# PFP: Parallel FP-Growth for Query Recommendation

Haoyuan Li
Google Beijing Research,
Beijing, 100084, China

Yi Wang
Google Beijing Research,
Beijing, 100084, China

Dong Zhang
Google Beijing Research,
Beijing, 100084, China

Ming Zhang
Dept. Computer Science,
Peking University, Beijing,
100071, China

Edward Y. Chang
Google Research, Mountain
View, CA 94043, USA

## ABSTRACT

Frequent itemset mining (FIM) is a useful tool for discovering frequently co-occurrent items. Since its inception, a number of significant FIM algorithms have been developed to speed up mining performance. Unfortunately, when the dataset size is huge, both the memory use and computational cost can still be prohibitively expensive. In this work, we propose to parallelize the FP-Growth algorithm (we call our parallel algorithm PFP) on distributed machines. PFP partitions computation in such a way that each machine executes an independent group of mining tasks. Such partitioning eliminates computational dependencies between machines, and thereby communication between them. Through empirical study on a large dataset of 802, 939 Web pages and 1, 021, 107 tags, we demonstrate that PFP can achieve virtually linear speedup. Besides scalability, the empirical study demonstrates that PFP to be promising for supporting *query recommendation* for search engines.

## Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]; H.4 [**Information Systems Applications**]

## General Terms

Algorithms, Experimentation, Human Factors, Performance

## Keywords

Parallel FP-Growth, Data Mining, Frequent Itemset Mining

## 1. INTRODUCTION

In this paper, we attack two problems. First, we parallelize frequent itemset mining (FIM) so as to deal with large-scale data-mining problems. Second, we apply our developed parallel algorithm on Web data to support *query recommendation* (or *related search*).

FIM is a useful tool for discovering frequently co-occurrent items. Existing FIM algorithms such as Apriori [9] and FP-Growth [6] can be resource intensive when a mined dataset is huge. Parallel algorithms were developed for reducing memory use and computational cost on each machine. Early efforts (related work is presented in greater detail in Section 1.1) focused on speeding up the Apriori algorithm. Since the FP-Growth algorithm has been shown to run much faster than the Apriori, it is logical to parallelize the FP-Growth algorithm to enjoy even faster speedup. Recent work in parallelizing FP-Growth [10, 8] suffers from high communication cost, and hence constrains the percentage of computation that can be parallelized. In this paper, we propose a MapReduce approach [4] of parallelizing FP-Growth algorithm (we call our proposed algorithm PFP), which intelligently *shards* a large-scale mining task into independent computational tasks and maps them onto MapReduce jobs. PFP can achieve near-linear speedup with capability of restarting from computer failures.

The resource problem of large-scale FIM could be worked around in a classic market-basket setting by pruning out items of low support. This is because low-support itemsets are usually of little practical value, e.g., a merchandise with low support (of low consumer interest) cannot help drive up revenue. However, in the Web search setting, the huge number of low-support queries, or long-tail queries [2], each must be maintained with high search quality. The importance of low-support frequent itemsets in search applications requires FIM to confront its resource bottlenecks head-on. In particular, this paper shows that a post-search recommendation tool called *related search* can benefit a great deal from our scalable FIM solution. Related search provides related queries to the user after an initial search has been completed. For instance, a query of 'apple' may suggest 'orange', 'iPod' and 'iPhone' as alternate queries. Related search can also suggest related sites of a given site (see example in Section 3.2).

### 1.1 Related Work

Some previous efforts [10] [7] parallelized the FP-Growth algorithm across multiple threads but with shared memory. However, to our problem of processing huge databases, these approaches do not address the bottleneck of huge memory requirement.


*RecSys'08,* October 23–25, 2008, Lausanne, Switzerland.

| Map inputs (transactions) key="": value | Sorted transactions (with infrequent items eliminated) | Map outputs (conditional transactions) key: value |
|---|---|---|
| f a c d g i m p | f c a m p | p: f c a m<br>m: f c a<br>a: fc<br>c: f |
| a b c f l m o | f c a b m | m: f c a b<br>b: f c a<br>a: f c<br>c: f |
| b f h j o | f b | b: f |
| b c k s p | c b p | p: c b<br>b: c |
| a f c e l p m n | f c a m p | p: f c a m<br>m: f c a<br>a: f c<br>c: f |

| Reduce inputs (conditional databases) key: value | | Conditional FP−trees |
|---|---|---|
| p: | { f c a m / f c a m / c b } | {(c:3)} \| p |
| m: | { f c a / f c a / f c a b } | { (f:3, c:3, a:3) } \| m |
| b: | { f c a / f / c } | {} \| b |
| a: | { f c / f c / f c } | { (f:3, c:3) } \| a |
| c: | { f / f / f } | { (f:3) } \| c |

**Figure 1: A simple example of distributed FP-Growth.**

To distribute both data and computation across multiple computers, Pramudiono et al [8] designed a distributed variant of the FP-growth algorithm, which runs over a cluster of computers. Some very recent work [5] [1] [3] proposed solutions to more detailed issues, including communication cost, cache consciousness, memory & I/O utilization, and data placement strategies. These approaches achieve good scalability on dozens to hundreds of computers using the MPI programming model.

