Large Scale Frequent Pattern Mining using MPI One-Sided Model

Abhinav Vishnu\(^1\) and Khushbu Agarwal\(^2\)

\(^{1,2}\)Advanced Computing, Mathematics and Data Division,
Pacific Northwest National Laboratory, 902 Battelle Blvd, Richland, WA 99352

Abstract—In this paper, we propose a work-stealing runtime — Library for Work Stealing (LibWS) — using MPI one-sided model for designing scalable FP-Growth — de facto frequent pattern mining algorithm — on large scale systems. LibWS provides locality efficient and highly scalable work-stealing techniques for load balancing on a variety of data distributions. We also propose a novel communication algorithm for FP-growth data exchange phase, which reduces the communication complexity from state-of-the-art \(\Theta(p)\) to \(\Theta(f + \frac{p}{f})\), for \(p\) processes and \(f\) frequent attributed-ids. FP-Growth is implemented using LibWS and evaluated on several work distributions and support counts. An experimental evaluation of the FP-Growth on LibWS using 4096 processes on an InfiniBand Cluster demonstrates excellent efficiency for several work distributions (91% efficiency for Poisson and 93% for Power-law). The proposed distributed FP-Tree merging algorithm provides 38x communication speedup on 4096 cores.

I. INTRODUCTION

Machine Learning and Data Mining (MLDM) algorithms are becoming increasingly important for data analysis due to the exorbitant volume of data being generated today. Frequent Pattern Mining (FPM) is an important area of data mining, which is concerned with finding frequently co-occurring attributes in a dataset. The intention of FPM is to discover strong association rules between the attributes. FPM has been applied to many tasks such as associations, correlation, clusters, and classifiers. Many algorithms have been proposed in the literature to scale FPM, such as Apriori [1], FP-Growth [2], Eclat [3] and GenMax [4]. The FP-growth algorithm has achieved significant attention in the community, while our focus is optimizing the complete FP-Growth algorithm. Buehrer et al. have proposed a scalable in-memory implementation of parallel FP-Growth algorithm [7]. In their approach, they concluded that communication and load balancing are the primary bottlenecks in parallel FP-Growth algorithm. However, they did not propose any solutions.

A. Contributions

In this paper, we address the limitations stated above and make the following contributions:

- A design of locality efficient work-stealing runtime (library for work-stealing - LibWS) using novel MPI-Remote Memory Access (MPI-RMA) features. We design a parallel FP-Growth algorithm using LibWS, which provides load-balancing on multiple data distributions among the processes. We consider several methods for stealing work across different processes such as stealsize selection, victim selection and scalable termination. We implement LibWS using MPI-3-RMA, which makes it a scalable and performance portable solution. While we demonstrate LibWS with FP-Growth, it can be readily used for other MLDM and scientific algorithms.

- A communication efficient merging of distributed FP-Trees: specifically the proposed algorithm reduces the communication complexity to \(\Theta(f + \frac{p}{f})\), while the state-of-the-art algorithm requires \(\Theta(p)\) communication, for \(p\) processes and \(f\) frequent attribute-ids.

- An implementation and evaluation of FP-Growth on LibWS using 100 million samples with several work distributions, support counts and number of cores. An experimental evaluation using 4096 processes on an InfiniBand cluster demonstrates an excellent efficiency of FP-Growth on LibWS for several work distributions.
The rest of the paper is organized as follows: Section II provides a background of the proposed work. Section III provides the preliminaries for scaling the FP-Growth algorithm on large scale systems. Section IV presents LibWS runtime based on MPI-RMA for work-stealing specifically designed for FP-Growth algorithm. Section V presents an algorithm for reducing the communication complexity of merging distributed FP-Trees. Section VI presents an empirical evaluation, section VII shows the related work, with section IX presenting the conclusions of the proposed work.

II. BACKGROUND

A. FP-Growth Algorithm

The FP-Growth algorithm is a frequent pattern mining algorithm, which requires precisely two-passes on the entire dataset. The first pass is used to compute a list of frequently occurring attribute-ids. The output of the first pass is an array sorted in non-decreasing order of frequent attribute-ids, and other associated data structures. These data structures are used in the second pass to build an FP-Tree — a modified prefix tree.

