# Deep Parallelization of Parallel FP-Growth Using Parent-Child MapReduce

Adetokunbo Makanju, Zahra Farzanyar, Aijun An, Nick Cercone
*EECS Department*
*York University*
*Toronto, Canada*
*Email: tokunbo,zfarzan,aan,nick@cse.yorku.ca*

Zane Zhenhua Hu, Yonggang Hu
*lBM Canada*
*Markham, Canada*
*Email: zane, yhu@ca.ibm.com*

***Abstract*—MapReduce is an important programming model for processing in distributed environments. Compared to other distributed programming models, MapReduce reduces communication overheads between computers and improves fault tolerance. However, the MapReduce model does not allow for automatic synchronization between jobs. A large number of data analytics algorithms use a recursive divide-and-conquer approach, which inherently allows for parallelism at each level of recursion. However, it is often difficult to parallelize such algorithms using the traditional MapReduce model if the process requires synchronization.**

**In this paper we introduce Parent-Child MapReduce, a version of the MapReduce programming model that allows for MapReduce tasks to be created dynamically and synchronized in a hierarchical parent-child fashion. Using the Parallel FP-Growth (PFP) algorithm for mining frequent patterns as a reference, we show that Parent-Child MapReduce can be used to parallelize recursive divide-and-conquer algorithms using the MapReduce model and that this can lead to significant speed ups in the computational speed of such algorithms. Our evaluation shows that we can achieve 68% (or 3 times) performance gain when used with PFP.**

***Keywords*-MapReduce; Frequent Pattern Mining; PFP; FP-Growth; Recursive Divide and Conquer**

## I. Introduction

Frequent Pattern Mining (FPM) is a fundamental problem in big data analytics. FPM works on a database of transactions (events). Each transaction has a number of non-repeated items i.e. discrete entities, associated with it. The goal is to find the co-occurrence relationships between these items across the transactions in the database. FPM has broad applications in data analytics for the retail, finance, health-care, telecommunications and transport industries. Information about frequent patterns can be used as the basis for activities such as promotional pricing or product recommendations.

A large number of algorithms have been proposed for the FPM problem [1] but by far the most popular algorithms are Apriori [2], Eclat[3] and FP-Growth[4]. One of the major problems in FPM is the potentially large size of the transaction database. The entire transaction database usually needs to fit into memory. In this wise, FP-Growth performs better due to its use of the FP-Tree data structure that compresses the transaction database [4]. Parallel FP-Growth (PFP) is a MapReduce-based extension of FP-Growth that achieves faster computation times in distributed environments and provides as output the Top-K frequent patterns for each of the items in the database[5].

PFP uses the MapReduce model, while FP-Growth is a recursive divide-and-conquer algorithm. This means that FP-Growth inherently allows parallelism in each level of recursion. Despite this fact, PFP avoids parallelizing the recursive function calls of FP-Growth. It achieves its parallelization by partitioning the database into independent shards that can be processed as single-node instances of FP-Growth. This was probably done as it is difficult (or impossible) to parallelize the recursive sub-tasks using the MapReduce framework. This is due, primarily, to the lack of automatic synchronization mechanism for the parallel sub-tasks in the MapReduce framework, which is needed for aggregating the results from all subtasks.

This limits the level of parallelization that can be achieved by PFP. This limitation in PFP's ability to parallelize becomes evident in cases when the amount of computation is driven not by the size of the dataset but the characteristics of the data. A significant combinatorial explosion in the number of recursive function calls is experienced when FP-Growth is run using low support values on datasets that are sparse, i.e. containing a large number of items relative to the number of transactions, and containing very long transactions. For any item-set of length $x$ in a transaction database , the number of possible frequent patterns that it posses is bounded by $2^x$, in the worst case. Partitioning the database does not remove the characteristics of the data because at least one partition of the dataset will retain these characteristics and slow down the computation. Low support frequent patterns are important in certain applications e.g. web search [5]. Data collected from the web can at times be sparse and contain long transactions, as the datasets used in our work show. It is therefore desirable to remove this limitation of PFP.

In this paper, we attack this problem by extending the level of parallelization that can be achieved by PFP, using the Parent-Child MapReduce feature of IBM Platform Symphony [6]. Parent-Child MapReduce is a feature of IBM Platform Symphony that allows for MapReduce tasks to be created dynamically and synchronized in a hierarchical fashion. Parent-Child MapReduce allows the processing of the shards of the database to be parallelized by FP-Growth when necessary. The implementations of MapReduce in platforms like Apache Hadoop and Apache Spark cannot do this as they are based on the traditional method [7], [8]. We refer to our proposed algorithm as Recursive-PFP (R-PFP). We also tackle some of the challenges that can be encountered when attempting to parallelize the recursive calls of FP-Growth using MapReduce. These challenges include predicting the processing loads of FP-Trees, limiting the amount of data that is written to file for the MapReduce tasks to process and achieving a balance between the number of parallel tasks generated without creating a new computational challenge.

Our initial evaluations on two publicly available datasets indicate that R-PFP, in the right conditions, can provide significant speed-ups over PFP.

The contributions of the this paper are as follows:

First, we introduce Parent-Child MapReduce, a version of the MapReduce programming model that allows MapReduce tasks to be created dynamically and recursively. The tasks are created with a hierarchical parent-child relationship between them. Second, we show that Parent-Child MapReduce can be used to parallelize recursive divide-and-conquer algorithms using the MapReduce model. Third, using PFP as reference, we improve the level of parallelization and create a new version of PFP, which we call R-PFP that uses Parent-Child MapReduce. R-PFP is able to reduce the computational time by approximately $68\%$ (or 3 times) over PFP in situations where significant increase in processing time occurs and the parallelization (depth-threshold) parameter is set appropriately.

## II. Related Work

In this section we describe some related work involving balancing the load of processing transaction databases in PFP and parallelizing FP-Growth. We note that PFP has three phases, which we will describe in detail in Section. III-B. The Reduce phase of the second step is responsible for mining frequent patterns and is equivalent to single node instance of the FP-Growth algorithm. So an attempt to increase the parallelization of PFP is similar to the goal of parallelizing FP-Growth.

