is not completely full. To address this, HPA-ELD duplicates the highest frequency itemsets on all processes until the system memory is full. Then, the support for these most frequent candidate itemsets is computed locally, just like in the CD algorithm, while the support for the rest of the candidate itemsets is computed in the same way as in HPA.

4 Frequent Sequence Mining

Many of the solutions to FSM problems follow algorithms developed for FIM. Furthermore, parallel FSM (PFSM) algorithms often directly extend serial ones and their parallelization strategies are heavily inspired by previously developed PFIM algorithms. We will briefly discuss some of the more prominent serial approaches to solving the FSM problem, and focus the remainder of the section on the challenges of parallelizing these approaches. For easy reference, Table 3 lists the serial and parallel methods described in this section.

4.1 Serial Frequent Sequence Mining

As direct extensions of the Apriori algorithm for the FIM problem, AprioriAll [5] and General Sequential Patterns (GSP) [54], by Srikant and Agrawal, make use of the downward closure property and follow the same general multi-pass candidate generation outline as described in Section 1. The join operation is redefined for the sequence domain s.t. only $(k-1)$-length frequent sequences with the same $(k-2)$-prefix are joined. GSP generalizes the problem definition, introducing time constraints (minimum or maximum time period between elements in a frequent sequence), a sliding window (sequence itemset items may come from a set of transactions with timestamps within a user-specified time window), and user-defined taxonomies (the set of items I may be provided as a hierarchy and sequential patterns may include items across all levels of the hierarchy).

Pattern growth methods also exhibit direct extensions to the sequence domain. Zaki [65] developed Sequential PAttern Discovery using Equivalence classes (SPADE), patterned after their ECLAT [62] algorithm for FIM, which is introduced in Section 1. During the first scan of the sequence database, SPADE creates, for each discovered sequence element (itemset), an *id-list* containing `<sequence id, timestamp>` pairs denoting locations of the element in the database. Candidate support counts can then be computed via id-list intersections. Zaki used lattice theory [12] to show that any possible frequent sequence can be obtained as a union or join of shorter frequent sequences. He proposed both *breath-first-search* and *depth-first-search* approaches to recursively decompose the lattice into prefix-based classes, providing an alternative to the breath-first-search approach of GSP.

Han et al. [21] introduced Frequent pattern-projected Sequential pattern mining (FreeSpan), which Pei et al. [42] followed with Prefix-projected Sequential pattern mining (PrefixSpan). FreeSpan and PrefixSpan recursively partition the sequence data based on frequent items and sequence prefixes, respectively. They first scan the database and derive frequent items. Then, projected databases are constructed for each of the frequent items, or length-1 frequent sequences, eliminating those sequences that only contain infrequent items. By choosing to partition on sequence prefixes, PrefixSpan is able to find all frequent sequences by examining only prefix sub-sequences and projecting only their corresponding postfix sub-sequences into projected databases. Pei et al. also explore pseudo-projections (when data fits in main memory) and bi-level projections in PrefixSpan, using the downward closure property to prune items in projected databases.

Guralnik and Karypis [18] altered a tree projection algorithm for discovering frequent itemsets by Agarwal et al. [2] to mine frequent sequences. As in the FIM approach, a lexicographic *projection tree* is grown through bi-level candidate sequence generation (nodes at level $k-1$ generate candidates for level $k+1$). Each node in the tree represents a frequent sequence, which can be extended either by adding a lexicographically correct item in the last element of the sequence (*itemset extension*) or a new itemset/element to the sequence (*sequence extension*). Each active node, one that still has a possibility of being extended, maintains four sparse *count matrices* used to efficiently update frequencies during the projection phase.

In the context of Web access mining, Pei et al. [43] developed Web Access Pattern miner (WAP-miner) for mining *symbol sequences* constructed from Web logs. The algorithm uses a Web access pattern tree (*WAP-tree*) as a compressed representation of the sequence database and a *conditional search* tree-projection mechanism for growing sequence patterns. Rajimol and Raju [46] surveyed several WAP-tree based extensions which improve the data structure and projection mechanism of WAP-miner.

While finding all frequent patterns is costly, the majority of the resulting patterns may not be interesting. CloSpan (Yan et al. [61]) and BIDE (Wang and Han [57]) focus on finding *frequent closed sequences*. Though following a candidate generate-and-test methodology, CloSpan prunes more of the search space through *CommonPrefix* and *Backward Sub-Pattern pruning*. It also speeds up testing by noting that projected databases with the same number of items are equivalent. BIDE uses a paradigm called *BI-Directional Extension* to both check closure of candidate sequences and prune the search space.

As a related problem first studied by Mannila et al. [36], *frequent episode mining* finds frequent collections of events that occur relatively close to each other in a long symbol sequence, given a partial event order. Joshi et al. [26] developed a universal formulation for sequence patterns, encompassing both Mannila et al.'s definition of frequent episodes and Srikant and Agrawal's generalized sequential patters. For an in-depth survey of serial frequent pattern mining methods, the reader may consult the work of Han et al. [20].