In the FP-Tree creation step, each sample is sorted in a non-increasing order of frequently occurring attribute-ids. The sorted sample is then inserted in the existing FP-Tree. The output of the algorithm is the final FP-Tree, which can be used for further data analysis.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Output sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, c, d, j, g, i, m, p</td>
</tr>
<tr>
<td>2</td>
<td>c, a, b, l, f, m, o</td>
</tr>
<tr>
<td>3</td>
<td>j, o, b, f, h</td>
</tr>
<tr>
<td>4</td>
<td>c, k, s, b, p</td>
</tr>
<tr>
<td>5</td>
<td>k, a</td>
</tr>
<tr>
<td>6</td>
<td>a, c, e, f, l, m, n, p</td>
</tr>
</tbody>
</table>

Table I shows an example of a dataset with six samples, a support count of 50% and sorted samples with frequent attribute-ids. The associated FP-Tree is created as shown in the Figure 1.

B. Message Passing Interface (MPI)

MPI supports mailbox style communication using MPI_Send and MPI_Recv primitives (several non-blocking and other variants are also supported by MPI). MPI also supports collective communication — primitives which allow processes in a group to synchronize/exchange data. Examples of collective communication are MPI_Bcast (single-root broadcast), MPI_Barrier (control synchronization) and MPI_Allreduce (reduction with the final result available on all processes in the group). An interested reader is encouraged to read MPI specification further [13], [14]. Several high performance MPI implementations are available on modern interconnects such as InfiniBand, Cray Gemini and IBM Blue Gene systems [15], [16], [17], [18], [19], [20], [21].

1) One-sided Semantics: MPI-Remote Memory Access (RMA): MPI One-sided model allows a process to expose an area of memory for reading/update by other processes in a group. MPI uses a window and optional attributes for exchanging available address spaces. It is natural to consider MPI-RMA to be an extension of CPU load/store in distributed memory. MPI-RMA provides several one-sided primitives such as MPI_Get, MPI_Put, which allow a process to read and write data from other processes’ memory asynchronously. MPI3-RMA provides atomic operations such as MPI_Fetch_and_Op, which are critical in designing LibWS. MPI-RMA supports two synchronization semantics — active and passive. In active semantics, each process participates during the data synchronization. In passive semantics, the target process is not involved in synchronization. For true asynchrony in data movement and work stealing, we leverage the passive semantics in designing and implementing LibWS.

III. PARALLEL FP-GROWTH DESIGN: PRELIMINARIES

A. Definitions

A dataset has one or more samples, and each sample has one or more attributes. Each attribute is identified by an attribute-id. An example of a dataset, samples and attribute-ids is shown in Table I. (For example, we say that the first sample’s 4th attribute has an attribute-id f). An attribute-id is considered frequent if its presence in the dataset exceeds a user-defined support count. An attribute-id is present at most once in each sample. Table II shows the parameters we use for modeling the space and time complexity of the proposed solution.

B. Data Layout

Many real-world datasets are sparse in nature. Hence, we use a compressed sparse row (CSR) representation of the
dataset.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>total process count</td>
<td>(p)</td>
</tr>
<tr>
<td>Dataset</td>
<td>(D)</td>
</tr>
<tr>
<td>number of frequent attribute-ids</td>
<td>(\alpha)</td>
</tr>
<tr>
<td>maximum attribute-id</td>
<td>(\beta)</td>
</tr>
<tr>
<td>number of samples in the dataset</td>
<td>(n)</td>
</tr>
<tr>
<td>Total occurrences of frequent attributes in (p_i)</td>
<td>(f_i)</td>
</tr>
</tbody>
</table>

**Table II**

Parameters for Modeling Time and Space Complexity of the Proposed Approach

**C. Finding Frequent Ones**

The first step in the FP-Growth algorithm is calculating the frequency count of each attribute-id in the dataset. Each process calculates the frequency on its local portion of the partitioned dataset. To get the global frequency of each attribute-id, we use an `MPI_Allreduce` at the end of this step. Initially, the space complexity of this step is \(\Theta(\beta)\), since we need to track each attribute-id. After the `MPI_Allreduce` step, the infrequent attribute-ids are eliminated, which reduces the space complexity to \(\Theta(\alpha)\).

**D. Speculative Elimination**

We observe that with increasing support count, the probability that a sample has any frequent attribute-id decreases. Recall, that each sample needs to be sorted in the non-decreasing order of occurrence frequency of its attribute-ids (Output sample column in Table I). Considering an approximately equal distribution of frequent attribute-ids to samples, the probability that an attribute in a sample is frequent is \(\frac{f_i}{|\alpha|/\beta}\). When support count is high, \(f_i \ll |\alpha|\). Hence, we can avoid a memory copy of the sample to the FP-Tree, using this probability.

Specifically, we iterate over the dataset, assuming that finding a frequent attribute-id in a sample is unlikely. When this assumption is true-positive, we save a memory copy. In case, the assumption is a false-positive, we simply copy the remaining sample to the FP-Tree and merge it. Hence, we never miss a frequent attribute-id, which keeps the accuracy of the FP-Growth algorithm intact. Figure 2 shows this with an example.