In [9], the authors propose BPFP (Balanced Parallel FP-Growth). This work extends PFP by including a mechanism to ensure that work load assignment in the Reduce phase of PFP's second step is balanced among the reducers. This approach however cannot solve the problem of having a single FP-Tree with a huge load factor, caused by the dataset characteristics, as the processing of that single FP-Tree cannot be distributed and the problem persists irrespective of what group the FP-Tree is assigned to.

In [10] the authors introduce a new algorithm called MLFPT (Multiple Local Frequent Pattern Tree). Much like FP-Growth the algorithm works in two steps, except that it constructs multiple FP-Trees in its first phase. However, these trees cannot be produced independently, so shared global counters are needed to ensure that only globally frequent patterns are mined from each tree. This is a drawback of this approach.

In [11], the authors also parallelize FP-Growth in a PC cluster environment. It achieves this by independently processing the recursive function calls used by FP-Growth.This work does not partition the database but improves on the work done in [10], as it is a Shared-Nothing algorithm. Their approach does not use MapReduce. MapReduce based approach reduces communication overheads between computers and enables the process recover more easily when node failures occur. For the same reason it also cannot be used in conjunction with PFP, which is MapReduce based.

## III. Problem and Algorithm Descriptions

In this section we describe both the FP-Growth and PFP algorithms in detail. We start by giving a proper definition of the FPM problem.

Let $I = \{i_1, i_2, i_3, ..., i_n\}$ be a **set of items**. We define a **transaction** $T$ as any subset of $I$ and a **transaction database** $D = \langle T_1, T_2, T_3, ..., T_n \rangle$ , where $T_i$ is a transaction, as a collection of transactions. Let us also define a **pattern** $P$ as a subset of $I$, the **support** of $P$ in $D$ is the percentage of the transactions in $D$ for which $P$ is a subset. A **frequent pattern** is a pattern whose support is no less than a user-specified threshold value called the **minimum support threshold**, $\epsilon$.

With the definitions above the FPM problem can be defined as follows: Given a database of transactions *D* and a user provided minimum support threshold $\epsilon$, find the set of frequent patterns in *D*.

### A. FP-Growth

The FP-Growth algorithm has two phases.These are

1) Construction of the FP-Tree
2) Discovery of frequent patterns in the FP-Tree

A complete discussion of how to build an FP-Tree is beyond the scope of this paper. However, given the example transaction database in Table I, the resulting FP-tree will look like the diagram in Fig.1. In the third column of Table I, the items are ordered in decreasing order of their support. The FP-Tree is made of two parts, the *Header table* and a *prefix tree*. Each node in the prefix tree except the *root* can have three items. These are (1) an item name, (2) an item count that represents the number of transactions that contain the item in that path of the tree and (3) a link to the next node in the tree with the same item name, if such a node exists. The entries in the header table are sorted in the order of decreasing support and each one consists of an item name and link to the first occurrence of the item in the prefix tree.

Once the initial FP-Tree is constructed the frequent patterns can be extracted using Algorithm 1 (*FP-Growth*). A

Table I: An example transaction database with ordered frequent items. assuming the support threshold is 60%

| TID | Items | Ordered Frequent Items |
|---|---|---|
| 1 | *f,a,c,d,,g,i,m,p* | *f,c,a,m,p* |
| 2 | *a,b,c,f,l,m,o* | *f,c,a,b,m* |
| 3 | *b,f,h,j,o* | *f,b* |
| 4 | *b,c,k,s,p* | *c,b,p* |
| 5 | *a,f,c,e,l,pm,n* | *f,c,a,m,p* |

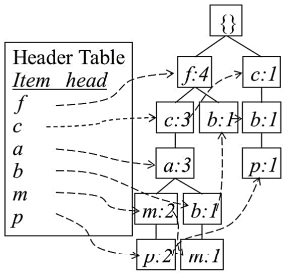


Figure 1: A sample FP-Tree built using the transaction database in Table I and a support threshold of 60% [4].

**Algorithm 1** FP-Growth Algorithm

```
1: Input: FP Tree $Tree$, Support threshold $\epsilon$
2: Output:Set of frequent patterns
3: Procedure: $FPGrowth(Tree, \epsilon, \alpha)$ [$\alpha$ is a frequent pattern and its intial value is nul]
4: if $Tree$ contains a single path then
5:     Generate all possible combinations of the path
6: else
7:     for all items $i_n$ in the headerTable of $Tree$ do
8:         Create pattern $\beta = i_n \cup \alpha$
9:         Construct $\beta$ conditional pattern base [The conditional pattern base is a projection of the transactions in $Tree$ that contains only paths the preceed nodes with item name $i_n$ ]
10:        Construct conditional tree $Tree_\beta$ from the conditional pattern base
11:        if $T_\beta \neq null$ then
12:            Call $FPGrowth(Tree_\beta, \epsilon, \beta)$
13:        end if
14:    end for
15: end if
```

complete discussion of the FP-Growth algorithm is beyond the scope of this paper. The reader can find more details on FP-Growth in the original paper by Han et. al. [4]. The algorithm uses a recursive divide-and-conquer strategy to explore the entire space of frequent patterns. For instance, if the FP-Tree example in Fig. 1 were to be processed using Algorithm 1, then the first recursive call to *FPGrowth()* on line 12 within the for-loop starting on line 7 would partition the search space as follows: (1) Patterns containing $p$, (2) Patterns containing $m$, but no $p$, (3) Patterns containing $b$, but neither $p$ nor $m$, (4) Patterns containing $a$, but neither $p$,$m$ nor $b$, (5) Patterns containing $c$, but neither $p$,$m$,$b$ nor $a$, and (6) Patterns containing $f$, but neither $p$,$m$,$b$,$a$ nor $c$.

Each call to *FPGrowth()* is independent of the other as the area of the search space that each call deals with do not overlap. This fact allows for each of these calls to *FPGrowth()* to be parallelized. With each subsequent recursive call to *FPGrowth()*, each of these search spaces would also be partitioned. For instance, the first call to grow patterns containing $p$ will be further partitioned within the for loop on line 7. This process continues until the conditional tree is empty, which indicates that there are no more partitions of the search space.

## B. PFP: Parallel FP-Growth

PFP uses three MapReduce steps to parallelize FP-Growth. These steps are summarized below.