Motif discovery (finding frequent sub-sequence patterns) is an important problem in the biological domain, especially as a sub-step in sequence alignment. Rigoutsos and Floratos [49] proposed a pattern growth based method for motif discovery with rigid gaps (sets of *don't care* items). Wang et al. [58] use a two step process for motif discovery. They first search for short patterns with no gaps, called segments, and then search for longer patterns made up of segments joined by variable length gaps. Liao and Chen [31] use a vertical-database format they call the *three-dimensional list* to speed up verification of patterns generated via direct spelling, i.e., extending patterns at each level by each of the characters in the often small genetic symbol alphabet.

4.2 Parallel Frequent Sequence Mining

In this section, we discuss a number of proposed parallel sequence mining algorithms, along the lines of their memory scalability, task partitioning, and load balancing choices.

4.2.1 Memory Scalability

Some algorithms aimed at distributed systems assume that either the set of input sequences, the set of candidate sequences, or global count data structures fit in the local memory of each process. This is an unrealistic expectation when mining Big Data. Although their focus is on balancing mining tasks, Cong et al. [9, 10] (Par-FP, Par-ASP, Par-CSP) accomplished the task using a sampling technique that requires the entire input set be available at each process. Shintani and Kitsuregawa [53] partitioned the input set in Non Partitioned Sequential Pattern Mining (NPSPM), yet they assumed that the entire candidate set can be replicated and will fit in the overall memory (random access memory and hard drive) of a process. Similar assumptions were made by Joshi et al. [27] in Event Distribution (EVE) and by Guralnik and Karypis [18] in their Data Parallel Formulation (DPF).

These straight-forward *data parallel formulations* assign work to processes based on the input data they have been assigned. This strategy does not scale well, as global candidate sequences and their counts must also fit in the local process memory. Most authors proposed alternatives that partition both input and intermediary data. Shintani and Kitsuregawa [53] used a hash function in Hash Partitioned Sequential Pattern Mining (HPSPM) to assign input and candidate sequences to specific processes. Joshi et al. [27] proposed to shard both input and candidate sequences in Event and Candidate Distribution (EVECAN) and rotate the smaller of the the two sets among the processes, in round-robin fashion. Guralnik and Karypis introduced several task-parallel formulations, which are discussed in the next section, that partition the problem into independent sub-problems via database projections.

Input partitioning is not inherently necessary for shared memory or MapReduce distributed systems. In the case of shared memory systems, the input data should fit in the aggregated system memory and is available to be read by all processes. However, care must be taken to ensure processes do not simultaneously attempt to read or write to the same block of memory. Zaki [64] extended his serial FSM algorithm (SPADE) to the shared memory parallel architecture, creating pSPADE. Input data is assumed residing on shared hard drive space, stored in the vertical-database format. The author proposed two data parallel formulations in pSPADE that partition the input space s.t. different processes are responsible for reading different sections of the input data (*id-lists*). Processes are either assigned id-lists for a subset of sequences, or portions of all id-lists associated with a range of sequences in $\mathcal{D}$. Then, processes collaborate to expand each node in the itemset lattice. The author found that these formulations lead to poor performance due to high synchronization and memory overheads. He then proposed two task distribution schemes, discussed in the following sections, which are able to avoid read/write conflicts through independent search space sub-lattice assignments.

The HDFS distributed file system provided by the Hadoop MapReduce framework ensures adequate space exists for a large input dataset. However, data elements are replicated across several processing nodes in the system and repeated scans of the data will incur severe hard drive and/or network I/O penalties. Qiao et al. [45] found that a straight-forward extension of GSP for a MapReduce distributed environment (DGSP) performed poorly due to repeated scans of the input data. Instead, the authors used the CD algorithm for 1-length candidate sequence counting in an algorithm called PartSpan, followed by a projection-based task partitioning for solving the remainder of the problem. Similar strategies were followed by Qiao et al. [44] in PLUTE and Miliaraki et al. [38] in MG-FSM.

4.2.2 Work Partitioning

The FSM solution space is inherently irregular. Some processes may be assigned subsections of the input with only short frequent sequences and in effect have less work to complete than the rest of the processes. A number of authors have introduced work partitioning schemes designed to combat these potential problems.

pSPADE [64] recursively decomposes the frequent sequence search space into suffix-based equivalence classes, similar to those shown in Section 1. A node's sub-forest can be computed independently using only the node's input data (id-lists). Zaki proposed a static task partitioning scheme, Static Load Balancing (SLB), in which nodes in the first level of the class-tree and their associated id-lists are assigned in a round-robin way to processes, in reverse order of the number of elements in each node's class. This scheme cannot gauge well the amount of work that is needed to mine each node's class and may still lead to load imbalance.

Solving the FSM problem via lexicographic tree projection in a distributed message-passing environment, Guralnik and Karypis proposed a Data Parallel Formulation (DPF) similar to the Count Distribution strategy described in Section 3. It partitions the input equally among the processes and requires each node to keep track of the lexicographic tree representing frequent candidates and the associated count matrices. Each node projects its local set of sequences

unto level $(k-1)$ of the tree to determine local supports for candidate itemsets at the $(k+1)$'th level. A reduction operation facilitates computing global support of candidate itemsets. Those meeting the minimum support threshold σ are then broadcast to all processes before continuing to the next level. For problems with large numbers of candidates, where the global tree and its associated count matrices do not fit in the local memory of a process, DPF has to partition the tree and perform multiple scans of the local database at each iteration. This added I/O overhead makes this formulation impractical in the Big Data context. The authors also showed that DPF's parallel efficiency will decrease as the number of processes increase, even when the amount of work increases, due to the overhead of maintaining the global tree at each process. To account for these limitations, they propose Static and Dynamic Task-Parallel Formulations (STPF and DTPF), which partition both the input data and the lexicographic tree during computation.