**E. FP-Tree Merging Algorithm**

The proposed merging approach is primarily based on the FP-Growth algorithm described by Buehrer et al. [7]. In their approach, they consider pruning of various FP-Tree branches by distributing the frequent attribute-ids to the processes.

We explain this with an example as shown in Figure 3. Here we consider three processes, such that the individual processes are responsible for attribute-ids \(c\), \(b\) and \(p\), respectively. As an example, one process (corresponding to left-most FP-Tree) is able to prune several branches which do not have the frequent attribute-id \(c\). Specifically, an FP-Tree branch can be pruned, if a process does not own the frequent attribute-id corresponding to root of that branch. This reduces the space-complexity incurred by each process, without loss of accuracy (since every branch is present at least on one process). During mining, any queries corresponding to the frequent attribute-id are forwarded to the associated process.

![Fig. 3. An example of FP-Tree pruning, as suggested by Buehrer et al. [7]. An example, a process which owns the center tree can eliminate the branches which do not contain \(b\).](image-url)

However, their approach uses a pointer-based FP-Tree representation. To alleviate this limitation, we first present a merging algorithm based on a compact-array representation of the FP-Trees, which can take better advantage of the memory hierarchy [22]. A compact-array representation eliminates the need for tree serialization and deserialization — required during the merge of distributed FP-Trees.

Algorithm 1 shows the steps in merging two samples (either of these samples may be an existing FP-Tree). In brief, the merging of two samples is conducted by comparing the attribute-ids at each index. In case, one of the sample size is zero (\(s1 == 0\) or \(s2 == 0\)), the other sample is returned as output. If the attribute-ids match \((t1[i1] == t2[i2])\), their frequencies are added to resulting FP-Tree. Otherwise, relative ranks of the attribute-ids are computed using a `FindRelRank` function (relative rank is non-decreasing order of attribute-id frequency in the dataset) and the subtree under the higher frequency attribute-id is appended to the resulting tree. If one sample size is smaller than the other, the remaining sample is simply copied to the output FP-Tree (while \((i1 < s1)\) or \((i2 < s2)\)).
Algorithm 1: FP-Tree Merge

Input: first sample \( t_1 \), first sample size \( s_1 \), second sample \( t_2 \), second sample size \( s_2 \), resulting tree \( r \)

Procedure \text{LPFMERGE} (\( t_1, s_1, t_2, s_2, r \))

\[
\begin{align*}
\text{if } & s_1 = 0 \text{ then } \\
& r \leftarrow t_2, \text{return}(s_2); \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{if } & s_2 = 0 \text{ then } \\
& r \leftarrow t_1, \text{return}(s_1); \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
i_1 & \leftarrow 0, i_2 \leftarrow 0, i \leftarrow 0; \\
\text{while } & (i_1 < s_1) \text{ & } (i_2 < s_2) \text{ do } \\
\text{if } & t_1[i_1] = t_2[i_2] \text{ then } \\
& r[i] = t_1[i_1]; \\
& r[i].f \leftarrow t_1[i_1].f + t_2[i_2].f; \\
& i_1 \leftarrow i_1 + 1, i_2 \leftarrow i_2 + 1, i \leftarrow i + 1; \\
\text{else } & \\
& \text{FindRelRank} (t_1[i_1], t_2[i_2], r_1, r_2); \\
& \text{mergeNodes} \leftarrow \text{AppendSubTree} (t_1, i_1, r_1, t_2, i_2, r_2, r); \\
& i \leftarrow i + \text{mergeNodes}; \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{while } & (i_1 < s_1) \text{ do } \\
& r[i].l \leftarrow t_1[i_1].l; \\
& r[i].f \leftarrow t_1[i_1].f; \\
& i_1 \leftarrow i_1 + 1, i \leftarrow i + 1; \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{while } & (i_2 < s_2) \text{ do } \\
& r[i].l \leftarrow t_2[i_2].l; \\
& r[i].f \leftarrow t_2[i_2].f; \\
& i_2 \leftarrow i_2 + 1, i \leftarrow i + 1; \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{return} (i);
\end{align*}
\]

Procedure \text{FindRelRank} (\( t_1i_1, t_2i_2, r_1, r_2 \))

\[
\begin{align*}
r_1 & \leftarrow \text{rank}(t_1i_1); \\
r_2 & \leftarrow \text{rank}(t_2i_2);
\end{align*}
\]

IV. LibWS and FP-Growth Implementation

Several languages/libraries such as X10 [23], Chapel [24], Co-Array Fortran [25], and Active Pebbles [26] provide a framework for work-stealing. At the same time, Hadoop and SPARK [27] provide functional programming constructs for designing fault tolerant MLDM algorithms [27].