**Step 1 - Parallel Counting:** Using MapReduce, these steps generates what is called an *F-List*. The *F-List* is a list of all the frequent items in the transaction database. All of the items are then divided into $Q$ groups. This list of $Q$ groups is called a *G-List*.

**Step 2 - Parallel FP-Growth:** This step of PFP is the most time consuming. The Map and Reduce phases of this step carry out different functions, so each one is described below.

*Mapper* - At this step each mapper is assigned a shard of the original database. The mapper determines to what group(s) each transaction in the shard is relevant. They then generate key/value pairs, where each key is a group-id and the value is the relevant transaction. One or more key/value pairs maybe generated per transaction, as the transaction maybe relevant to multiple groups.

*Reducer* - The MapReduce infrastructure automatically groups the output from the map phase based on their key values. A group of output values with the same key constitutes a shard of the original database that contains all transactions that are relevant to the items in that group. These are called *group-dependent shards*. Each reducer is assigned a number of group dependent shards to process. For each shard, an independent local FP-Tree is constructed and mined for only the frequent patterns containing the items in the group, by a single node call to Algorithm 1, the original sequential *FPGrowth()* algorithm. The processing of each shard is equivalent to a single node run of FP-Growth as described in Section III-A. No parallelism is taking place in the reducer.

**Step 3 - Aggregation:** Using MapReduce this step of the algorithm takes the output of the previous phase and puts them together as the final output of the process. The output is the Top-K frequent patterns for each item in the *F-List*.

The algorithm for the reduce phase of Step-2 is given in Algorithm 2. In this algorithm the *G-List* is the same as that created in Step-1 of the algorithm. The keys in the $\langle key, value\rangle$ pairs processed by the reducer are unique ids that identify each group of items, while the values are the transactions that are relevant to the group. The reducer users generates a $localFlist$ and a $LocalFPTree$ for the group. The $localFlist$ is a list of all the items in the group. It then proceeds to generate the frequent patterns that are relevant to each item in the $localFlist$ with a call to *FPGrowth* (Algorithm 1). Since PFP only outputs the Top-K frequent patterns for each item in the group, a max heap is used to store the frequent patterns. The key in the output $\langle key, value\rangle$ pairs of the reducer is an item from the $localFlist$ and a max heap of all the frequent patterns that contain the item.

The Reduce phase of Step-2 can constitute a bottleneck for PFP in some cases. If PFP is run with a low support threshold, a Parallel FP-Growth Reduce task that is assigned a database shard with long transactions will take considerable amount of time to complete. In its current form PFP offers no way to distribute the processing of such reducers. Indeed, our experiments showed that increasing the number of reducers or the number of groups $Q$ can not fix the problem because the reducer that was assigned the shard containing long transactions always lagged behind. For more details on PFP, please see [5].

**Algorithm 2** Parallel FP-Growth Reducer Algorithm

```
1: **Input:** Support threshold $\epsilon$, HeapSize $k$
2: **Procedure:** Reducer($\epsilon$,$k$)
3: Load G-List
4: **for all** $key = gid, value = DB_{gid}$ pairs **do**
5:     localFlist ← G-List$_{gid}$
6:     LocalFPTree ← BuildFPTree($DB_{gid}$)
7:     **for all** items $f$ in localFlist **do**
8:         Create $MaxHeap$ of size $k$
9:         $Maxheap.add(FPGrowth(LocalFPTree, \epsilon, f))$
10:        Call Output($\langle f, MaxHeap\rangle$)
11:    **end for**
12: **end for**
```

### C. Mahout Implementation of PFP

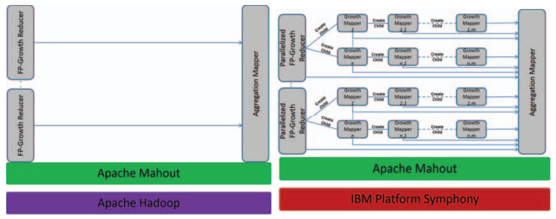
Apache Mahout [12] is an open-source code repository of scalable implementations of machine learning and data mining algorithms. The implementations in the repository run on such platforms as Apache Hadoop[7], Apache Spark [8] and Apache Flink [13]. Mahout has an implementation of PFP built to run on the Hadoop platform in its 0.9 release. We used and modified this implementation of PFP in our work.

It should be noted that Mahout's implementation of PFP includes FP-Bonsai tree pruning [14]. FP-Bonsai integrates the Ex-Ante data reduction technique into FP-Growth [15]. This technique can help to mitigate the problem of fitting large transaction databases into memory by pruning conditional FP-Trees for redundant information.

Aside from the difference noted above Mahout's implementation of PFP is based on the description of PFP described in the preceding paragraph. So it inherits the same limitations.

## IV. Improvements to PFP

In this section we describe our algorithm for parallelizing the reduce phase of Step 2 of PFP (Section III-B) using the Parent-Child MapReduce feature of IBM Platform Symphony. Our algorithm is called Recursive-PFP or R-PFP in brief. R-PFP retains the Step-1, Step-3 and the map phase of Step 2 of the original PFP algorithm. The only change is in the reduce phase of Step-2. We provide a side-by-side comparison of how the reduce phase of Step 2 works for original



(a) PFP Pattern Growth (b) R-PFP Pattern Growth

Figure 2: Comparing the architectures of Parallel FP-Growth(a) and Recursive-PFP (b).

PFP and our proposed method for growing frequent patterns in Fig. 2. In PFP each FP-Growth reducer in Step 2 grows its frequent patterns using non-parallelized Algorithm 2 and passes its output to a Aggregation mapper, which collates the results from all of the Parallel FP-Growth reducer tasks. However, for R-PFP, we replace Apache Hadoop with IBM Platform Symphony and add a new process, which we call $GrowthMapper$ in Step 2. The $GrowthMapper$ enables us to use MapReduce to parallelize the recursive calls to *FPGrowth()* (Algorithm 1), which takes place in the Reduce phase of Step 2.