Both task-parallel formulations first use DPF to expand the tree up to a level $k+1,\ k>0$. Further expanding a $(k+1)$-level node requires count data from its parent node. Therefore, nodes at level k are partitioned among the processes for further processing, along with their *projected databases*. A node's projected database contains a subset of the sequence database $\mathcal{D}$ s.t. each item in the itemsets of each transaction is still viable to extend the sequence represented by the node into a possible frequent sequence. These viable items are called *active items*. The question remains how to partition the k-level nodes among the processes. The authors proposed two static approaches. The first uses a bin-packing algorithm based on relative computation time estimates for expanding the nodes, computed as the sum of their children's corresponding sequential patterns' support. The second approach aims to minimize the overlap in the nodes' projected databases via repeated minimum-cut partitioning of the bipartite graph formed by the set of nodes and the set of itemsets at level $(k+1)$ in the tree. The authors found that the bipartite graph partitioning approach is able to substantially reduce the overlap in projected databases and lead to smaller execution times as opposed to other static task parallel formulations. However, its workload prediction accuracy decreases as the numbers of processes increases, leading to unbalanced loads. As a result, the authors introduced dynamic load balancing in DTPF that monitors process workloads and reassigns work as necessary.

Cong et al. took a sampling approach to accomplish static task partitioning in Par-ASP [9], which they named *selective sampling*. After gathering 1-length frequent sequence counts, they separate a sample $\mathcal{S} \subset \mathcal{D}$ of k-length frequent prefixes of sequences in $\mathcal{D}$. The size of the prefix k is a function of the average length of sequences in $\mathcal{D}$. The authors then use one process to mine the sample via a pattern growth algorithm, recording execution times for the found frequent sub-sequences. Mining each frequent sub-sequence becomes a task. Projected databases for these frequent sequences are computed during the mining time estimation. Task distribution is then done in the same way as in the Par-FP algorithm, as described in Section 3. The authors found that the serial sampling component of their algorithm accounted on average for 1.1% of the serial mining time, which limits the speedup potential as the number of processes increases.

Unlike in the general sequence mining problem, infrequent items cannot be removed from projected sequences in the gap-constrained frequency mining problem. Miliaraki et al. proposed several ways to compress projected databases in MG-FSM, based on the concept of *w-equivalency*, i.e., the equivalency of projected databases with respect to a pivot w. Projected databases are constructed and compressed for all 1-length frequent sequences in $\mathcal{D}$ in the Map phase of a MapReduce job. Run-length and variable-byte encoding are used to reduce the size of the projected databases before transmitting them to reducers, which provides a significant performance boost, as reported by the authors in their experiments. A serial algorithm is used in the Reduce phase of the MapReduce job to mine the independent partitions of the sequence database.

4.2.3 Dynamic Load Balancing

The advantage of separating work into partitions that processes can accomplish independently seems clear. Yet it is not always clear how long each partition will take to mine. Dynamic load balancing aims to (re)-distribute the work as necessary when processes finish their assigned tasks, in such a way that processes are allotted equal amounts of work overall.

Zaki extended his static task partitioning scheme (SLB) in pSPADE by forming a task queue. First level class nodes are entered into the queue in the same way as they would have been assigned in SLB. In the inter-Class Dynamic Load Balancing (CDLB) scheme, processes pick up new tasks from the queue, one at a time, as soon as they finish their current work. An abnormally large sub-class may still lead to load imbalance. The Recursive Dynamic Load Balancing (RDLB) scheme exploits both inter and intra-class parallelism, allowing a process to share its work with free ones. Mining of a class sub-tree at the process level takes place in a breath-first manner, level by level. A process signals it is free via a global counter when it finishes its current work and the queue is empty. Another process that still has work then enters its unprocessed nodes on the level he is currently processing into the global queue, providing work for the first process. The author reported RDLB performs best among the three

proposed schemes.

The static task-parallel formulation (STPF) presented by Guralnik and Karypis may suffer some load imbalance as the number of processes increases. The authors combat this through a receiver initiated load-balancing with random polling scheme [17]. Processes are initially assigned tasks according to the STPF scheme. As a process completes its currently assigned work, it sends a request for more work to a randomly chosen other *donor* process. The donor process responds positively, by sending half of the nodes on the current level of the projection tree it is expanding, along with their projected databases, if it has just started expanding the level. If it is in the middle of processing the level, it responds with a wait message, including an estimated completion time for processing the current level. Otherwise, if it is nearing the end of processing the level, it can simply refuse to share its load. The requesting process may choose another random process to request work from if receiving a neutral or negative message from a donor process. In their experiments, the authors show that DTPF achieves comparable or significantly lower run-times than STPF, demonstrating the utility of dynamically re-distributing work in parallel FSM.

