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 Power-law and 93% for Poisson). 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, since it requires only two passes on the dataset. It also provides a significant compression of the original dataset using an *FP-Tree* structure — a modified prefix-tree.

The increasing data size and availability of distributed memory on supercomputers has resulted in several implementations of parallel FP-Growth algorithm [5], [6], [7]. While these approaches are efficient for balanced work-distributions and small scale, little attention has been given to the load balancing and communication bottlenecks of parallel FPM algorithms. The data distribution of many real world datasets is non-uniform (such as Poisson and power-law [8]). A skew in the data-distribution can degrade the parallel efficiency significantly. Functional programming paradigms such as MapReduce [9] address this problem using a master-slave model. However, these models incur significant data movement, due to I/O to the mappers and reducers.

Among parallel FP-Growth implementations, Pramudiono *et al.* have presented one of the first implementations of FP-Growth on a cluster [10]. However, the authors have not addressed the issue of communication complexity and load balancing in their approach. Fang *et al.* have proposed and implemented FP-Growth for GPUs [5]. However, the focus of our research is large scale distributed memory systems, which may not have GPUs. Li *et al.* have proposed a parallel FP-Growth implementation using the Mapreduce framework [6]. However, their approach is customized to query recommendation, 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 steal-size selection, victim selection and scalable termination. We implement `LibWS` using MPI3-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

(91% efficiency for Power-law and 93% for Poisson). The proposed distributed FP-Tree merging algorithm provides 38x communication speedup on 4096 cores. We plan to integrate the proposed software with Machine Learning Toolkit for Extreme Scale (MaTEx) [11], [12] and make it available for public use.

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.

	Sample	Output sample
1	a, c, d, f, g, i, m, p	a, c, f, m, p
2	c, a, b, l, f, m, o	a, c, f, b, m
3	j, o, b, f, h	f, b
4	c, k, s, b, p	c, b, p
5	k, a	a
6	a, c, e, f, l, m, n, p	a, c, f, m, p

TABLE I
A dataset with 6 samples, 16 unique attributed-ids (a - p), (support count=50%: the attribute-id should occur in at least 50% samples of the dataset). The output sample is sorted in non-decreasing frequency of attribute-ids

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

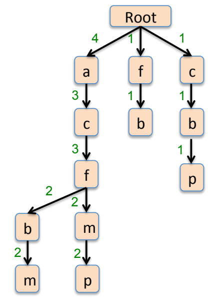

Fig. 1. FP-Tree example, each edge is marked with the number of occurrences of an attribute-id. Note that most frequent attribute-ids occur near the root, while least frequent attribute-ids occur near the leaves

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.

	Parameter	Symbol
1	total process count	p
2	Dataset	D
3	number of frequent attribute-ids	α
4	maximum attribute-id	β
5	number of samples in the dataset	n
6	Total occurrences of frequent attributes in p_i	f_i

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 $\frac{f_i}{|D|/p}$. When support count is high, $f_i \ll |D|$. 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 a 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.

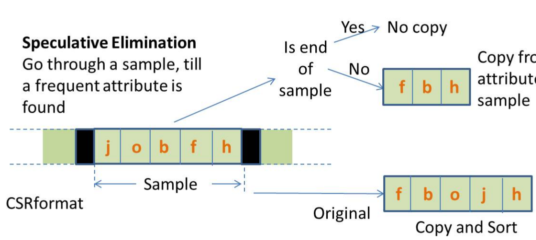

Fig. 2. Speculative Elimination: Reduces Memory Copy and Data Sorting

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.

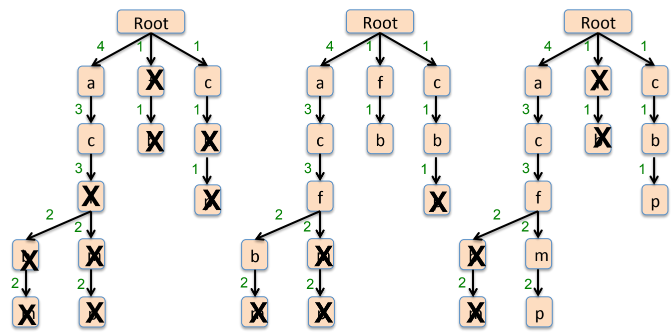

Fig. 3. An example of FP-Tree pruning, as suggested by Buehrer *et al.* [7]. An an example, a process which owns the center tree can eliminate the branches which do not contain b.