Each of the *Growth Mappers* is a task in a MapReduce job that has no *Reduce* phase. It takes as input $\langle key, value\rangle$ pairs of the form $\langle f, Obj\rangle$, where $f$ is the item from the G-List whose frequent patterns are being mined and $Obj$ is an object containing all the parameters required for the *FPGrowth()* call like the minimum support threshold etc. With this input each mapper finds all the frequent patterns containing item $f$ but not other items, depending on the search space it takes care of.

### A. IBM Platform Symphony and Parent-Child MapReduce

IBM Platform Symphony is a software product that provides parallel computing and application grid management for compute-intensive and data-intensive applications across heterogeneous IT resources. It automatically optimizes multi-user and multi-tenant resource sharing and workload scheduling for various applications including SOA (Service-Oriented Applications), High Performance Computing (HPC), batch, Apache Hadoop MapReduce, Apache Spark, Web and database applications.

It supports parent-child workloads in SOA, which allows an application program to submit parent-child jobs and tasks to a scheduler in the system. A job can have one or more tasks that can be scheduled by the scheduler to run on resource slots in parallel, one task per slot. This feature is adapted for Apache Hadoop MapReduce jobs in this work, hence the term parent-child Mapreduce.

If a running task in a job finds it has much more to do than expected, the task can split its work, and submit one or more child jobs, each child job can contain one or more child tasks. Symphony will track the dependencies of parent-child jobs and tasks, schedule the child tasks in child jobs to run on multiple resource slots to speed up the computation, and manage the life cycle of a parent job so that the parent job can be finished only if its own tasks and all its child jobs and tasks are finished.

This allows a parent job to know when its child jobs finish and signal the next MapReduce step (e.g. Step-3 in the PFP algorithm) to start. This achieves the synchronization needed for the final aggregation step of PFP. Details of how the parent-child feature is realized in IBM Platform Symphony MapReduce is beyond the scope of this paper. We focus on how to use this feature to parallelize recursive divide-and-conquer tasks that need synchronization.

### B. Recursive-PFP ( R-PFP)

Parallelizing the *Reduce* phase of the *Parallel FP-Growth* step of PFP comes with unique challenges. The first challenge is the fact that the recursive process of FP-Growth proceeds in a depth-first search (DFS) fashion. A DFS is an inherently sequential process [16], therefore any attempt to parallelize can only be done dynamically during the DFS traversal. This creates a problem for MapReduce parallelization, which requires that all tasks be defined at the start of processing. In this regard, the Parent-Child MapReduce framework overcomes this limitation as it allows MapReduce tasks to be created dynamically.

The second challenge is the need to create a delicate balance between speed and the number of tasks created. Technically, a single task can be created to process any call to *FPGrowth()*. This approach ensures that no time is wasted before processing. However, just as is the case with writing data, this approach can lead to the creation of billions of tasks. On the other hand, collecting all calls and processing all in a single task will require waiting until the completion of the FP-Growth Reduce task before the child task can be created. Therefore, a balance must be created between both extremes. This issue is addressed in our solution by allowing only the top-level reduce tasks to create as many child tasks as needed. Subsequent child reduce tasks are limited to creating only a single child.

The third challenge is the amount of data that needs to be written. Each record of the child MapReduce task is a *FPGrowth()* function call (see Algorithm 5). Therefore all of the parameters of the function call need to be written to disk. In the worst case, the number of function calls required to process an FP-Tree is a combinatorial factor of the number of items in the header table. We could therefore be dealing with billions of records when dealing with an FP-Tree with a header table size greater than 13. The conditional FP-Tree is one of the parameters required during a call to *FPGrowth()*. Writing a billion conditional FP-Trees to disk will be counter productive. This issue is addressed in our solution by not serializing conditional FP-Trees and providing a mechanism for conditional FP-Trees to be generated dynamically when required.

The last challenge is the need to synchronize tasks. PFP's three phases are sequential, so the MapReduce tasks for each of these phases need to be synchronized to start one after another. Traditionally, MapReduce tasks are not meant to communicate with each other, any synchronization that occurs must be hard-coded by the programmer. This is another case where Parent-Child MapReduce helps. Since we can create tasks that are tied to a parent job, we can ensure that the parent job does not complete until all of its children have finished.

The algorithm for the *Reduce* phase of the *Parallel FP-Growth* step of R-PFP can be found in Algorithm 3. It works in fashion similar to Algorithm 2 but takes a new input parameter, depth threshold $\delta$. We use this parameter to determine which FP-Trees will be processed in parallel. We include a discussion on how and why we use this parameter as load predictor in Section IV-C. The main idea of Algorithm 3 is that in the recursive FP-Growth process, if the depth of a local (conditional) FP-tree is greater than $\delta$, it creates a child job by calling GrowthMapper (Algorithm 5) to process the FP-tree in parallel. Any FP-Tree that does not meet this criterion is processed sequentially to completion. Algorithm 3 does not write a single key/value pair to the input file for the child job. It writes as many as $|localFList| \times HeaderTableSize$ key/value pairs to the file. These records represent all the independent top-level function calls that would have been made sequentially if the tree had been processed directly. These calls can now be processed in parallel by the $GrowthMapper$ tasks in the child job. The second challenge mentioned above is addressed inAlgorithm 3, which creates child tasks for top-level FP-Trees and in Algorithm 5, which creates child tasks for conditional FP-Trees.

**Algorithm 3** Parent-Child FP-Growth Reducer Algorithm

```
1: Input:Depth threshold $\delta$, Support threshold $\epsilon$,
2: HeapSize $k$
3: Procedure: Reducer($\delta$,$\epsilon$,$k$)
4: Load G-List
5: for all $key = gid, value = DB_{gid}$ pairs do
6:     localFlist ← G-List$_{gid}$
7:     LocalFPTree ← BuildFPTree($DB_{gid}$)
8:     if LocalFPTree.depth $\geq \delta$ then
9:         LocalFPTree.Write() [Write FP-Tree to disk]
10:        for all items $f$ in localFlist do
11:            for all items $l$ in the headerTable of LocalFPTree do
12:                itemlist ← newList()
13:                itemlist.add($l$)
14:                Obj ← newGrowthObject(itemlist)
15:                Call WriteChildRecords($f$, $Obj$)
16:            end for
17:        end for
18:        Call GrowthMapper($\delta$,$\epsilon$,$k$)
19:    else
20:        for all items $f$ in localFlist do
21:            Create $MaxHeap$ of size $k$
22:            $Maxheap.add(FPGrowth(LocalFPTree, \epsilon, f))$
23:            Call Output($\langle f, MaxHeap \rangle$)
24:        end for
25:    end if
26: end for
```