However, with recent proposition of MPI3-RMA, it is possible to design and implement a work-stealing runtime which provides scalable, and portable performance as indicated by Hoefler et al. [28]. In addition, with the recently proposed algorithms for fault tolerant MPI-RMA [29], it is natural to consider MPI-RMA as the choice for designing LibWS.

There are several advantages of using MPI3-RMA as the back-end for designing and implementing LibWS. MPI is an industry standard with strong support from industry vendors — a primary reason for expecting portable performance. Using a library instead of a language for work-stealing can result in better performance in practice, as recently observed with UPC++ [30].

A. LibWS Requirements

An ideal work-stealing runtime should scale well for an arbitrary work distribution such as random, balanced or power-law. Another important characteristic of the runtime should be to maximize locality — it should complete as much local work as possible (work first), before helping other processes with their computation. It should have scalable termination mechanism and victim (process where the work is stolen from) selection, while providing asynchronous movement of work, without an explicit involvement of the victim for data movement. The last step is critical, since the victim is involved in its own computation. We consider each of these design elements in implementing LibWS.

B. Initial Conditions

Algorithm 2 shows the steps in setting up the initial conditions. In LibWS, each process is classified as a thief or victim. Specifically, \( W_{avg} \in \Theta(\frac{|D|}{T}) \), where \( W_{avg} \) represents average work and \( |D| \) represents the size of the dataset. However, the work computed by each process is \( \in \Theta(f_p) \). In essence, \( W_{p_i} \in \Theta(f_i) \), where \( f_i \) depends up on the property of the dataset and/or data distribution itself. Hence, each process can be classified as a thief or a victim: \( p_i \in \mathcal{V} \iff W_{p_i} \geq W_{avg} \), otherwise \( p_i \in \mathcal{T} \). where \( \mathcal{V} \) and \( \mathcal{T} \), represent victim and thief processes set, respectively.

In LibWS, \( W_{p_i} \) is locally calculated (section III-C) and \( W_{avg} \) is calculated using MPI_Allreduce. For a process \( p_i \in \mathcal{V} \), the work indices are exchanged and cached using MPI_Allgather. The contribution from the thieves is zero. At the completion of this step, each process has the set of victims, and an associated hashmap of work indices — the start and end indices of work exposed by the victim. After the MPI_Allgather step, the space requirement for the hashmap is \( O(|\mathcal{V}|) \).

C. Victim Selection

Once the initial conditions are setup, each process \( p_i \) uses a work-first policy for completing as much local work as possible, before stealing any work from victims. A few possibilities for victim selection are presented below:

1) Work-Size Sorted: In this policy, each thief sorts the victims in the non-increasing order of contributed work-size. This approach is intuitive, since thieves initiate work-stealing on the victims in the sorted order, as soon as they have completed their local work. A potential problem with this approach is that the victims may become a bottleneck, since thieves select the victim in a well-defined order. Specifically, considering a power-law distribution of \( f_i \), a few victims may suffer from severe network contention at the end-points. Hence, it is important to consider other approaches for victim selection.
Algorithm 2: Locality Aware Load Balancing Library and Application to FP-Growth

Procedure SetupInitConditions($p_i$)

// Use MPI Allreduce to calculate relative load, $g_{work}$: global work, $l_{work}$: local work and $c_{work}$: work contributed to work-stealing;
$g_{work} \leftarrow \text{Allreduce}(l_{work}, p)$;

// $\lambda_{p_i}$ is an array with two indices: $\lambda_{p_i}[0]$ is start-index and $\lambda_{p_i}[1]$ is the end-index of the samples owned by $p_i$ that are candidates for work-stealing; if $\frac{g_{work}}{p} > l_{work}$ then

// Local work is less than average
$\begin{array}{c}
\Rightarrow \in T)\\
\end{array}$
$c_{work} \leftarrow 0, \lambda_{p_i}[0] \leftarrow 0, \lambda_{p_i}[1] \leftarrow 0;
\]

else

// Local work is greater than average
$\begin{array}{c}
\Rightarrow \in V)\\
\end{array}$
$c_{work} \leftarrow l_{work} - \frac{g_{work}}{p};
\lambda_{p_i}[0] \leftarrow \frac{g_{work}}{p}, \lambda_{p_i}[1] \leftarrow l_{work};
\]

end

// Use MPI Allgather to find global victims
$V \leftarrow \text{Allgather}(c_{work}, p)$;
Sort ($V$);
while ($l_{work}$) do

// Merge the local work in existing tree, each process completes local work before entering work stealing;
$\text{LFPMERGE}(\ldots)$;
\end

// Add yourself to the victim set for facilitating terminating condition;
$V \leftarrow V \cup p_i$

Procedure CheckTerminate($v, p_i$)

if $|V| == 0 \& \lambda_{p_i}[0] > \lambda_{p_i}[1]$ then
$\text{return(true)}$;
else

($\text{return the first process from victim set}$)
$v \leftarrow V_0$;
$\text{return(false)}$;
end

2) Locality Aware: Locality aware work-stealing has a significant potential in alleviating the network contention at the victims. In locality aware work stealing, victims co-located on the same node are first searched for work. Remaining victims are selected using the work-size sorted approach presented above.