In the context of mining closed patterns from symbol sequences, Cong et al. [10] proposed using pseudo-projections of the dataset $\mathcal{D}$ in Par-CSP, one for each found 1-length frequent sequences. The projection for the frequent 1-sequence $\langle d \rangle$ is made up of sequences in $\mathcal{D}$ containing d, minus any symbols before d in those sequences. Assuming each process has access to the input data file, a pseudo-projection is simply a set of file pointers noting the beginning of each sub-sequence included in the projection. Each process can then be assigned a sub-set of the 1-length frequent sequences and their associated pseudo-projections, which they mine using the BIDE [57] serial algorithm. Pseudo-projections are communicated to all processes via an all-to-all broadcast. Dynamic load balancing is then achieved through a master-worker scheme [17], where each process contacts the master process for another 1-length frequent sequence to process when they finish their current work. In an effort to ensure tasks in the work queue have similar sizes, the authors use a *selective sampling* scheme similar to that in Par-ASP [9] to sub-divide large tasks.

A key task in bioinformatics is the identification of frequently recurring patterns, called *motifs*, in long genomic sequences. Sequences are defined over a small alphabet, e.g. $\mathcal{A} = \{A, C, G, T\}$ for DNA sequences. Patterns are constrained by minimum and maximum length parameters, and are required to be highly similar, rather than identical, to count towards a frequently occurring pattern. Sahli et al. developed ACME [50], a parallel combinatorial motif miner that decomposes the problem into many fine-grained sub-tasks, dynamically schedules them to be executed by all available processes, and uses a cache aware search technique to perform each sub-task. They construct a full-text suffix-tree index of the sequence at each *worker process* in linear time and space, using Ukonnen's algorithm [55], and annotate each node in the tree with the number of leaves reachable through the node. Motif mining is then reduced to finding tree nodes that represent candidate patterns and summing up their annotated leaf count. The search space, represented as a trie data structure, is partitioned into prefix-coherent sub-tries by a *master process* s.t. the average number of sub-tries per worker process is at least 16. The authors show experimentally that this leads to a near-optimal workload balance in both distributed message passing and shared memory environments. A worker then requests a new prefix (sub-task) to search for when it finishes processing the current assignment.

5 Frequent Graph Mining

Successful algorithms for frequent graph mining (FGM) are often directly related to those designed for FIM. In this section, we provide a brief overview of some key serial methods for FGM and then discuss strategies and challenges for parallelizing FGM. For easy reference, Table 4 lists the serial and parallel methods described in this section.

5.1 Serial Frequent Graph Mining

Frequent graph mining comes with an added computational expense that is not present in frequent itemset or sequence mining. Determining if one graph is a subgraph of another is an NP-complete problem, known as the *subgraph isomorphism problem*. The cost of pruning infrequent candidates by performing isomorphism checks is very costly and thus most FGM methods make an effort to reduce the amount of pruning necessary.

Inokuchi et al. [25] developed Apriori-based Graph Mining (AGM), which extended the Apriori algorithm to FGM. During candidate generation, subgraphs are extended by joining two frequent subgraphs and expanding by one edge in all possible ways. Kuramochi and Karypis [29] described the Frequent Subgraph mining algorithm (FSG), another candidate generation based method. Size-k candidates are generated by joining two frequent $(k-1)$-subgraphs. Subgraphs are joined only if they share the same $(k-2)$-subgraph. When candidates are joined, an edge is *grown* by either connecting two existing vertices or adding a new vertex. Infrequent subgraphs must be pruned after all size-k candidates are generated. FSG optimizes the pruning stage by only adding edges during candidate generation which are known to be frequent. This can greatly reduce the number of infrequent subgraphs that need to be pruned.

Table 4: Serial and parallel frequent graph mining algorithms.

Type	Acronym	Name	Cite
ser	AGM	Apriori-based Graph Mining	[25]
ser	FSG	Frequent Subgraph Mining	[29]
ser	gSpan	Graph-based Substructure Pattern Mining	[60]
ser	HSIGRAM	Horizontal Single Graph Miner	[30]
ser	MoFa	Molecular Fragment Miner	[6]
ser	SUBDUE	Substructure Discovery	[24]
ser	VSIGRAM	Vertical Single Graph Miner	[30]
par	p-MoFa	Parallel MoFa	[37]
par	p-gSpan	Parallel gSpan	[37]
par	d-MoFa	Distributed MoFa with Dynamic Load Balancing	[14]
par	MotifMiner	MotifMiner Toolkit	[56]
par	MRFSE	MapReduce-based Frequent Subgraph Extraction	[35]
par	MRPF	MapReduce-based Pattern Finding	[34]
par	SP-SUBDUE	Static Partitioning SUBDUE	[11]
par	SP-SUBDUE-2	Static Partitioning SUBDUE 2	[47]

Pattern growth methods, which traverse the pattern lattice in depth-first order, have been quite successful in solving the FGM problem. They have smaller memory footprints than candidate generation ones, because only subgraphs located on the path from the start of the lattice to the current pattern being explored need to be kept in memory. A challenge that pattern growth methods must face, however, is the added risk of duplicate generation during exploration.

Yan and Han [60] developed the first pattern growth FGM method, named graph-based Substructure pattern mining (gSpan). It avoids duplicates by only expanding subtrees which lie on the *rightmost path* in the depth-first traversal. A major speedup in gSpan is obtained by searching for the next extensions to make at the same time as executing isomorphism tests.