The parameters that will be grouped into $Obj$ (Line 14 of

Algorithm 3) would differ between actual implementations. However, we note that one of the object member is a list of items, $itemlist$. This list of items will contain a sequential list of the all items that need to be used to generate the conditional tree required for the call. In order to save time required to write conditional FP-Trees for each record, we write the Top-Level tree only once, so that each $GrowthMapper$ task can read this FP-Tree into memory once and then subsequently generate the conditional tree required for the call using the items in the list in sequence (Algorithm 4). As the $GrowthMapper$ (Algorithm 5) is able to generate its own children, any $Obj$ instance created will just need to add the new item to the end of the previous list of items, for its child job to use to generate its own conditional trees using the same FP-Tree that was written to the disk by the Parent task. This design addresses the third challenge raised previously.

**Algorithm 4** Build Conditional Tree Using a List

```
1: Procedure: buildWithList($Tree, itemList, \epsilon$)
2: conditionalTree = newFPTree()
3: a $\leftarrow$ itemList.getFirst()
4: buildCondTree($Tree, ConditionalTree, a, \epsilon$)
5: for i = 1, to i= len(itemList) do
6: [First item in itemList is at index 0]
7:     tempTree $\leftarrow$ conditionalTree
8:     conditionalTree = newFPTree()
9:     buildCondTree($tempTree, ConditionalTree, itemList(i), \epsilon$)
10: end for
11: return conditionalTree
```

**Algorithm 5** Growth Mapper Algorithm

```
1: Input:Depth threshold $\delta$, Support threshold $\epsilon$,
2: HeapSize $k$
3: Procedure: GrowthMapper($\delta$,$\epsilon$,$k$)
4: Load FP-Tree $Tree$ from file
5: for all $key, value = Obj$ pairs do
6:     if This is the first record then
7:         Create $MaxHeap$ of size $k$
8:         $currentAttribute = key$
9:     else
10:        if $key \neq currentAttribute$ then
11:            Call Output($\langle currentAttribute, MaxHeap \rangle$)
12:        end if
13:    end if
14:    $condTree = buildWithList(Tree, Obj.attributeList, \epsilon)$
15:    if $condTree.depth \geq \delta || condTree.headerTableSize \geq \delta$ then
16:        for all items $l$ in the headerTable of condTree do
17:            Obj.itemlist.add($l$)
18:            $newObj \leftarrow newGrowthObject(Obj.itemlist)$
19:            Obj.itemlist.pop()
20:            Call WriteChildRecords($key, newObj$)
21:        end for
22:    else
23:        $MaxHeap.add(FPGrowth(condTree, \epsilon, key))$
24:    end if
25: end for
26: if childTaskRecordsWereWritten then
27:     Call GrowthMapper($\delta$,$\epsilon$,$k$)
28: end if
```

The $GrowthMapper$ algorithm proceeds in a similar manner as the *Reduce* phase of the *Parallel FP-Growth* step of R-PFP. As it processes its records, it writes records to disk for any conditional FP-Tree that has a depth or header table size that is greater than or equal to $\delta$. Any other conditional trees are processed to completion. Unlike in the*Reduce* phase of the *PFP*, $GrowthMapper$ tasks can only generate a single child job. This prevents the number of child jobs from exploding. As the child job can have multiple child tasks that can process in parallel what would have been processed sequentially by the parent, this approach leads to a reduction in processing time. There is no limit to the depth of the child job generations of $GrowthMapper$ child tasks.

Each child task writes its output to file separately, they do not return any information to the parent. The final $Aggregation$ job will combine all of the outputs. Since IBM Platform Symphony manages the life cycle of parent-child jobs, the top-level parent job will not end until all of its children are done. Since the $Aggregation$ job is hard-coded to start when *Parallel FP-Growth* job ends, there is no possibility of the $Aggregation$ job missing part of the final result due to unfinished child tasks before the Aggregation job was started. Details of the framework can be found in Fig. 2b.

## C. Predicting FP-Tree Load

During the mining of an FP-Tree, the number of recursive calls to *FPGrowth()*, is proportional to the number of frequent patterns. This can number in the billions if a low support threshold is used with a database with long transactions. So we choose to only parallelize the calls with the heaviest load.

Predicting the processing load of an FP-Tree is a very difficult problem. Indeed no single factor can be used to effectively predict the load of an FP-Tree [17]. However, there are a few properties of an FP-Tree that can give an indication on the possible amount of computation that the tree will require, which include:

**Depth:** This is the longest path from any leaf node to the root of the tree. The depth of the tree is influenced by the length of transactions in the database. The depth of the tree should be the most effective means of load prediction because the number of item-sets is exponentially proportional to the depth of the tree in the worst case. However, this is not always the case, as the depth of the conditional trees generated in progressive recursive calls can reduce significantly between calls.

**Path Depth:** This is the longest path from a node whose count satisfies minimum support to the root node [11]. The path depth of tree is affected by the support value and the length of transactions in the database. In typical cases the path depth should be a better load indicator as it is likely that the conditional trees generated recursively will not have a depth that is less than the path depth of its parent [11].

**#Nodes:** This is the number of nodes in the FP-Tree. If a tree has a large number of nodes then it will take longer to traverse it.

**HeaderTable Size:** This is the number of items in the header table list. This number has a direct impact on the number of iterations that will need to be completed to

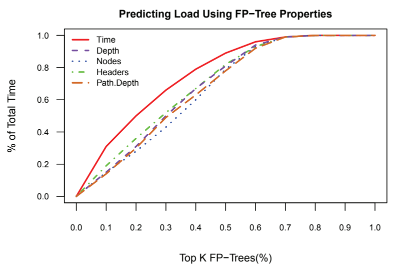


Figure 3: ROC Curve: Predicting the load of an FP-Tree using its properties. This plot was created using the Top-Level FP-Trees of the Kosarak dataset at a support count of 50. The "$Time$" curve indicates the best possible performance.

process the FP-Tree. The header list size is influenced by the number of unique items in the transaction database.