However, to further improve the scalability to thousands or even more computers, we have to further reduce communication overheads between computers and support automatic fault recovery. In particular, fault recovery becomes a critical problem in a massive computing environment, because the probability that none of the thousands of computers crashes during execution of a task is close to zero. The demands of sustainable speedup and fault tolerance require highly constrained and efficient communication protocols. In this paper, we show that our proposed solution is able to address the issues of memory use, fault tolerance, in addition to more effectively parallelizing computation.

## 1.2 Contribution Summary

In summary, the contributions of this paper are as follows:

1. We propose PFP, which shards a large-scale mining task into independent, parallel tasks. PPF then uses MapReduce to take advantage of its recovery model. Empirical study shows that PFP achieves near-linear speedup.

2. With the scalability of our algorithm, we are able to mine a tag/Webpage atlas from `del.icio.us`, a Web 2.0 application that allows users tagging Webpages they have browsed. It takes $2,500$ computers only 24 minutes to mine the atlas consisting of $46,000,000$ patterns from a set of $802,939$ URLs and $1,021,107$ tags. The mined tag itemsets and Webpage itemsets are readily to support *query recommendation* or *related search*.

# 2. PFP: PARALLEL FP-GROWTH

To make this paper self-contained, we first restate the problem of FIM. We then define parameters used in PFP-Growth, and depict the algorithm. Starting in Section 2.2, we present our parallel FP-Growth algorithm, or PFP.

Let $I = \{a_1, a_2, \ldots, a_m\}$ be a *set of items*, and a *transaction database* $DB$ is a set of subsets of $I$, denoted by $DB = \{T_1, T_2, \ldots, T_n\}$, where each $T_i \subset I$ $(1 \leq i \leq n)$ is said a *transaction*. The *support* of a *pattern* $A \subset I$, denoted by $\text{supp}(A)$, is the number of transactions containing $A$ in $DB$. $A$ is a frequent pattern if and only $\text{supp}(A) \geq \xi$, where $\xi$ is a predefined minimum support threshold. Given $DB$ and $\xi$, the problem of finding the *complete set* of frequent patterns is called the *frequent itemset mining* problem.

## 2.1 FP-Growth Algorithm

FP-Growth works in a *divide and conquer* way. It requires two scans on the database. FP-Growth first computes a list of frequent items sorted by frequency in descending order (F-List) during its first database scan. In its second scan, the database is compressed into a FP-tree. Then FP-Growth starts to mine the FP-tree for each item whose support is larger than $\xi$ by recursively building its conditional FP-tree. The algorithm performs mining recursively on FP-tree. The problem of finding frequent itemsets is converted to constructing and searching trees recursively.

Figure 1 shows a simple example. The example $DB$ has five transactions composed of lower-case alphabets. The first step that FP-Growth performs is to sort items in transactions with infrequent items removed. In this example, we set $\xi = 3$ and hence keep alphabets $f, c, a, b, m, p$. After this step, for example, $T_1$ (the first row in the figure) is pruned from $\{f, a, c, d, g, i, m, p\}$ to $\{f, c, a, m, p\}$. FP-Growth then compresses these "pruned" transactions into a prefix tree, which root is the most frequent item $f$. Each path on the tree represents a set of transactions that share the same prefix; each node corresponds to one item. Each level of the tree corresponds to one item, and an item list is formed to link all transactions that possess that item. The FP-tree is a

**Procedure**: FPGrowth($DB$, $\xi$)
Define and clear F-List : $F[]$;
**foreach** *Transaction* $T_i$ *in* $DB$ **do**
  **foreach** *Item* $a_j$ *in* $T_i$ **do**
    $F[a_i]$ ++;
  **end**
**end**
Sort $F[]$;
Define and clear the root of FP-tree : $r$;
**foreach** *Transaction* $T_i$ *in* $DB$ **do**
  Make $T_i$ ordered according to $F$;
  Call $ConstructTree(T_i, r)$;
**end**
**foreach** *item* $a_i$ *in* $I$ **do**
  Call $Growth(r, a_i, \xi)$;
**end**

**Algorithm 1**: FP-Growth Algorithm

compressed representation of the transactions, and it also allows quick access to all transactions that share a given item. Once the tree has been constructed, the subsequent pattern mining can be performed. However, a compact representation does not reduce the potential combinatorial number of candidate patterns, which is the bottleneck of FP-Growth.