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 while($i2 < s2$)).

Algorithm 1: FP-Tree Merge

```
Input: first sample t1, first sample size s1, second
       sample t2, second sample size s2, resulting
       tree r
Procedure LFPMERGE (t1, s1, t2, s2, r)
    if s1 == 0 then
        r ← t2, return(s2);
    end
    if s2 == 0 then
        r ← t1, return(s1);
    end
    i1 ← 0, i2 ← 0, i ← 0;
    while (i1 < s1) & (i2 < s2) do
        if t1[i1] == t2[i2] then
            r[i].l ← t1[i1].l;
            r[i].f ← t1[i1].f + t2[i2].f;
            i1 ← i1 + 1, i2 ← i2 + 1, i ← i + 1;
        else
            FindRelRank (t1[i1], t2[i2], r1, r2);
            mergeNodes ← AppendSubTree
            (t1, i1, r1, t2, i2, r2, r);
            i ← i + mergeNodes;
        end
    end
    while (i1 < s1) do
        r[i].l ← t1[i1].l;
        r[i].f ← t1[i1].f;
        i1 ← i1 + 1, i ← i + 1;
    end
    while (i2 < s2) do
        r[i].l ← t2[i1].l;
        r[i].f ← t2[i2].f;
        i2 ← i2 + 1, i ← i + 1 ;
    end
    return(i);
Procedure FindRelRank(t1i1, t2i2, r1, r2)
    r1 ← rank[t1i1];
    r2 ← rank[t2i2];
```

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 implmenting `LibWS`.

B. Initial Conditions

Algorithm 2 shows the steps in setting up the initial conditions. In `LibWS`, each each process is classified as a *thief* or *victim*. Specifically, $\mathcal{W}_{avg} \in \Theta(\frac{|D|}{p})$, where $\mathcal{W}_{avg}$ represents average work and $|D|$ represents the size of the dataset. However, the work computed by each process is $\in \Theta(f_{p_i})$. In essence, $\mathcal{W}_{p_i} \in \Theta(f_{p_i})$, where (f_{p_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*: $p_i \in \mathcal{V} \iff \mathcal{W}_{p_i} \geq \mathcal{W}_{avg}$, otherwise $p_i \in \mathcal{T}$. where $\mathcal{V}$ and $\mathcal{T}$, represent victim and thief processes set, respectively.

In `LibWS`, $\mathcal{W}_{p_i}$ is locally calculated (section III-C) and $\mathcal{W}_{avg}$ 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*);

$g_{work} \leftarrow$ `Allreduce` (l_{work}, p);

(*// λ_{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 $\Longrightarrow \in \mathcal{T}$*)

$c_{work} \leftarrow 0, \lambda_{p_i}[0] \leftarrow 0, \lambda_{p_i}[1] \leftarrow 0$;

else

(*// Local work is greater than average $\Longrightarrow \in \mathcal{V}$*);

$c_{work} \leftarrow l_{work} - \frac{g_{work}}{p}$;

$\lambda_{p_i}[0] \leftarrow \frac{g_{work}}{p}, \lambda_{p_i}[1] \leftarrow l_{work}$;

end

(*//Use MPI_Allgather to find global victims*)

$\mathcal{V} \leftarrow$ `Allgather` (c_{work}, p);

`Sort` $(\mathcal{V})$;

while (l_{work}) **do**

(*// Merge the local work in existing tree, each process completes local work before entering work stealing*);

`LFPMERGE(...)`;

end

(*// Add yourself to the victim set for facilitating terminating condition*);

$\mathcal{V} \leftarrow \mathcal{V} \cup p_i$;

Procedure `CheckTerminate` (v, p_i)

if $|\mathcal{V}| == 0 \ \& \ \lambda_{p_i}[0] > \lambda_{p_i}[1]$ **then**

`return`$(true)$;

else

(*return the first process from victim set*)