The graph in Fig. 3 shows the effectiveness of using #Nodes, Header table size, Path Depth and Depth in predicting the load of an FP-Tree. We produced the graph using the Kosarak dataset (see Table II). We ran Mahout's implementation of PFP against the Kosarak dataset, while collecting statistics about the top-level FP-Trees and the time it took for them to be processed. We also measured the total time it took for the run to complete. In the graph, the $Time$ line is generated by sorting in decreasing order the list of FP-Trees by the time it took to process them and then summing the processing time of the top 10%, 20%, 30%, ..., 100% of FP-Trees, which is the value plotted on the x-axis. We then calculate the percentage of the total processing time each of these sums represent, which is the value on the y-axis.

With this we can deduce that 10% of the FP-Trees contribute about 30% of the total processing time. This *Time* line is characteristic of what we should see if we had a 'perfect' load predictor. The other lines are produced in a similar manner except that the trees are sorted in decreasing order of the statistic that they represent rather than processing time.

The graphs show that none of the factors were able to predict the load perfectly. The graphs also show that Header table size and Depth consistently beating PathDepth and Node count. Apart from the PathDepth, these results are not surprising. Contrary to what should be expected the PathDepth performs worse than Depth as a load predictor. We found out that due to the sparseness of the dataset majority i.e $> 95\%$ of the trees had a path depth of 0. This made PathDepth ineffective as a load predictor. Please note that path depth is different from depth, please see earlier definitions.

Also, Header table size was a more effective predictor than Depth. Again, this was due to the sparseness of the dataset. The Header table size represents an upper-bound on the depth of any FP-Tree. For any FP-Tree $t$, $depth(t) \leq headerTableSize(t)$. This is true because each element in the header table appears at least once in the FP-Tree and cannot appear more than once in a given path of the FP-Tree. On the other hand, for a sparse dataset the relationship is likely $depth(t) \ll headerTableSize(t)$, as most items are not likely to occur on the similar paths of the FP-Tree. So its more than likely that the header table size will have a greater impact on the number of iterations than the depth.

We are of the opinion that more work needs to be done to properly predict the load of FP-Trees. However, based on the observations above we decided to use both the header table size and depth as load predictors for our implementation.

## V. Evaluations

In this section we describe our evaluations of our proposed algorithm. We provide the technical specifications of the cluster environment, the nature of the datasets and the methodology of the experiments we carried out.

### A. Datasets

We refer to the datasets used in our evaluation as the $Kosarak$ and $Twitter$ datasets. The Kosarak is a dataset available from the Frequent Itemset Mining Dataset Repository [18]. It contains anonymized click-stream data from a Hungarian on-line news portal. Each user visit is considered a transaction and the web-pages visited are the items. The Twitter dataset was prepared using a collection of records extracted from tweets for the month of November 2012 containing both #hashtags and URLs as part of the tweet. The data contains approximately 27.8M tweets [19], [20]. Each record in the data contains a timestamp, a user id, a list of hashtags used and a list of URLs used. We created the dataset by extracting the hashtags. The list of hashtags in a tweet are treated as a transaction and each hashtag is an item

A summary of the statistics of each of these datasets can be found in Table II. These datasets meet the criteria required for our evaluations as they are both sparse, with the Twitter dataset being extremely sparse. They both also contain long transactions, with the Kosarak dataset containing very long transactions. These characteristics are sufficiently representative of the types of datasets that might experience combinatorial explosion in frequent itemset counting.

We would like to note that the statistics of these datasets alone cannot be used to judge the scale of the task. Our goal is to parallelize the *FPGrowth()* calls of FP-Growth using a MapReduce framework. So the scale can only be truly measured by the number of *FPGrowth()* calls that are made. The number of *FPGrowth()* calls is directly proportional to the number of frequent patterns in the data. This number can be very large at low support rates. For any item-set of length $x$, the maximum number of frequent patterns is proportional to $2^x$. This number starts to be measured in the millions for an item-set where$l = 20$ and there can be several item-sets of this length in any dataset.

Table II: Dataset Statistics

| | Kosarak | Twitter |
|---|---|---|
| **# Records** | 990,002 | 22,524,846 |
| **# Unique Items** | 41,270 | 3,380,085 |
| **Avg. Transaction Length** | 8 | 1 |
| **Max. Transaction Length** | 2,498 | 30 |

Table III: Comparison of FP-Tree Depth

| | Kosarak | Twitter |
|---|---|---|
| **Min** | 7 | 1 |
| **Mean** | 627 | 7.5 |
| **Median** | 610 | 7 |
| **Max** | 1,461 | 18 |

## B. Methodology

The aim of our experiments was to measure the improvements in computational speed that R-PFP provides over PFP. Our implementation of R-PFP was built by modifying version 0.9 of Apache Mahout's implementation of PFP. In our experiment we compared our implementation of R-PFP against PFP, while varying the minimum support count threshold parameter $(-s)$ of the Mahout implementation from 50 to 140 in steps of 10. The Mahout parameters that were set are *-k (Maximum Heap Size): 50, -g (Number of Groups): 1,000 and -method (Processing Method): mapreduce*. These parameters always had the same value. Any parameter whose value is not mentioned here was set to its default value.

We used the median value of the depth of FP-Trees to select a depth threshold for our evaluations using R-PFP. The depth of FP-Trees and conditional FP-Trees will differ greatly between datasets and depending on the parameters used. The values in Table III, show the range of values for the depth of the top-level FP-Trees built using the datasets at a minimum support threshold of 50. There is a marked difference between the datasets, however this is not surprising as the FP-Tree depth is closely related to the length of the transactions in the database, see Table II. The combinatorial explosion of frequent patterns can get really bad for any FP-Tree or conditional FP-Tree with a depth $>=$ 20. However this value is too low for the Kosarak dataset, while it is too high for Twitter. Using the median value of the length of transactions (depth of FP-Trees) in the database is a satisfactory method for selecting the depth threshold. It ensures that a representative value is set for the dataset concerned and ensures that it will not be set too high or too low.