Algorithm 1 presents the pseudo code of FP-Growth [6]. We can estimate the time complexity of computing F-List to be $O$(DBSize) using a hashing scheme. However, the computational cost of procedure $Growth()$ (the detail is shown in Algorithm 2) is at least polynomial. The procedure $FPGrowth()$ calls the recursive procedure $Growth()$, where multiple conditional FP-trees are maintained in memory and hence the bottleneck of the FP-Growth algorithm.

FP-Growth faces the following resource challenges:

1. *Storage.* For huge $DB$'s, the corresponding FP-tree is also huge and cannot fit in main memory (or even disks). It is thus necessary to generate some *small DBs* to represent the complete one. As a result, each new *small DB* can fit into memory and generate its local FP-tree.

2. *Computation distribution.* All steps of FP-Growth can be parallelized, and especially the recursive calls to $Growth()$.

3. *Costly communication.* Previous parallel FP-Growth algorithms partition $DB$ into groups of successive transactions. Distributed FP-trees can be inter-dependent, and hence can incur frequent synchronization between parallel threads of execution.

4. *Support value.* The support threshold value $\xi$ plays an important role in FP-Growth. The larger the $\xi$, the fewer result patterns are returned and the lower the cost of computation and storage. Usually, for a large scale $DB$, $\xi$ has to be set large enough, or the FP-tree would overflow the storage. For Web mining tasks, we typically set $\xi$ to be very low to obtain long-tail itemsets. This low setting may require unacceptable computational time.

**Procedure**: Growth($r$, $a$, $\xi$)
**if** $r$ *contains a single path* $Z$ **then**
  **foreach** *combination(denoted as* $\gamma$*) of the nodes in* $Z$ **do**
    Generate pattern $\beta = \gamma \cup a$ with support = minimum support of nodes in $\gamma$;
    **if** $\beta.support > \xi$ **then**
      Call $Output(\beta)$;
    **end**
  **end**
**else**
  **foreach** $b_i$ *in* $r$ **do**
    Generate pattern $\beta = b_i \cup a$ with support = $b_i$.support;
    **if** $\beta.support > \xi$ **then**
      Call $Output(\beta)$;
    **end**
    Construct $\beta's$ conditional database ;
    Construct $\beta's$ conditional FP-tree $Tree_\beta$;
    **if** $Tree_\beta \neq \phi$ **then**
      Call $Growth(Tree_\beta, \beta, \xi)$;
    **end**
  **end**
**end**

**Algorithm 2**: The FP-Growth Algorithm.

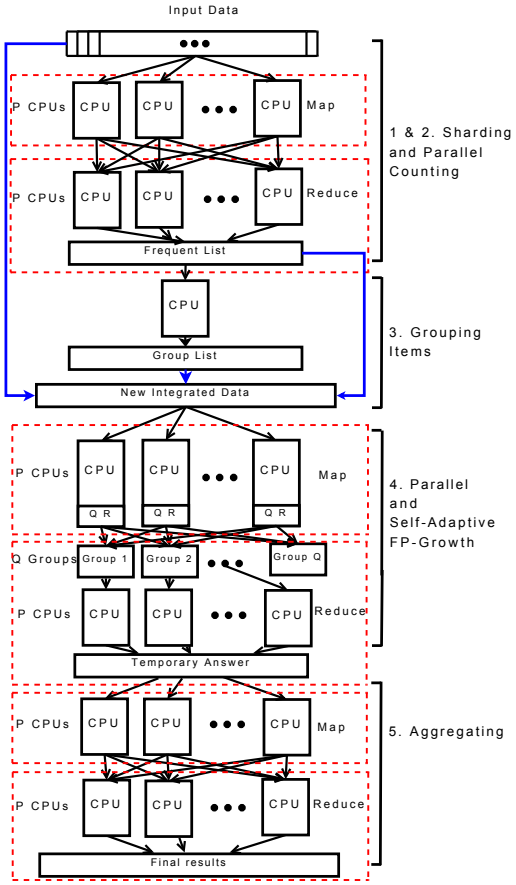



**Figure 2: The overall PFP framework, showing five stages of computation.**

## 2.2 PFP Outline

Given a transaction database *DB*, PFP uses three MapReduce [4] phases to parallelize PF-Growth. Figure 2 depicts the five steps of PFP.

**Step 1:** *Sharding*: Dividing $DB$ into successive parts and storing the parts on $P$ different computers. Such division and distribution of data is called *sharding*, and each part is called a *shard*[1].

**Step 2:** *Parallel Counting* (Section 2.3): Doing a MapReduce pass to count the support values of all items that appear in *DB*. Each mapper inputs one shard of *DB*. This step implicitly discovers the items' vocabulary $I$, which is usually unknown for a huge *DB*. The result is stored in F-list.