3) Random: Random work-stealing has proven to be a successful policy for several computation kernels. Random work stealing has the potential to alleviate network bottle-necks, since the victims are not selected in any particular order. However, it may increase the number of network requests, especially when the work exposed by the victims diminishes. We implement random work stealing in LibWS using $\text{ srand(time(NULL))}$ as the key.

D. Work-size Selection

Work-size — unit of stolen work — selection has the potential to reduce the overall communication time, and address the degree of load-imbalance. In LibWS, we consider several approaches for selecting a work-size. They are presented below:

1) Fixed: Let $w_p$ represent the unit of stolen work. A fixed value of $w_p$ is helpful in detecting termination, and gives a fair chance to each thief for stealing work from victims. A larger value of $w_p$ can result in starvation for other thieves, as they may generate many futile requests (aborted steals) for stealing work. Similarly, a smaller $w_p$ can significantly increase the number of network requests. In LibWS, we consider two parameters — communication overhead and the average amount of work to be completed by each process — for work-size selection. They are presented below.

2) Communication Overhead Driven: Communication overhead, typically modeled using LogGP[31] is an important factor for consideration in work-stealing. A careful analysis of the communication overhead is required to reduce the overall cost of communication to computation. Let $t_{p_i}$ represent the overall time spent by process $p_i$ in work-stealing and building FP-Tree. Then $t_{LFP(d)}$ is expected to be $t_{avg} \cdot \frac{d[1] - d[0]}{d[1]} - G$, where $t_{avg}$ is the average time taken for inserting a sample in an existing FP-Tree. The communication time is the sum of $(l + (r[1] - r[0]) \cdot G) \cdot \text{row-pointer}$ and $(l + (d[1] - d[0]) \cdot G) \cdot \text{dataset}$, using the LogGP model (ignoring the time for $w$). An acceptable ratio of communication to computation can be decided by the user. Specifically, in LibWS, we use 0.1 to be the acceptable overhead, which is then used to select the work-size.

3) $W_{avg}$ Bounded Work Unit: For balanced datasets, each process completes $W_{avg}$ amount of work. An ideal runtime should strive to ensure that each process completes $W_{avg}$, irrespective of the work distribution. In this technique for work-size selection, $p_i \in T$ steals as much work as possible, such that its overall work is approximately $W_{avg}$. Depending up on the work-distribution this objective may require many steals. As an example, for power-law distribution — most processes are thieves, and few are victims — this objective can be achieved by a few steal attempts. For more balanced workloads, this may take a significant number of work steals.

An advantage of this approach is that it has the potential to reduce the overall time spent in communication. The primary downside of this approach is at the ramp-down phase (when there is little work left in the system), where the number
of aborted steals may increase. The other problem with this approach is that it assumes that the cost of inserting a sample in existing FP-Tree is constant. However, the cost of insertion is dependent on the size of existing FP-Tree (algorithm 1). Hence, this approach is necessary, but insufficient in addressing the load-imbalance issue effectively.

E. Scalable Termination

In work stealing, each process must determine when to abort stealing more work from victims. Specifically, in FP-Growth algorithm, $W_{avg}$ can be used as an indicator to stealing more work. For correctness, each process must enter control synchronization (such as MPI_Barrier), after ensuring that the locally exposed work for load-balancing is complete. In LibWS, each process adds itself in the victim set to guarantee this property. While this approach is simple, it does not necessarily provide the best load-balancing. As presented earlier, the cost of insertion in an FP-Tree is not constant, hence this approach may be sub-optimal. An alternative approach is to exhaust the victim set, as shown in CheckTerminate (Algorithm 3). This approach provides a method for maximizing load-balancing, especially for large datasets, where the cost of insertion in FP-Tree cannot be predicted statically. However, this approach can result in significant aborted steals — especially when victim set is large.

We address this issue by proposing a novel termination detection approach. For each victim and work-size selection approach, a process first completes $W_{avg}$ amount of work. After this, it shuffles the victim set and looks for a communication overhead driven work from a small subset of victims. Specifically, we use an upper bound to be \( \log(V) \). This allows us to balance the remaining-work without actually looking at the entire victim set.