The cluster on which our experiments were run had a total of 6 nodes, one master node and 5 compute/data nodes. Each node in the cluster is a IBM System x3650 M4 server with Intel(R) Xeon(R) CPU E5-2670 at 2.60GHz, 32 vcores (2 CPU, 8 physical cores per CPU, 2 hyperthreads per core) and 64GB of RAM. Each node has a total of 7 local disks. One local disk for OS and software installs and 6 local disks for data. The operating system on all nodes is RHEL 6.3. All nodes are connected using a 1GbE network.

## VI. Results

The results of our evaluations are presented in Fig. 4. The results show that using Parent-Child MapReduce to improve the parallelization of PFP has immense benefits. It is able to reduce the time for processing data by up to approximately 68% (or 3 times) over PFP with the same parameter values. We also note that the results confirm that the computational load is affected more by the characteristics of the data than by the size of the data. It takes about 20 - 30 times more time to process the Kosarak dataset than the twitter dataset, even though the latter contains about 22 times the number of transactions of the former (see Table II).

In Fig. 4a and Fig. 4b, we see the processing times of running PFP and R-PFP against the Kosarak and Twitter datasets respectively. Each plot point on these graphs is an average over 5 runs. Fig. 4c and Fig. 4d are stacked bar graphs that show the number of child jobs that were spawned at each support count value for the Kosarak and Twitter datasets respectively. The categories on each bar are the generations at which the task was spawned in respect to the first parent job. While, Fig. 4e and Fig. 4f are box plots that show the distribution of processing times for child jobs that were spawned for the Kosarak and Twitter datasets respectively. The y-axis on both graphs uses a log-scale.

R-PFP was able to reduce the processing time on the Kosarak dataset from about $6000 secs$ to about $2000 secs$ at a support count of 50 (see Fig. 4a). This performance however decreases as the minimum support count is increased to 140. We note that this decrease in performance is a result of less parallelization. The range of depth values for the FP-Trees change as the support rate is varied. Specifically the size of the trees reduces as the minimum support count increases. Since we keep the depth threshold constant fewer FP-Trees will be selected for deep-parallelization. This fact is attested to by Fig. 4c. This implies that similar time reductions could have been achieved by reducing the depth threshold in line with the support count increases. It also implies that the 68% reduction achieved is not an upper-bound on performance. More significant time reduction is possible with lower depth thresholds.

Except with a minor decrease at support count value of 50, R-PFP did not reduce the processing time on the Twitter dataset. However, we note that even with the extra overheads involved in writing data to disk and managing the child tasks R-PFP did not increase the processing time significantly. The FP-Trees in the Twitter dataset are not very deep when compared to those in the Kosarak dataset (see Table. III). This influenced our choice for the depth threshold and led to only a few trees been selected for deep parallelization (see Fig. 4d). Just as in the case of Kosarak dataset we could improve on this result by decreasing the depth threshold. However, we are of the opinion that we have already set it low enough. PFP can process FP-Trees at such depths without experiencing exponential increases in computational time. The low depth threshold chosen may also be responsible for a significant number of child tasks being spawned after the first generation for the Twitter dataset when compared to Kosarak dataset (see Fig. 4d and Fig. 4c).

The box-plots in Fig. 4e show that a significant number child jobs have processing times that are outliers in respect to other child jobs. It would seem that this indicates that we could still maintain our current performance in reducing the processing time while spawning fewer child jobs because these jobs contains significantly more processing load. This would require devising a better method for choosing which FP-Trees whose processing will be parallelized. Such a method would theoretically only parallelize the outliers, leading to fewer child jobs and similar reduction in processing time.

## VII. Conclusions and Future Work

In this paper we presented Parent-Child MapReduce. A MapReduce technique that allows the dynamic creation and synchronization of MapReduce jobs in a hierarchical parent-child fashion. Using PFP we show the capability of Parent-Child MapReduce for parallelizing the frequent pattern growth recursive calls that are carried out during the Reduce phase of the parallel FP-Growth step of PFP. We note that it would have been impossible to parallelize this portion of the algorithm using MapReduce without the Parent-Child MapReduce feature.

Our evaluations show that incorporating Parent-Child MapReduce into PFP can significantly reduce the processing time by as much as 68% (or 3 times) on the Kosarak dataset. The algorithm did not fail with the Twitter dataset as the runtime had not started to explode exponentially in our experiments. An exponential increase in runtime would have been noticed with the Twitter dataset by reducing the minimum support threshold further, in which case the runtime reduction would be noticed. This result also shows that transaction length is a more important factor than data sparseness in determining what type of datasets might experience an exponential increase in runtime at low support thresholds.

Some improvements are needed in respect to deep parallelization of PFP. The number of child tasks spawned needs to be reduced. One possible way to control the number of child tasks created is to reduce the number of FP-Trees and conditional FP-Trees that are processed in parallel. In order to do this, we will need to be able to predict the load of the trees more effectively. As we stated previously not all trees with significant depth have a high computational load. In our future work we intend to find more effective ways of predicting the computational load of conditional FP-Trees and FP-Trees without using the depth threshold parameter.

A significant number of data mining algorithms use a recursive divide-and-conquer approach. Hitherto, it would have been difficult to parallelize such algorithms using regular MapReduce. In the future we plan to apply the Parent-Child MapReduce feature to parallelize other such algorithms.

## VIII. Acknowledgments

This work is funded in part by Natual Sciences and Engineering Research Council of Canada (NSERC), and the Big Data Research, Analytics, and Information Network (BRAIN) Alliance established by the Ontario Research Fund.

## References

[1] B. Goethals, “Survey on Frequent Pattern Mining,” Tech. Rep., 2002.

[2] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB ’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. [Online]. Available: http://dl.acm.org/citation.cfm?id=645920.672836

[3] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, “New algorithms for fast discovery of association rules,” Rochester, NY, USA, Tech. Rep., 1997.

[4] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’00. New York, NY, USA: ACM, 2000, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/342009.335372

[5] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, “Pfp: Parallel fp-growth for query recommendation,” in *Proceedings of the 2008 ACM Conference on Recommender Systems*, ser. RecSys ’08. New York, NY, USA: ACM, 2008, pp. 107–114. [Online]. Available: http://doi.acm.org/10.1145/1454008.1454027

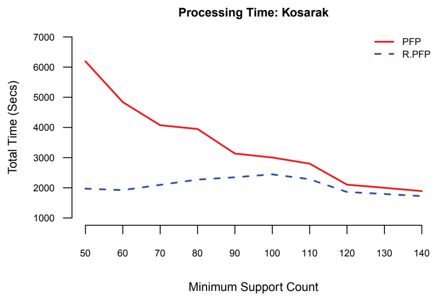
[6] I. Corporation, “IBM Platform Computing,” http://www-03.ibm.com/systems/platformcomputing/products/symphony/, accessed: 2016-02-04.