$v \leftarrow \mathcal{V}_0$;

`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 bottlenecks, 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 `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_i}represent the unit of stolen work. A fixed value of w_{p_i} is helpful in detecting termination, and gives a fair chance to each thief for stealing work from victims. A larger value of w_{p_i} can result in starvation for other thieves, as they may generate many futile requests (*aborted steals*) for stealing work. Similarly, a smaller w_{p_i} 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 $t_{LFP(d)_{p_i}}$ — FP-Tree merge time with $d[1] - d[0]$ (Algorithm 3) — is expected to be $t_{avg} \cdot \frac{d[1]-d[0]}{[r[1]-r[0]]}$, where t_{avg} is the average time taken for inserting a sample in an existing FP-Tree. The communication time is the sum of $(l + (r[1] - r[0]) \cdot G)$ (row-pointer) and $(l + (d[1] - d[0]) \cdot G)$ (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) $\mathcal{W}_{avg}$ Bounded Work Unit: For balanced datasets, each process completes $\mathcal{W}_{avg}$ amount of work. An ideal runtime should strive to ensure that each process completes $\mathcal{W}_{avg}$, irrespective of the work distribution. In this technique for work-size selection, $p_i \in \mathcal{T}$ steals as much work as possible, such that its overall work is approximately $\mathcal{W}_{avg}$. 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 up 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, $\mathcal{W}_{avg}$ 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 if large.

We address this issue by proposing a novel termination detection approach. For each victim and work-size selection approach, a process first completes $\mathcal{W}_{avg}$ 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 $\log(\mathcal{V})$. 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 ($win_{indices}$) for work-stealing, (win_{row}) for row-pointer and (win_{data}) 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 `SHARED` to improve the performance of `MPI_Get` and `MPI_Fetch_and_Op` primitives.

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,
    λ_v[1] cached during
    MPI_Allgather, s is the start index
    of work steal);
    Lock (v, win_indices);
    s ← FetchOp(v, w);
    Unlock (v, win_indices);
    if (s < λ_v[1]) then
      (// Work available, steal equal to
      the w or λ[1], whichever is lesser)
      w ← Min (w, λ_v[1] − s);

      (// Get the associated row
      pointer: r);
      Lock (v, win_row);
      r ← Get (s, s + w);
      Unlock (v, win_row);

      (// Get the associated dataset:
      d);
      Lock (v, win_data);
      d ← Get (r[0], r[1]);
      Unlock (v, win_data);

      (// Call the LFPMERGE
      algorithm);
      size_merged ←LFPMERGE
      (d, d[1] − d[0], t_orig, size_orig, t_merged);