Molecular Fragment miner (MoFa), introduced by Borgelt and Berthold [6], is a pattern-growth method that was designed specifically for mining molecular substructures, but is also capable of general FGM. MoFa maintains *embedding lists*, which are pointers to the exact locations within $\mathcal{G}$ where frequent subgraphs are found. Embedding lists trade off a large amount of memory for the need to perform isomorphism checks across $\mathcal{G}$ when a new subgraph is explored. To avoid duplicates, MoFa maintains a hash table of all subgraphs that have been expanded and checks whether a graph exists in the hash table before considering it as an extension. A graph G is hashed by first transforming it to a string representation unique to G and graphs isomorphic to G.

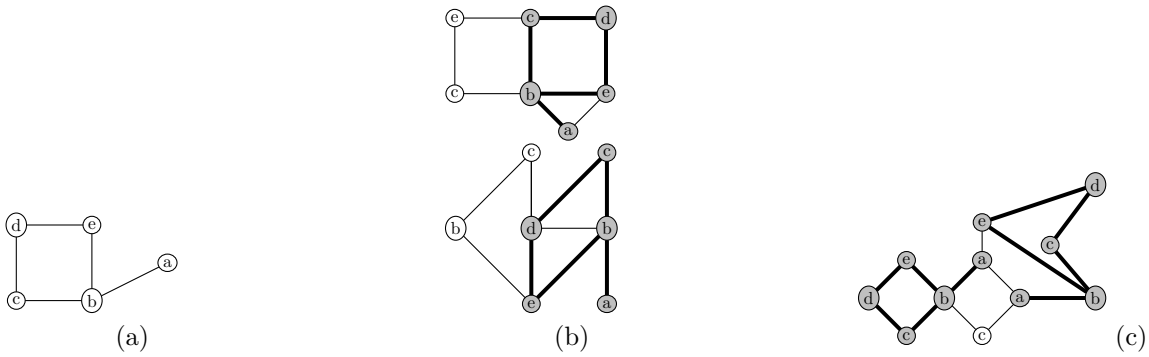

Figure 5: Example of the FGM problem using both a set of graphs and a large graph as input. The embedding of the candidate subgraph in (a) is highlighted by bold lines and gray nodes in the example databases in (b) and (c).

Some FGM algorithms define $\mathcal{G}$ as a single large graph, instead of the typical set of graphs. In this formulation, the support of a graph G is defined as the number of times G appears as a subgraph in $\mathcal{G}$. Figure 5 portrays the difference between the alternate problem definitions. A complication that appears when considering support in a single graph is that some embeddings of a subgraph G could be overlapping. Kuramochi and Karypis [30] show that, if overlapping subgraphs are counted, the problem is no longer downward closed. Thus, they only consider *edge-disjoint* embeddings, i.e., those embeddings of G that do not have any shared edges in $\mathcal{G}$.

SUBDUE, by Holder et al. [24], operates on a single graph and was one of the first developed FGM tools. It uses a greedy candidate generation approach and bounds the cost of doing isomorphism checks by doing inexact graph matches. Due to its greedy nature and inexact matches, SUBDUE is an approximate algorithm that does not find the complete set of frequent subgraphs. Kuramochi and Karypis [30] also addressed the scenario of $\mathcal{G}$ being a single graph. They presented two algorithms, Horizontal Single Graph Miner (HSIGRAM) and Vertical Single Graph Miner (VSIGRAM), which are candidate generation and pattern growth methods, respectively. HSIGRAM and VSIGRAM are complete FGM methods that also have the ability to do approximate counting. In order to reduce the time for frequency counting on each lattice level, HSIGRAM makes use of a modified form of embedding lists: instead of storing each embedding found in $\mathcal{G}$, a single edge called the *anchor-edge* is stored. This significantly reduces the memory expense of full embedding lists at the cost of some recomputation for each embedding.

5.2 Parallel Frequent Graph Mining

The exponential cost of subgraph isomorphism inherent in FGM makes parallelism for this problem vital for Big Data input. We now describe existing parallel FGM (PFGM) methods, focusing on how they address the issues of memory scalability, work partitioning, and dynamic load balancing.

5.2.1 Memory Scalability

Many of the PFIM and PFSM techniques for achieving memory scalability are also applicable in PFGM algorithms. Methods commonly require access to the entirety of $\mathcal{G}$. This has performance implications for both shared and distributed memory systems. Meinl et al. [37] created parallel versions of gSpan and MoFa for a shared memory system. In both algorithms, $\mathcal{G}$ is globally accessible to all processes, but could be split across multiple machines, causing unexpected network delays. Additionally, MoFa must be able to keep track of the already generated subgraphs in order to avoid generating duplicates.

Cook et al. [11] developed Static Partitioning SUBDUE (SP-SUBDUE), a parallel, message-passing implementation of SUBDUE. Like its serial predecessor, SP-SUBDUE expects $\mathcal{G}$ to be a single large graph. It uses a weighted graph partitioning algorithm [28] to divide $\mathcal{G}$ into as many partitions as there are processes. Each process can then store only their local portion of $\mathcal{G}$ in memory. The consequence of partitioning $\mathcal{G}$ is that any frequent subgraphs that exist across partition boundaries will be missed. The algorithm attempts to minimize these lost subgraphs by assigning frequent edges large weights before partitioning, leading to fewer cuts across them.