**Step 3:** *Grouping Items*: Dividing all the $|I|$ items on F-List into $Q$ groups. The list of groups is called *group list (*G-list), where each group is given a unique *group-id (*gid). As F-list and G-list are both small and the time complexity is $O(|I|)$, this step can complete on a single computer in few seconds.

**Step 4:** *Parallel FP-Growth* (Section 2.4): The key step of PFP. This step takes one MapReduce pass, where the map stage and reduce stage perform different important functions:

*Mapper – Generating group-dependent transactions:* Each mapper instance is fed with a shard of *DB* generated in Step 1. Before it processes transactions in the shard one by one, it reads the G-list. With the mapper algorithm detailed in Section 2.4, it outputs one or more key-value pairs, where each key is a group-id and its corresponding value is a generated group-dependent transaction.

*Reducer – FP-Growth on group-dependent shards:* When all mapper instances have finished their work, for each group-id, the MapReduce infrastructure automatically groups all corresponding group-dependent transactions into a shard of group-dependent transactions.

Each reducer instance is assigned to process one or more group-dependent shard one by one. For each shard, the reducer instance builds a local FP-tree and growth its conditional FP-trees recursively. During the recursive process, it may output discovered patterns.

**Step 5:** *Aggregating* (Section 2.5): Aggregating the results generated in Step 4 as our final result. Algorithms of the mapper and reducer are described in detail in Section 2.5.

## 2.3 Parallel Counting

Counting is a classical application of MapReduce. Because the mapper is fed with shards of *DB*, its input key-value pair would be like $\langle key, value = T_i\rangle$, where $T_i \subset DB$ is a transaction. For each item, say $a_j \in T_i$, the mapper outputs a key-value pair $\langle key' = a_j, value' = 1\rangle$.

After all mapper instances have finished, for each $key'$ generated by the mappers, the MapReduce infrastructure collects the set of corresponding values (here it is a set of 1's), say $\mathcal{S}(key')$, and feed the reducers with key-value pairs $\langle key', \mathcal{S}(key')\rangle$. The reducer thus simply outputs

$$\langle key'' = null, value'' = key' + sum(\mathcal{S}(key'))\rangle.$$

It is not difficult to see that $key''$ is an item and $value''$ is supp($key''$). Algorithm 3 presents the pseudo code of the first two steps: sharding and parallel counting. The space complexity of this algorithm is $O(DBSize/P)$ and the time complexity is $O(DBSize/P)$.

[1]MapReduce provides convenient software tools for sharding.

```
Procedure: Mapper(key, value=T_i)
foreach item a_i in T_i do
    Call Output(⟨a_i, ′1′⟩);
end
Procedure: Reducer(key=a_i, value=S(a_i))
C ← 0;
foreach item ′1′ in T_i do
    C ← C + 1;
end
Call Output(⟨null, a_i + C⟩);
```

**Algorithm 3**: The Parallel Counting Algorithm

## 2.4 Parallel FP-Growth

This step is the key in our PFP algorithm. Our solution is to convert transactions in *DB* into some new databases of *group-dependent transactions* so that local FP-trees built from different *group-dependent transactions* are independent during the recursive conditional FP-tree constructing process. We divide this step into Mapper part and Reducer part in details.

Algorithm 4 presents the pseudo code of step 4, Parallel FP-Growth. The space complexity of this algorithm is $O(Max(NewDBSize))$ for each machine.

### 2.4.1 Generating Transactions for Group-dependent Databases

When each mapper instance starts, it loads the G-list generated in Step 3. Note that G-list is usually small and can be held in memory. In particular, the mapper reads and organizes G-list as a hash map, which maps each item onto its corresponding group-id.

Because in this step, a mapper instance is also fed with a shard of *DB*, the input pair should be in the form of $\langle key, value = T_i\rangle$. For each $T_i$, the mapper performs the following two steps:

1. For each item $a_j \in T_i$, substitute $a_j$ by corresponding group-id.
2. For each group-id, say *gid*, if it appears in $T_i$, locate its right-most appearance, say $L$, and output a key-value pair $\langle key' = gid, value' = \{T_i[1]\dots T_i[L]\}\rangle$.

After all mapper instances have completed, for each distinct value of $key'$, the MapReduce infrastructure collects corresponding group-dependent transactions as value $value'$, and feed reducers by key-value pair $\langle key' = key', value'\rangle$. Here $value'$ is a group of group-dependent transactions corresponding to the same group-id, and is said a group-dependent shard.