F. Putting it All Together: Implementation Details

We implement LibWS using MPI-RMA, especially leveraging its MPI3 features such as MPI_Fetch_and_Op. We create three separate windows \((\text{winindices})\) for work-stealing, \((\text{winrow})\) for row-pointer and \((\text{windata})\) for dataset. Recall, that passive MPI one-sided semantics require a process to call MPI_Unlock for ensuring that the data is available locally. For extracting further performance, we use hints for MPI windows such as \textsc{shared} to improve the performance of MPI\_Get and MPI\_Fetch\_and\_Op primitives.

V. SCALABLE MERGE OF DISTRIBUTED FP-TREES: HIERARCHICAL RINGS

In the previous section, we proposed LibWS for building local FP-Tree. In this section, we present the limitations of existing work in merging distributed FP-Trees and propose a novel communication method to merge these FP-Trees.

The approach proposed by Buehrer et al. [7] uses a ring algorithm for merging distributed FP-Trees. This algorithm is shown in Figure 4. In their approach, they assign a set of frequent attribute-ids to each process, such that they store the branches associated with only those attribute-ids (Figure 3).

Algorithm 3: LibWS and FP-Growth Implementation

```
Procedure LibWS(a)
  SetupInitConditions(p_i)
  // setup the initial conditions
  k ← CheckTerminate(v, p_i);
  while k == false do
    // get the work size
    w ← WorkSize();
    // atomically update the load balance counter on the victim
    // implemented using MPI\_Fetch\_Op
    \( \lambda_v[1] \) cached during
    \( \text{MPI\_Allgather} \), \( s \) is the start index of work steal!
    Lock (v, win_indices);
    \( s ← \text{FetchOp}(v, w); \)
    Unlock (v, win_indices);
    if ( \( s < \lambda_v[1] \) ) then
      // Work available, steal equal to the w or \( \lambda[1] \), whichever is lesser
      \( w ← \min (w, \lambda_v[1] - s); \)
      // Get the associated row pointer: r
      Lock (v, win_row);
      \( r ← \text{Get}(s, s + w); \)
      Unlock (v, win_row);
      // Get the associated dataset: d
      Lock (v, win_data);
      \( d ← \text{Get}(r[0], r[1]); \)
      Unlock (v, win_data);
      // Call the LFP\_MERGE algorithm
      \( \text{size}_{\text{merged}} ← \text{LFP\_MERGE} \)
      \( (d, d[1] - d[0], t_{\text{orig}}, \text{size}_{\text{orig}}, \text{t}_{\text{merged}}); \)
      \( t_{\text{orig}} ← \text{t}_{\text{merged}}; \)
      \( \text{size}_{\text{orig}} ← \text{size}_{\text{merged}}; \)
    else
      \( \text{remove v from V}; \)
      \( V ← V - v; \)
  end
  k ← CheckTerminate(v, p_i);
end
```

The other branches are simply pruned, which reduces the overall space complexity of the solution. During the merge step, the ring algorithm is used to communicate the local FP-Trees. When a process receives an FP-Tree, it prunes the tree
Let \( p \geq \alpha \), each process \( p_i \) is responsible for \( \approx \frac{\alpha}{p} \) attribute-ids. The communication complexity of such an algorithm is \( \Theta(p) \). This approach works efficiently when \( \alpha \geq p \). However, with strong scaling and/or a high support count, each process has a diminishing number of frequent attribute-ids assigned to itself. In many cases, \( p > \alpha \). In this scenario, each process is responsible for at most one frequent attribute-idi and the original algorithm [7] would still use a \( \Theta(p) \) algorithm.

We propose a new method to merge distributed FP-Trees for this scenario. In this approach, each process is responsible for exactly one frequent attribute-id. Since \( p > \alpha \), the same frequent attribute-id is replicated on \( \frac{p}{\alpha} \) processes. Hence, for each process, there is a group of processes, which have the same frequent attribute-id. Hence, the overall merging algorithm can be split in two steps: In the first step, a ring of processes with disjoint attribute-ids exchange their FP-Trees. In the second step, another ring of processes, which have identical frequent attribute-id exchange the trees. Since the ring communication occurs at two levels, we call our approach hierarchical rings.