MapReduce offers a framework for developing parallel programs that operate gracefully using secondary non-volatile storage. Secondary storage devices, such as hard disk drives, are almost always larger than the amount of random access memory available on a machine. MapReduce-based Pattern Finding (MRPF) by Liu et al. [34] is a MapReduce framework for the FGM problem that assumes $\mathcal{G}$ is a single graph. It employs a candidate generation scheme that uses two iterations of MapReduce to explore each level of the search lattice. In the first iteration, the Map phase generates size-k candidates. The following Reduce phase detects duplicates and removes them. Finally, the second MapReduce pass computes the support of new candidates. Lu et al. [35] developed MapReduce based Frequent Subgraph Extraction (MRFSE), a candidate generation MapReduce implementation for the traditional many-graph $\mathcal{G}$. On iteration k, a Map task takes one graph from $\mathcal{G}$ and emits size-k candidates. Each Reduce task takes all of the appearances of a candidate and computes its support. MRFSE also stores embedding lists to maintain the location of all appearances of frequent subgraphs in $\mathcal{G}$. By doing so, isomorphism tests are no longer necessary, at the cost of additional memory usage.

5.2.2 Work Partitioning

Candidate generation methods consider a different set of candidates at each iteration. Much like the parallel Apriori methods seen in Section 3, the candidate generation process can easily be decomposed into parallel tasks. Wang and Parthasarathy developed the MotifMiner Toolkit for distributed memory systems [56]. MotifMiner is designed for both the case when $\mathcal{G}$ is a set of graphs and when $\mathcal{G}$ is a single graph. It has a similar structure as the Data Distribution method from Section 3. $\mathcal{G}$ is assumed to be available to all processes and each process is assigned some subset of candidate generation to perform. Once all processes have generated their local candidate sets, each process shares its local candidates with all others.

Many parallel candidate generation methods identify the individual candidates to expand as independent tasks. MRFSE, a MapReduce algorithm by Lu et al. [35], assigns instead each graph in $\mathcal{G}$ as a task. The authors noted that the simple assignment of the same number of graphs to each Map task is not a good work partitioning scheme because graphs in $\mathcal{G}$ can vary greatly in size. Instead, they count the edges of each graph in $\mathcal{G}$ and assign tasks such that each Map is assigned a roughly equal number of edges.

SP-SUBDUE partitions the single graph $\mathcal{G}$ across processes in a distributed setting. Each process then runs an independent instance of SUBDUE and support counts are finalized by broadcasting the frequent subgraphs to all other processes. When a process receives a new frequent subgraph from another partition, it computes its support on the local portion of $\mathcal{G}$. Ray and Holder [47] extended this work, proposing SP-SUBDUE-2, in which each process first searches their locally discovered subgraphs before doing expensive isomorphism tests across their partition of $\mathcal{G}$.

Parallelism for pattern growth methods is commonly achieved by exploiting the independent nature of the subtrees before expansion. One of the most straight-forward ways to partition work is to assign adjacent subtrees to processes and explore each one independently. This essentially operates as if a separate FGM program is being executed on each process. Meinl et al. [37] took this approach in their shared memory implementations of gSpan and MoFa.

Within the shared memory system context, Reinhardt and Karypis [48] created a parallel implementation of VSIGRAM, which works on a single large graph. Unlike SUBDUE and its parallel extensions, it is not an approximate algorithm. Their implementation features both coarse and fine-grained work partitioning: subgraphs can be extended vertically in parallel, and parallelism can also be exploited in the support count of a single subgraph.

5.2.3 Dynamic Load Balancing

Load balancing for FGM is a notoriously difficult problem. Not only is the search space highly irregular, but accurately estimating the workload of a task is very difficult. Di Fatta and Berthold [14] showed that the time required for subtree exploration in a biological dataset follows a power-law distribution.

Shared memory systems offer good opportunities for cheap dynamic load balancing. Work queues can be easily accessed by all processes and requesting or sending work is generally an inexpensive operation. Buehrer and Parthasarathy [8] experimented with several work queue designs inside of a shared-memory parallel implementation of gSpan. In their work, a task is defined as a tuple of the subgraph to be mined and the list of graphs in $\mathcal{G}$ that it occurs in. They evaluated three dynamic load balancing schemes: (1) a global work queue, in which all processes cooperate to enqueue and dequeue tasks; (2) hierarchical queues, in which each process maintains a local private queue but enqueues to a shared one if their local queue becomes full; and (3) distributed queues, in which each process maintains a private queue and offers any additional work to idle processes if the local queue becomes full. Their experiments showed that a distributed queueing model offered the most effective load balancing.

Meinl et al. [37] evaluated another scheme for dynamic load balancing in their shared memory implementations of MoFa and gSpan. Each process maintains a local last-in-first-out (LIFO) work queue that can be shared with other processes. When work needs to be distributed to another process, the stack is split in half. The authors identify a heuristic for evenly splitting work. Since the work queue is LIFO, the tasks at the bottom of the stack will often require more work because they were discovered at the beginning of execution and therefore will have more subtrees to explore. In an effort to provide a balanced work distribution, each process is assigned every-other task. The authors also compared sender-initiated and receiver-initiated load balancing and found that there was not a significant difference relative to the rest of the computation time.