*Notably*, this algorithm makes use of a concept introduced in [6], *pattern ending at...*, to ensure that if a group, for example $\{a, c\}$ or $\{b, e\}$, is a pattern, this support of this

**Procedure**: Mapper(key, value=$T_i$)
Load G-List;
Generate Hash Table $H$ from G-List;
a[] ← Split($T_i$);
**for** $j = |T_i| - 1$ *to* 0 **do**
  $HashNum \leftarrow$ getHashNum($H$, $a[j]$);
  **if** $HashNum \neq Null$ **then**
    Delete all pairs which hash value is $HashNum$ in $H$;
    Call $Output(\langle HashNum, a[0] + a[1] + ... + a[j]\rangle)$;
  **end**
**end**
**Procedure**: Reducer(key=$gid$,value=$DB_{gid}$)
Load G-List;
$nowGroup \leftarrow$ G-List$_g id$;
$LocalFPtree \leftarrow$ clear;
**foreach** $T_i$ *in* $DB_(gid)$ **do**
  Call $insert - build - fp - tree(LocalFPtree, T_i)$;
**end**
**foreach** $a_i$ *in* $nowGroup$ **do**
  Define and clear a size K max heap : $HP$;
  Call $TopKFPGrowth(LocalFPtree, a_i, HP)$;
  **foreach** $v_i$ *in* $HP$ **do**
    Call $Output(\langle null, v_i + supp(v_i)\rangle)$;
  **end**
**end**

**Algorithm 4**: The Parallel FP-Growth Algorithm

**Procedure**: Mapper(key, value=$v + \text{supp}(v)$)
**foreach** *item* $a_i$ *in* $v$ **do**
  Call $Output(\langle a_i, v + \text{supp}(v)\rangle)$;
**end**
**Procedure**: Reducer(key=$a_i$, value=$S(v + \text{supp}(v))$)
Define and clear a size K max heap : $HP$;
**foreach** *pattern* $v$ *in* $v + \text{supp}(v)$ **do**
  **if** $|HP| < K$ **then**
    insert $v + \text{supp}(v)$ into $HP$;
  **else**
    **if** $\text{supp}(HP[0].v) < \text{supp}(v)$ **then**
      delete top element in $HP$;
      insert $v + \text{supp}(v)$ into $HP$;
    **end**
  **end**
**end**
Call $Output(\langle null, a_i + C\rangle)$;

**Algorithm 5**: The Aggregating Algorithm

| | TTD | WWD |
|---|---|---|
| URLs | 802,939 | 802,739 |
| Tags | 1,021,107 | 1,021,107 |
| Transactions | 15,898,949 | 7,009,457 |
| Total items | 84,925,908 | 38,333,653 |

**Table 1: Properties of the TTD (tag-tag) and WWD (webpage-webpage) transaction databases.**

pattern can be counted only within the group-dependent shard with $key' = gid$, but does not rely on any other shards.

### 2.4.2 FP-Growth on Group-dependent Shards

In this step, each reducer instance reads and processes pairs in the form of $\langle key' = gid, value' = \mathcal{DB}_(gid)\rangle$ one by one, where each $\mathcal{DB}_(gid)$ is a group-dependent shard.

For each $\mathcal{DB}_(gid)$, the reducer constructs the local FP-tree and recursively builds its conditional sub-trees similar to the traditional FP-Growth algorithm. During this recursive process, it outputs found patterns. The only difference from traditional FP-Growth algorithm is that, the patterns are not output directly, but into a max-heap indexed by the support value of the found pattern. So, for each $\mathcal{DB}_(gid)$, the reducer maintains $K$ mostly supported patterns, where $K$ is the size of the max-heap $HP$. After the local recursive FP-Growth process, the reducer outputs every pattern, $v$, in the max-heap as pairs in the form of

$$\langle key'' = null, value'' = v + \text{supp}(v)\rangle$$

## 2.5 Aggregating

The aggregating step reads from the output from Step 4. For each item, it outputs corresponding top-$K$ mostly supported patterns. In particular, the mapper is fed with pairs in the form of $\langle key = null, value = v + \text{supp}(v)\rangle$. For each $a_j \in v$, it outputs a pair $\langle key' = a_j, value' = v + \text{supp}(v)\rangle$.

Because of the automatic collection function of the MapReduce infrastructure, the reducer is fed with pairs in the form of $\langle key' = a_j, value' = \mathcal{V}(a_j)\rangle$, where $\mathcal{V}(a_j)$ denotes the set of transitions including item $a_j$. The reducer just selects from $\mathcal{S}(a_j)$ the top-$K$ mostly supported patterns and outputs them.

Algorithm 5 presents the pseudo code of Step 5, Aggregating. The space complexity of this algorithm is $O(K)$ and the time complexity is $O(|I| * Max(ItemRelatedPattersNum) * log(K)/P)$.

To wrap up PFP, we revisit our example in Figure 1. The parallel algorithm projects DB onto conditional DBs, and distributes them on $P$ machines. After independent tree building and itemset mining, the frequent patterns are found and presented on the right-hand side of the figure.