      t_orig ← t_merged;
      size_orig ← size_merged;
    else
      (remove v from V);
      V ← V − v;
    end
    k ←CheckTerminate(v, p_i);
  end
```

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). 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

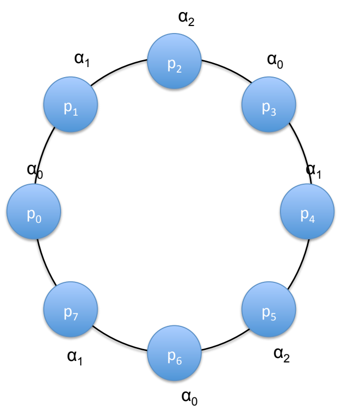

Fig. 4. An example of merging distributed FP-Trees with 8 processes and 3 frequent attribute-ids.

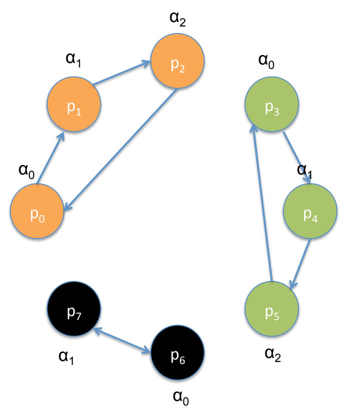

Fig. 5. The first step of proposed hierarchical rings for merging distributed FP-Trees.

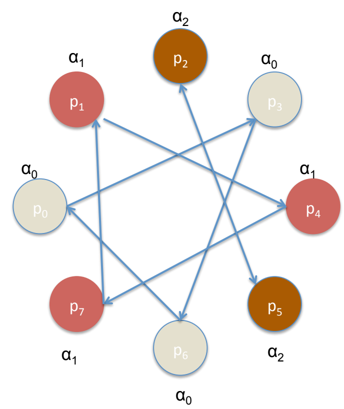

Fig. 6. The second step of proposed hierarchical rings for merging distributed FP-Trees.

using its attribute-ids as the key. It then merges the pruned tree in its FP-Tree. Specifically, when $\alpha \geq p$, 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-id 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 $\underbrace{0 \ldots \alpha}, \underbrace{\alpha + 1 \ldots 2 \cdot \alpha}, \ldots, \underbrace{\alpha_{\frac{p}{\alpha}} \ldots p - 1}$. 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: $\underbrace{0, \alpha, 2 \cdot \alpha \ldots}, \underbrace{1, \alpha + 1 \ldots}$. 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 μ $\approx$ to be the average work completed by each process in the balanced case. For power-law distribution, we use τ as `1.1`.

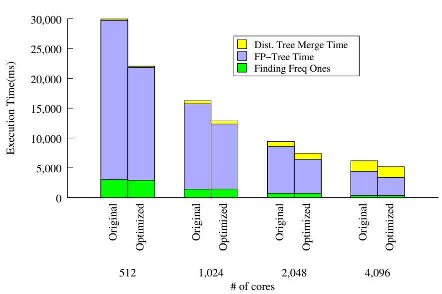

Fig. 7. Impact of Speculative Elimination (1% Support);strong scaling with 100M samples. Distributed FP-Trees are merged using Buehrer's algorithm [7]

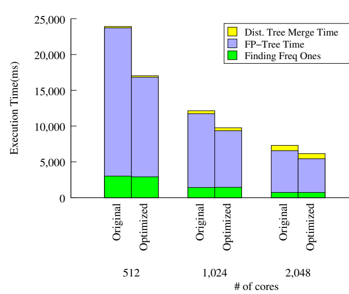

Fig. 8. Impact of Speculative Elimination (3% Support);strong scaling with 100M samples. Distributed FP-Trees are merged using Buehrer's algorithm [7]

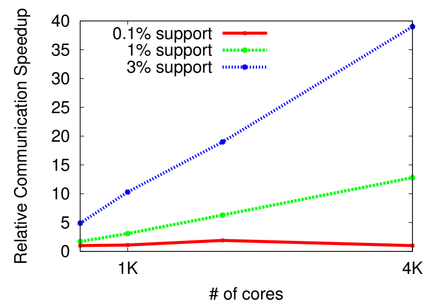

Fig. 9. Relative Communication Speedup to state-of-the-art Buehrer's algorithm [7]

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.

	Approach	Symbol	Impl. (y/n)	Eval (y/n)
3	Random WS	Ra	y	y
4	Locality WS	Lo	y	y
5	Sorted WS	So	y	y
6	Fixed Chunk	Fi	y	y
7	Comm. Overhead	Ov	y	y
8	W_{avg}	Av	y	y

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

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 work-size 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.

D. *Impact of Hierarchical Communication Rings*

Figure 9 shows the relative communication speedup 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. Ghoting *et al.* have studied the performance implications of data layout and data re-use in FP-Growth, GenMax and Apriori algorithm [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

[1] 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, 1994, pp. 487–499.

[2] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000.

[3] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "Parallel algorithms for discovery of association rules," *Data Min. Knowl. Discov.*, vol. 1, no. 4, pp. 343–373, 1997.

[4] K. Gouda and M. J. Zaki, "Genmax: An efficient algorithm for mining maximal frequent itemsets," *Data Min. Knowl. Discov.*, vol. 11, no. 3, pp. 223–242, Nov. 2005.

[5] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent itemset mining on graphics processors," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, ser. DaMoN '09, 2009, pp. 34–42.