Di Fatta and Berthold [14] created a distributed implementation of MoFa using message passing. Dynamic load balance is achieved by receiver-initiated work requests. When issuing a work request, a process must choose another worker to request from. The authors present a policy called *random-ranked polling*, in which each process maintains a list of all other processes, sorted by the time when their current work unit was started. The requesting process randomly selects from the list, with a bias towards those which have spent more time on the current work unit. A process being asked for work first heuristically determines whether they should split their work based on the stack size, the support, and branching factor of the pattern currently being tested. If they are able to provide work to the requesting process, a process sends one pattern from the bottom of their work stack. The requesting process is then responsible for re-generating the embedding list of the received pattern before processing it.

6 Conclusion

Many efficient serial algorithms have been developed for solving the frequent pattern mining problem. Yet they often do not scale to the type of data we are presented with today, the so-called "Big Data". In this chapter, we gave an overview of parallel approaches for solving the problem, looking both at the initially defined frequent itemset mining problem and at its extension to the sequence and graph mining domains. We identified three areas as key challenges to parallel algorithmic design in the context of frequent pattern mining: memory scalability, work partitioning, and load balancing. With these challenges as a frame of reference, we extracted key algorithmic design patterns from the wealth of research conducted in this domain. We found that, among parallel candidate generation based algorithms, memory scalability is often the most difficult obstacle to overcome, while for those parallel algorithms based on pattern growth methods, load balance is typically the most critical consideration for efficient parallel execution.

The parallel pattern mining problem is in no way "solved". Many of the methods presented here are more than a decade old and were designed for parallel architectures very different than those that exist today. Moreover, they were not evaluated on datasets big enough to show scalability to Big Data levels. While most works included limited scalability studies, they generally did not compare their results against other existing parallel algorithms for the same problem, even those designed for the same architecture. More research is needed to validate existing methods at the Big Data scale.

Work partitioning and load balancing continue to be open problems for parallel frequent pattern mining. Better methods to estimate the cost of solving sub-problems at each process can lessen the need for dynamic load balancing and improve overall efficiency. Additionally, they can help processes intelligently decide whether to split their work with idling ones or not. Another open problem is that of mining sub-patterns in a large object, where sub-patterns can span multiple process' data. Current methods for sequence motif mining and frequent subgraph mining in a large graph either rely on maximum pattern length constraints that allow each process to store overlapping data partition boundaries or transfer data partitions amongst all processes during each iteration of the algorithm. Neither solution scales when presented with Big Data, calling for efficient methods to solve this problem exactly.

References

[1] Big data meets big data analytics. `http://www.sas.com/resources/whitepaper/wp_46345.pdf`. Accessed: 2014-03-06.

[2] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, March 2001.

[3] Rakesh Agrawal and John C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–969, 1996.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[6] Christian Borgelt and Michael R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM 2002, pages 51–58. IEEE, 2002.

[7] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[8] Gregory Buehrer, Srinivasan Parthasarathy, Anthony Nguyen, Daehyun Kim, Yen-Kuang Chen, and Pradeep Dubey. Parallel graph mining on shared memory architectures. Technical report, The Ohio State University, Columbus, OH, USA, 2005.

[9] Shengnan Cong, Jiawei Han, Jay Hoeflinger, and David Padua. A sampling-based framework for parallel data mining. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 255–265, New York, NY, USA, 2005. ACM.

[10] Shengnan Cong, Jiawei Han, and David Padua. Parallel mining of closed sequential patterns. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 562–567, New York, NY, USA, 2005. ACM.

[11] Diane J Cook, Lawrence B Holder, Gehad Galal, and Ron Maglothin. Approaches to parallel graph-based knowledge discovery. *Journal of Parallel and Distributed Computing*, 61(3):427–446, 2001.

[12] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, Cambridge, 1990.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[14] Giuseppe Di Fatta and Michael R. Berthold. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):773–785, 2006.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[16] Carole A. Goble and David De Roure. The impact of workflow tools on data-centric research. In Tony Hey, Stewart Tansley, and Kristin M. Tolle, editors, *The Fourth Paradigm*, pages 137–145. Microsoft Research, 2009.

[17] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, second edition, 2003.

[18] Valerie Guralnik and George Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Computing*, 30(4):443–472, April 2004.

[19] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 277–288, New York, NY, USA, 1997. ACM.

[20] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: Current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, August 2007.

[21] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 355–359, New York, NY, USA, 2000. ACM.

[22] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 1–12, New York, NY, USA, 2000. ACM.

[23] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the third International Joint Conference on Artificial intelligence*, IJCAI-73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[24] Lawrence B Holder, Diane J Cook, Surnjani Djoko, et al. Substucture discovery in the subdue system. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, KDD-94, pages 169–180, 1994.

[25] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer, 2000.

[26] Mahesh V. Joshi, George Karypis, and Vipin Kumar. A universal formulation of sequential patterns. Technical Report 99-021, Department of Computer Science, University of Minnesota, 1999.

[27] Mahesh V. Joshi, George Karypis, and Vipin Kumar. Parallel algorithms for mining sequential associations: Issues and challenges. Technical report, Department of Computer Science, University of Minnesota, 2000.

[28] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, Dec 1998.

[29] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, ICDM 2001, pages 313–320. IEEE, 2001.