Figure 4 shows the original algorithm with an example of 8 processes and 3 frequent attribute-ids. Figures 5 and 6 show the first and second merge steps, respectively of the proposed merging algorithm. Further details are presented here:

1) **Merge with Disjoint Frequent attribute-ids:** The first step is to create a process ring of the disjoint frequent attribute-ids and merge their respective FP-Trees. Recall, that a frequent attribute-id is replicated at most \( \frac{p}{\alpha} \) times. A ring in this step has \( \frac{p}{\alpha} \) groups with process ranks from \( \{0, \alpha, \alpha + 1, \ldots, 2 \cdot \alpha, \ldots, \frac{p}{\alpha} \cdot \alpha - 1\} \). Let \( (s_d) \) be the average size of FP-tree, which is contributed by each process. Using LogGP model, the communication complexity of this step is \( (l + s_d \cdot G) \cdot \alpha \in \Theta(\alpha) \).

2) **Merge with Identical Frequent attribute-ids:** The second step is to merge the FP-Trees of the processes, which have identical frequent attribute-id. The number of processes in this group is \( \Theta(\frac{p}{\alpha}) \). For each group, following processes are involved in the ring communication: \( \{0, \alpha, \alpha + 1, \ldots, 1, \alpha + 1, \ldots\} \). The FP-Trees are exchanged using MPI_Isend, MPI_Irecv, and MPI_Waitall routines to overlap the communication as much as possible. The expected communication time of this step is \( (l + s_r \cdot G) \cdot \frac{p}{\alpha} \), where \( s_r \) is the average communication size during each step.

As a result, the combined time complexity of the distributed FP-Tree merging is \( \Theta(\alpha + \frac{p}{\alpha}) \). A local minima in the communication time is observed when \( \alpha \leftarrow \sqrt{p} \).

VI. PERFORMANCE EVALUATION

A. Setup

1) **Experimental Testbed:** We use PNNL Constance supercomputer for performance evaluation. PNNL Constance consists of 300 Intel Haswell-based nodes in quad form. Each node features dual-socket Intel Haswell E5-2670v3 (12-core-per-socket, running at 2.3 GHz) with 64 GB of 2133 MHz ECC memory, an FDR Infiniband network card, and 480 GB local solid-state drive disk storage.

2) **Datasets:** We use the IBM Quest dataset generator for creating the datasets. The IBM Quest dataset generator represents the samples in several domains [7], [32], [4], [5], [33], [10]. This generator allows us to use very large datasets, while representing a broad category of domains. We generate up to 100 Million samples, with an average of 20 attributes per sample. A total of 1000 attribute-ids are used for generating the dataset.

3) **Work Distribution:** We use several non-uniform data distributions to emulate load-imbalance between processes: balanced, Poisson and power-law. The de facto work distribution is balanced. For Poisson work distribution we use \( \mu \approx \frac{T}{\tau} \) to be the average work completed by each process in the balanced case. For power-law distribution, we use \( \tau \) as 1.1.
B. Performance with Preliminary Optimizations

Figures 7 and 8 show the performance of the proposed FP-Growth algorithm on 1% and 3% support, respectively. We use the balanced work-distribution and compare the performance of speculative elimination with the default approach.

We observe that the overhead of finding frequent ones decreases linearly with the number of processes. However, it is a small fraction in comparison to the FP-Tree build phase. We can also observe the impact of speculative elimination which reduces the time for FP-Tree build phase significantly. Specifically, for 512 processes on 1% support, the speedup with a simple, yet effective technique is \( \approx 1.3x \). For 3% support count, the speedup is better, as expected.

With strong scaling, we observe two trends: the time to build the local FP-Tree (blue) reduces, and the time for exchanging the distributed FP-Trees (yellow) increases. As evident from Figures 7 and 8, communication time is negligible for 512 processes, but dominant on 4096 cores. In each of these charts, we have used the original merging algorithm [7]. This indicates that the proposed merging algorithm — which reduces the theoretical time complexity — has a potential to reduce the communication time. We show the actual performance results in the later part of the section.

C. LibWS Performance Evaluation with FP-Growth

Table III shows the symbols for victim selection and work-size selection approaches considered in this paper. There are ten possible combinations including balanced case.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Symbol</th>
<th>Impl. (y/n)</th>
<th>Eval. (y/n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Random WS</td>
<td>Ra</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>Locality WS</td>
<td>Lo</td>
<td>y</td>
</tr>
<tr>
<td>3</td>
<td>Sorted WS</td>
<td>So</td>
<td>y</td>
</tr>
<tr>
<td>4</td>
<td>Fixed Chunk</td>
<td>Fi</td>
<td>y</td>
</tr>
<tr>
<td>5</td>
<td>Comm. Overhead</td>
<td>Ov</td>
<td>y</td>
</tr>
<tr>
<td>6</td>
<td>W*avg</td>
<td>Av</td>
<td>y</td>
</tr>
</tbody>
</table>