[7] A. S. Foundation, “Welcome to Apache Hadoop,” http://hadoop.apache.org/, accessed: 2016-01-21.

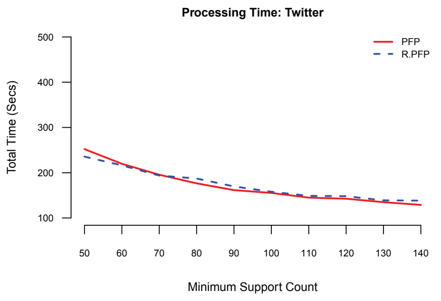
[8] ——, “Apache Spark: Lightning-fastCluster Computing,” http://spark.apache.org/, accessed: 2016-01-21.

[9] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng, “Balanced parallel fp-growth with mapreduce,” in *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*, Nov 2010, pp. 243–246.

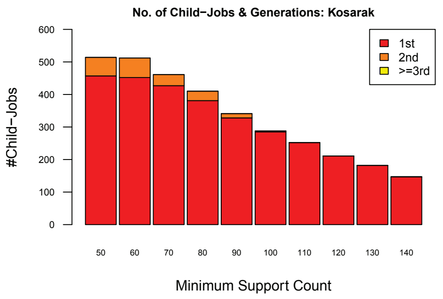
[10] O. Zaiane, M. El-Hajj, and P. Lu, “Fast parallel association rule mining without candidacy generation,” in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, 2001, pp. 665–668.



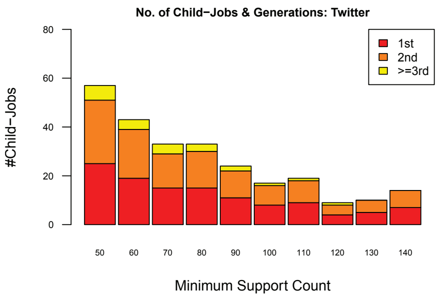
(a) Processing times for the Kosarak dataset using varying support threshold counts for PFP and R-PFP, and a constant depth threshold for R-PFP. Each measurement is an average over 5 runs.



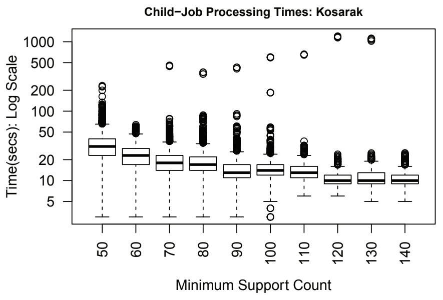
(b) Processing times for the Twitter dataset using varying support threshold counts for PFP and R-PFP, and a constant depth threshold for R-PFP. Each measurement is an average over 5 runs.



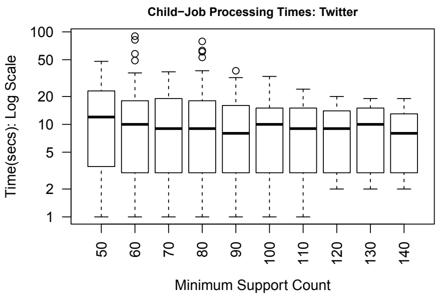
(c) Stacked Bar Graph of the number of child jobs spawned by R-PFP on the Kosarak dataset. The categories represent the generation in which the task was spawned.



(d) Stacked Bar Graph of the number of child jobs spawned by R-PFP on the Twitter dataset. The categories represent the generation in which the task was spawned.



(e) Boxplots showing the distribution of processing times for all the child jobs at different support count values for R-PFP on the Kosarak dataset. The y-axis (Time) is on a log-scale.



(f) Boxplots showing the distribution of processing times for all the child jobs at different support count values for R-PFP on the Twitter dataset. The y-axis (Time) is on a log-scale.

Figure 4: Evaluation Results: Figures (a), (c) and (e) are for the Kosarak dataset, while Figures (b), (d) and (f) are for the Twitter dataset.

[11] P. Iko and M. Kitsuregawa, “Shared nothing parallel execution of fp-growth,” *DBSJ Letters*, vol. 2, no. 1, pp. 43–46, 2003. [Online]. Available: www.scopus.com

[12] A. S. Foundation, “Apache Mahout: Scalable Machine Learning and Data Mining,” http://mahout.apache.org/, accessed: 2016-01-21.

[13] ——, “Apache Flink: Scalable Batch and Stream Data Processing,” https://flink.apache.org/, accessed: 2016-01-21.

[14] F. Bonchi, B. Goethals, F. Bonchi, and B. Goethals, “Fp-bonsai: the art of growing and pruning small fp-trees,” in *In Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2004)*. Springer-Verlag, 2004, pp. 155–160.

[15] F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi, “Exante: Anticipated data reduction in constrained pattern mining,” in *In Proc. of PKDD03*. Springer-Verlag, 2003, pp. 59–70.

[16] J. H. Reif, “Depth-First Search is Inherently Sequential,” *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[17] I. Pramudiono, K. Takahashi, A. KH Tung, and M. Kitsuregawa, “Procssing Load Prediction for Parallel FP-Growth,” in *Proc. 16th Institute of Electronics, Information and Communication Engineers Data Engineering Workshop (DEWS2005)*, 2005.

[18] B. Goethals, “Frequent Itemset Mining Dataset Repository,” http://fimi.ua.ac.be/data/, accessed: 2016-01-20.

[19] K. McKelvey and F. Menczer, “Truthy: Enabling the Study of Online Social Networks,” in *Proc. 16th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion (CSCW)*, 2013. [Online]. Available: http://arxiv.org/abs/1212.4565

[20] L. Weng, “Web-accesible Data Repository for NaN Group,” http://carl.cs.indiana.edu/data/, accessed: 2016-01-15.