[30] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[31] Vance Chiang-Chi Liao and Ming-Syan Chen. Dfsp: a depth-first spelling algorithm for sequential pattern mining of biological sequences. *Knowledge and Information Systems*, pages 1–17, 2013.

[32] Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter*, 14(2):6–19, 2013.

[33] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In *Proceedings of the Sixth International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 76:1–76:8, New York, NY, USA, 2012. ACM.

[34] Yang Liu, Xiaohong Jiang, Huajun Chen, Jun Ma, and Xiangyu Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In *Advanced Parallel Processing Technologies*, pages 341–355. Springer, 2009.

[35] Wei Lu, Gang Chen, Anthony KH Tung, and Feng Zhao. Efficiently extracting frequent subgraphs using mapreduce. In *2013 IEEE International Conference on Big Data*, pages 639–647. IEEE, 2013.

[36] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, January 1997.

[37] Thorsten Meinl, Marc Worlein, Ingrid Fischer, and Michael Philippsen. Mining molecular datasets on symmetric multiprocessor systems. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 2 of *SMC '06*, pages 1269–1274. IEEE, 2006.

[38] Iris Miliaraki, Klaus Berberich, Rainer Gemulla, and Spyros Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 797–808, New York, NY, USA, 2013. ACM.

[39] Sandy Moens, Emin Aksehirli, and Bart Goethals. Frequent itemset mining for big data. In *2013 IEEE International Conference on Big Data*, pages 111–118. IEEE, 2013.

[40] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical report, University of Maryland at College Park, College Park, MD, USA, 1995.

[41] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, 1998.

[42] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE '01, pages 215–224, Washington, DC, USA, 2001. IEEE Computer Society.

[43] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, and Hua Zhu. Mining access patterns efficiently from web logs. In *Proceedings of the Fourth Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications*, PAKDD '00, pages 396–407, London, UK, UK, 2000. Springer-Verlag.

[44] Shaojie Qiao, Tianrui Li, Jing Peng, and Jiangtao Qiu. Parallel sequential pattern mining of massive trajectory data. *International Journal of Computational Intelligence Systems*, 3(3):343–356, 2010.

[45] Shaojie Qiao, Changjie Tang, Shucheng Dai, Mingfang Zhu, Jing Peng, Hongjun Li, and Yungchang Ku. Partspan: Parallel sequence mining of trajectory patterns. In *Proceedings of the 2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery - Volume 05*, FSKD '08, pages 363–367, Washington, DC, USA, 2008. IEEE Computer Society.

[46] A. Rajimol and G. Raju. Web access pattern mining — a survey. In *Proceedings of the Second International Conference on Data Engineering and Management*, ICDEM '10, pages 24–31, Berlin, Heidelberg, 2012. Springer-Verlag.

[47] Abhik Ray and Lawrence B. Holder. Efficiency improvements for parallel subgraph miners. In *Florida Artificial Intelligence Research Society Conference*, FLAIRS '12, 2012.

[48] Steve Reinhardt and George Karypis. A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph. In *International Symposium on Parallel and Distributed Processing*, IPDPS 2007, pages 1–8, 2007.

[49] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: The teiresias algorithm. *Bioinformatics*, 14(1):55–67, 1998.

[50] Majed Sahli, Essam Mansour, and Panos Kalnis. Parallel motif extraction from very long sequences. In *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 549–558, New York, NY, USA, 2013. ACM.

[51] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases*, VLDB '95, pages 432–444, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[52] Takahiko Shintani and Masaru Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Fourth International Conference on Parallel and Distributed Information Systems*, pages 19–30, Dec 1996.

[53] Takahiko Shintani and Masaru Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In Xindong Wu, Kotagiri Ramamohanarao, and Kevin B. Korb, editors, *Proceedings of the Second Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 1394 of *PAKDD '98*, pages 283–294. Springer, 1998.

[54] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the Fifth International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, pages 3–17, London, UK, UK, 1996. Springer-Verlag.

[55] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[56] Chao Wang and Srinivasan Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 31–40, New York, NY, USA, 2004. ACM.

[57] Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 79–91, Washington, DC, USA, 2004. IEEE Computer Society.

[58] Ke Wang, Yabo Xu, and Jeffrey Xu Yu. Scalable sequential pattern mining for biological sequences. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, pages 178–187, New York, NY, USA, 2004. ACM.

[59] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[60] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM 2002, pages 721–724. IEEE, 2002.

[61] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large databases. In Daniel Barbará and Chandrika Kamath, editors, *Proceedings of the SIAM International Conference on Data Mining*, SDM 2003. SIAM, 2003.

[62] Mohammed J. Zaki. Efficient enumeration of frequent sequences. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, CIKM '98, pages 68–75, New York, NY, USA, 1998. ACM.

[63] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May 2000.

[64] Mohammed J. Zaki. Parallel sequence mining on shared-memory machines. *Journal of Parallel and Distributed Computing*, 61(3):401–426, Mar 2001. Special issue on High Performance Data Mining.

[65] Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1-2):31–60, January 2001.

[66] Mohammed J. Zaki, Mitsunori Ogihara, Srinivasan Parthasarathy, and Wei Li. Parallel data mining for association rules on shared-memory multi-processors. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pages 43–43, 1996.

[67] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.