# 3. QUERY RECOMMENDATION

Our empirical study was designed to evaluate the speedup of PFP and its effectiveness in supporting *query recommendatoin* or *related research*. Our data were collected from `del.icio.us`, which is a well-known bookmark sharing application. With `del.icio.us`, every user can save their bookmarks of Webpages, and tag each bookmarked Webpage with tags. Our crawl of `del.icio.us` comes from the Google search engine index and consists of a bipartite graph covering 802, 739 Webpages and 1, 021, 107 tags. From the crawled data, we generated a tag transaction database and name it TTD, and a URL transaction database WWD. Statistics of these two databases are shown in Table 1.

Because it is often that some tags are labelled many times by many users to a Webpage and some Webpages being associated with a tag many times, some tag/Webpage transactions are very long and result in very deep and inefficient FP-trees. So we divide each long transaction into many short ones. For example, a long transaction containing 100 $a$'s, 100 $b$'s and 99 $c$'s is divided into 99 short transactions $\{a, b, c\}$ and a transaction of $\{a, b\}$. This method keeps the total number of items as well as the co-occurrences of $a$, $b$ and $c$.

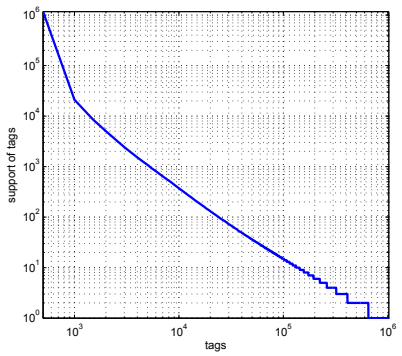
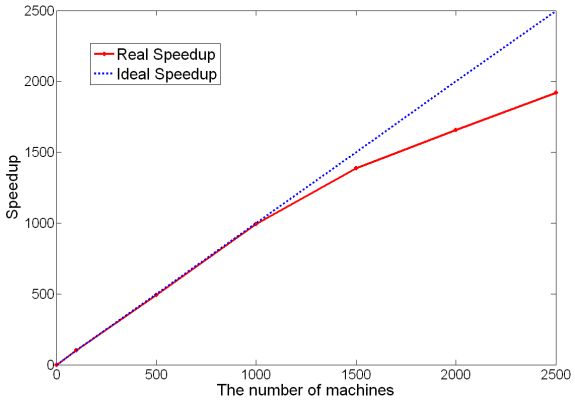



**Figure 3: The long-tail distribution of the del.icio.us tags.**




| #. machines | #. groups | Time (sec) | Speedup |
|---|---|---|---|
| 100 | 50000 | 27624 | 100.0 |
| 500 | 50000 | 5608 | 492.6 |
| 1000 | 50000 | 2785 | 991.9 |
| 1500 | 50000 | 1991 | 1387.4 |
| 2000 | 50000 | 1667 | 1657.1 |
| 2500 | 50000 | 1439 | 1919.7 |

**Figure 4: The speedup of the PFP algorithm.**

## 3.1 Speedup Evaluation Of PFP

We conducted performance evaluation on Google's MapReduce infrastructure. As shown in Section 2.2, our algorithm consists of five steps. When we distributed the processing of the TTD dataset (described in Table 1) on 2,500 computers, Step 1 and Step 2 takes 0.5 seconds, Step 5 takes 1.5 seconds, Step 3 uses only one computer and takes 1.1 seconds. Therefore, the overall speedup depends heavily upon Step 4. The overall speedup is virtually identical to the speedup of Step 4.

The evaluation shown in Figure 4 was conducted at Google's distributed data centers. Some empirical parameter values were: the number of groups, $Q$, is 50,000 and $K$ is 50. We used various numbers of computers ranging from 100 up to 2,500. It is notable that the TTD data set is so large that we had to distribute the data and computation on at least 100 computers. To quantify speedup, we took 100 machines as the baseline and made the assumption that the speedup when using 100 machines is 100, compared to using one machine. This assumption is reasonable for our experiments, since our algorithm does enjoy linear speedup when the number of machines is up to 500. From Figure 4, we can see that up to 1500 machines, the speedup is very close to the ideal speedup of 1:1. As shown in the table attached with Figure 4, the accurate speedup can be computed as $1920/2500 = 76.8\%$. This level of scalability, to the best of our knowledge, is far better than previous attempts [10, 8]. (We did not use the same data set as that used in [10, 8] to perform a side-by-side comparison. Nevertheless, the substantial overhead of these algorithms hinder them from achieving a near-linear speedup.)

Notice that the speedup cannot be always linear due to Amdahl's law. When the number of machines reaches a level that the computational time on each machine is very low, continue adding machines receives diminishing return. Nevertheless, when the dataset size increases, we can add more machines to achieve higher speedup. The good news is that the larger a mined dataset, the later Amdalh's law would take effect. Therefore, PFP is scalable for large-scale FIM tasks.

## 3.2 PFP for Query Recommendation

The bipartite graph of our `del.ico.us` data embeds two kinds of relations, Webpage-tags and tag-Webpages. From the TTD and WWD transaction databases, we mined two kinds of relationships, tag-tag and Webpage-Webpage, respectively.