TABLE III
VICTIM/WORK-SIZE APPROACHES CONSIDERED IN THIS PAPER AND THEIR IMPLEMENTATIONS: A COMBINATION OF SEVERAL APPROACHES IS CONSIDERED AS ONE CHOICE. AS AN EXAMPLE, RANDOM WORK-STEALING AND FIXED CHUNK WOULD BE RA-FI

D. Impact of Hierarchical Communication Rings

1) Performance with Power-Law Distribution: Figure 10 shows the performance with power-law load imbalance. A few processes have significant work, while others have very amount of work — due to long tail — for LFPMERGE. We observe that victim selection makes little difference to performance, since number of victims is small. The workload selection makes a significant difference. We observe that */Av approaches do very well. We expect this because this approach utilizes network bandwidth very well. The proposed termination policy ensures that using an additional few steals actually alleviates the issue of assuming a constant overhead of inserting a sample in an FP-Tree. We also observe that */Ov approaches are worse, especially because a small mis-prediction in communication overhead escalates the relative communication to computation time. The */Fi approaches are worst, since their communication overhead is the highest. In comparison to the balanced case — which is the baseline, we achieve 87% efficiency on 4096 processes.

2) Performance with Poisson Distribution: Figure 11 shows the results with Poisson distribution. The number of victims with Poisson distribution is much higher in comparison to power-law distribution. Hence, the victim selection makes a significant difference here. While we expected that Lo-* approaches would be the best, the results indicate otherwise. We attribute this to the fact that even with this distribution, the number of locally available victims are small (zero in many cases). Hence, this approach reduces to So-* based victim selection. In fact, Ra-* based approaches are the best, since they mitigate network contention better than So-* based victim selection. We also observe that */Av work-size selection is the best. This can be explained using the argument presented for power-law distribution. In comparison to the balanced baseline, we achieve 91% efficiency for 4096 processes.

Figures 7 and 8 show the performance of the proposed hierarchical rings algorithm to Buehrer’s algorithm [7]. We observe that with an exception of very low support count (0.1%), the benefits of the proposed approach are realized for increasing scale. At 4096 cores, the relative communication speedup is 38x and 13x for 3% and 1%
support counts, respectively. We expect similar speedups in communication with increasing scale (and increasing dataset sizes as well).

VII. RELATED WORK

Several attempts have been made to study the performance and provide parallel design and implementations of the FP-Growth Algorithm. Ghonting et al. have studied the performance implications of data layout and data re-use in FP-Growth, GenMax and Apriori algorithms [22]. However, these algorithms are sequential. Pramudiono et al. have presented one of the first implementations of FP-Growth on a cluster [10]. Li et al. have proposed parallel FP-Growth implementation using Mapreduce framework [6]. However, their primary target is query recommendation, and it does not solve the generic problem undertaken by the FP-Growth algorithm. Buehrer et al. have also proposed scalable design of FP-Growth algorithm [7]. In that design, Buehere et al. have presented problems with communication and load balancing in FP-Growth algorithm. However, no algorithms have been proposed to improve the load balancing. Our approach is similar to Buehrer’s approach, with major contributions to load balance the FP-Tree generation phase, minimizing space complexity in load balancing and improving the merge phase significantly in comparison to the previously proposed approaches.

VIII. ACKNOWLEDGEMENT

The research described in this paper is part of the Analysis in Motion Initiative at Pacific Northwest National Laboratory. It was conducted under the Laboratory Directed Research and Development Program at PNNL, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy.

IX. CONCLUSIONS

In this paper, we have proposed a work-stealing runtime — Library for Work Stealing (LibWS) — using MPI one-sided model for designing scalable FP-Growth — de facto frequent pattern mining algorithm — on large scale systems. LibWS provides locality efficient and highly scalable work-stealing approaches for load balancing on a variety of data distributions. We have also proposed a novel communication algorithm for FP-growth data exchange phase, which reduces the communication complexity from state-of-the-art \( \Theta(p) \) to \( \Theta(f + \frac{p}{f}) \), for \( p \) processes and \( f \) frequent attributed-ids. FP-Growth is implemented using LibWS and evaluated on several work distributions and support counts. An experimental evaluation of the FP-Growth on LibWS using 4096 processes on an InfiniBand Cluster demonstrates excellent efficiency for several work distributions (91% efficiency for Power-law and 93% for Poisson). The proposed distributed FP-Tree merging algorithm provides 38x communication speedup on 4096 cores.

REFERENCES

Fig. 10. Performance of FP-Growth on LibWS using Power-law distribution, 1% support count on 100M samples. Hierarchical rings are used to improve the communication time.

Fig. 11. Performance of FP-Growth on LibWS using Poisson distribution, 1% support count on 100M samples. Hierarchical rings are used to improve the communication time.