Figure 5 shows some randomly selected patterns from the mining result. The support values of these patterns vary significantly, ranging from 6 to 60,726, which could show the characteristic of long tail Web data.

### 3.2.1 Tag-Tag Relationship

To the left of the figure, each row in the table shows a patten consisting of tags. The tags are in various languages, including English, Chinese, Japanese and Russian. So we have to write a short description for each pattern to explain the meaning of tags in their language.

The first three patterns contain only English tags and associate technologies with their inventors. Some rows include tags in different languages and can act as translators. Row 7, 10 and 12 are between Chinese and English; Row 8 is between Japanese and English; Row 9 is between Japanese and Chinese; and row 11 is between Russian and English.

One interesting pattern conveys more complex semantics is on Row 13, where *'Whorf'* and *'Chomsky'* are two experts in areas of *'anthropology'* and *'linguistics'*; and they did research on a tribe called *'Piraha'*. One other pattern on Row 2 associates *'browser'* with *'firebox'*. These tag-tag relationship can be effectively utilized in suggesting related queries.

### 3.2.2 Webpage-Webpage Relationship

To the right of Figure 5, each row of the table shows pattern consisting of URLs. By browsing the URLs, we find out and describe their goals and subjects. According to the descriptions, we can see that URLs in every pattern are intrinsically associated. For example, all URLs in Row 1 point to cell phone software download site. All pages in Row 10 are popular search engines used in Japan. Please refer to Fig-

ure 6 for snapshots of Web pages of these four search engines. Note that although Google is world-wide search engine and Baidu is run by a Chinese company, what are included in this pattern are their `.jp` mirrors. These Webpage-Webpage association can be used to suggested related pages of a returned page.

### 3.2.3 *Applications*

The frequent patterns can serve many applications. In addition to the previously mentioned dictionary and query suggestion, we think an interesting and practical one is visualizing the highly correlated tags as an atlas, which allows users browsing the massive Web data while keeping in their interests easily.

The output formats our PFP algorithm fits this application well — for each item, a pattern of items are associated. When the items are text like tags or URLs, many well developed methods can be used to build an efficient index on them. Therefore, given an tag (or URL), the system can instantly returns a group of tightly associated tags (or URLs). Considering a tag as a geological place, the tightly associated tags are very likely interesting places nearby. Figure 7 shows a screen shot of the visualization method implemented as a Java program. This shot shows the tag under current focus of the user in the center of the screen, with neighbors scattered around. To the left of the screen is a long list of top 100 tags, which are shown with fisheye technique and serve as a global index of the atlas.

## 4. CONCLUSIONS

In this paper we presented a massively parallel FP-Growth algorithm. This algorithm is based on a novel data and computation distribution scheme, which virtually eliminates communication among computers and makes it possible for us to express the algorithm with the MapReduce model. Experiments on a massive dataset demonstrated outstanding scalability of this algorithm. To make the algorithm suitable for mining Web data, which are usually of long tail distribution, we designed this algorithm to mine top-$k$ patterns related to each item, rather than relying on a user specified value for global minimal support threshold. We demonstrated that PFP is effective in mining tag-tag associations and WebPage-WebPage associations to support *query recommendation* or *related search*. Our future work will apply PFP on query logs to support *related search* for Google search engine.


**Figure 6: Examples of mining webpage-webpages relationship: all the three webpages (`www.google.co.jp`, `www.livedoor.com`, `www.baidu.jp`, and `www.namaan.net`) are related to Web search engines used in Japan.**

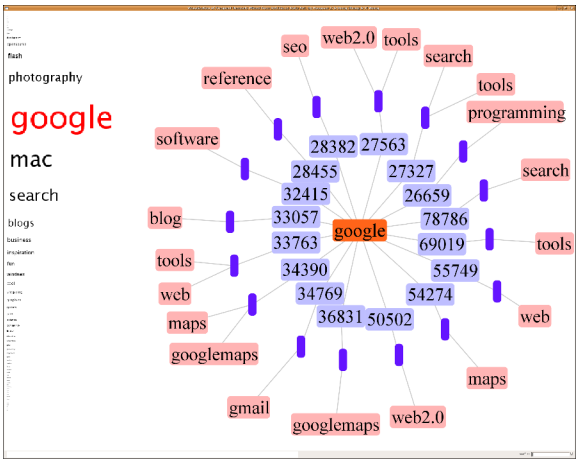



**Figure 7: Java-based Mining UI**

## 5. REFERENCES


[1] Lamine M. Aouad, Nhien-An Le-Khac, and Tahar M. Kechadi. Distributed frequent itemsets mining in heterogeneous platforms. *Engineering, Computing and Archtecture*, 1, 2007.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[3] Gregory Buehrer, Srinivasan Parthasarathy, Shirish Tatikonda, Tahsin Kurc, and Joel Saltz. Toward terabyte pattern mining: An architecture-conscious solution. In *PPOPP*, 2007.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[5] Mohammad El-Hajj and Osmar R. Za iane. Parallel leap: Large-scale maximal pattern mining in a distributed environment. In *ICPADS*, 2006.

[6] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[7] Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, 2007.

[8] Iko Pramudiono and Masaru Kitsuregawa. Parallel FP-Growth on PC cluster. In *PAKDD*, 2003.

[9] Agrawal Rakesh and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, 1994.

[10] Osmar R. Zaïane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *ICDM*, 2001.

| Tags | Description | Support |
|---|---|---|
| internet web2.0 web blog | Web2.0 is a new technique which better supports internet applications, such as blogs. | 60726 |
| browser firefox | Firefox is a famous browser | 27414 |
| macosx apple osx mac software | Mac OS X is a line of graphical operating systems of Apple Inc. | 25628 |
| dotnet c# .net development programming | Dotnet and .net are Java like development and computational platform provided by Microsoft. C# is the major programming language supported on .NET. | 6963 |
| Adobe howto photoshop tutorial | There are a lot of online tutorials teaching people how to use photoshop, a well-known software of Adobe Inc. | 6812 |
| iphone apple mac software | IPhone and mac are both famous products of Apple Inc. Apple also developed a lot of other softwares. | 2697 |
| 日志 博客 Chinese web2.0 blog | 日志 and 博客 are Chinese. 日志 means log. 博客 means blog. | 19 |
| セカンドライフ ゲーム secondlife | セカンドライフ and ゲーム are Japanese. セカンドライフ means secondlife. ゲーム means game. And secondlife is a game (http://secondlife.com/). | 14 |
| 映像 映画 PV CM 動画コンテンツ 動画投稿 動画 ウェブ | These are all Japaness Kanji and Chinese words means "image" | 13 |
| 李开复 study learning education google blog | 李开复 is the Chinese name of Kaifu Lee, a vice president of Google. He is famous for his contribution on Chinese colleage eduction. His famous blog delivers many his suggestions for Chinese students on the skills of studying and learning. | 10 |
| христианство православие orthodox | христианство and православие are Russian. христианство means Christianity. православие means Orthodox. All these three words relate to religious. | 8 |
| 正妹 taiwan album beauty photo | 正 妹 is tranditional Chinese in Taiwan. 正妹 means beauty. Lots of people search 正妹 for beauties' photos. | 7 |
| whorf piraha chomsky anthropology linguistics | Whorf and Chomsky are all experts in anthropology and linguistics, and they did research in a tribe named Piraha. | 6 |
| billgates microsoft stevejobs apple mac technology | Bill Gates and Steve Jobs are founders of Microsoft and Apple respectively, which are both high technology companies. Mac is a famous product of Apple. | 6 |

| Webpages | Description | Support |
|---|---|---|
| www.openmoko.org www.grandcentral.com www.zyb.com | Cell phone software download. | 2607 |
| www.simpy.com www.furl.net www.connotea.org del.icio.us www.masternewmedia.org/news/2006/12/01/social_bookmarking_services_and_tools.htm | Social bookmarking services. | 242 |
| www.troovy.com www.flagr.com outside.in www.wayfaring.com flickrvision.com | Online maps. | 240 |
| mail.google.com/mail www.google.com/ig gdisk.sourceforge.net www.netvibes.com | Gmail service. | 204 |
| www.trovando.it www.kartoo.com www.snap.com www.clusty.com www.aldaily.com www.quintura.com | Search engines. | 151 |
| wwwl.meebo.com www.ebuddy.com www.plugoo.com | Instant message software. | 112 |
| www.easyhotel.com www.hostelz.com www.couchsurfing.com www.tripadvisor.com www.kayak.com | Traveling agencies. | 109 |
| www.easyjet.com/it/prenota www.ryanair.com/site/IT www.edreams.it www.expedia.it www.volagratis.com/vg1 www.skyscanner.net | Italian traveling agencies. | 98 |
| www.google.com/codesearch www.koders.com www.bigbold.com/snippets www.gotapi.com 0xcc.net/blog/archives/000043.html | Code search engines and explanations. | 98 |
| www.google.co.jp www.livedoor.com www.baidu.jp www.namaan.net | Japanese Web search engines. | 36 |
| www.operator11.com www.joost.com www.keepvid.com www.getdemocracy.com www.masternewmedia.org | Video streaming. | 34 |
| www.technorati.com www.listible.com www.popurls.com | Yellow pages. | 17 |
| www.trobar.org/prosody librarianchick.pbwiki.com www.quotationspage.com www.visuwords.com | Literature sites. | 9 |

**Figure 5: Examples of mining tag-tags and webpage-webpages relationships.**