[6] 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, 2008, pp. 107–114.

[7] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz, "Toward terabyte pattern mining: an architecture-conscious solution," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '07, 2007, pp. 2–12.

[8] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Rev.*, vol. 51, no. 4, pp. 661–703, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1137/070710111

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[10] I. Pramudiono and M. Kitsuregawa, "Parallel fp-growth on pc cluster," in *Proceedings of the 7th Pacific-Asia conference on Advances in knowledge discovery and data mining*, ser. PAKDD'03, 2003, pp. 467–473.

[11] MaTEx, "Machine Learning Toolkit for Extreme Scale." [Online]. Available: http://hpc.pnl.gov/matex

[12] J. Narasimhan, A. Vishnu, L. Holder, and A. Hoisie, "Fast support vector machines using parallel adaptive shrinking on distributed systems," *CoRR*, vol. abs/1406.5161, 2014. [Online]. Available: http://arxiv.org/abs/1406.5161

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[14] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *Euro-Par, Vol. I*, 1996, pp. 128–135.

[15] A. Vishnu and M. Krishnan, "Efficient On-demand Connection Management Protocols with PGAS Models over InfiniBand," in *International Conference on Cluster, Cloud and Grid Computing*, 2010.

[16] A. Vishnu and D. K. P. M. K. Krishnan, "A Hardware-Software Approach to Network Fault Tolerance wwith InfiniBand Cluster," in *International Conference on Cluster Computing*, 2009, pp. 479–486.

[17] A. Vishnu, B. Benton, and D. K. Panda, "High Performance MPI on IBM 12x InfiniBand Architecture," in *International Workshop on High-Level Parallel Programming Models and Supportive Environments, held in conjunction with IPDPS '07 (HIPS'07)*, 2007.

[18] A. Vishnu, A. Mamidala, S. Narravula, and D. K. Panda, "Automatic Path Migration over InfiniBand: Early Experiences," in *Proceedings of Third International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS'07*, March 2007.

[19] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu, "Noncollective communicator creation in mpi," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 282–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042508

[20] P. Balaji, R. Gupta, A. Vishnu, and P. Beckman, "Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems," *Comput. Sci.*, vol. 26, no. 3-4, pp. 247–256, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0168-y

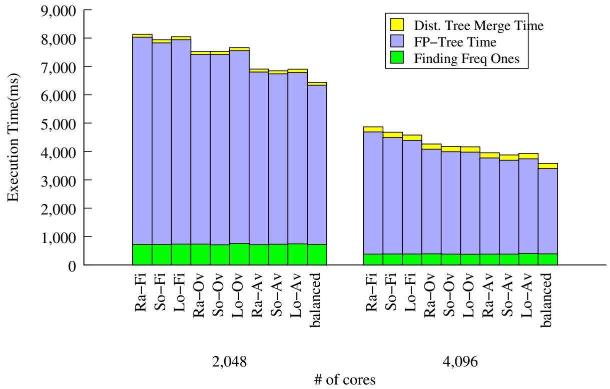

Fig. 10. Performance of FP-Growth on `LibWS` using Power-law distribution, 1% support count on 100M samples. Hierarchical rings are used to improve the communication time

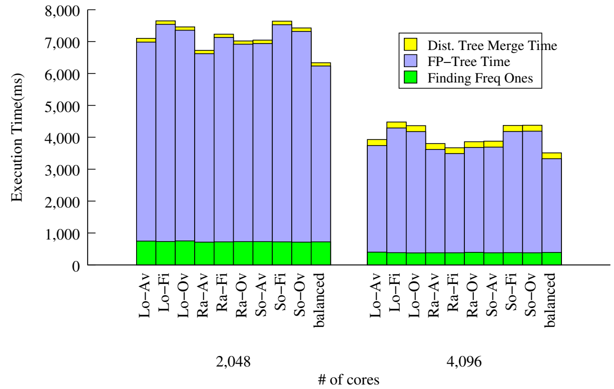

Fig. 11. Performance of FP-Growth on `LibWS` using Poisson distribution, 1% support count on 100M samples. Hierarchical rings are used to improve the communication time

[21] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda, "High performance distributed lock management services using network-based remote atomic operations." in *CCGRID*, 2007, pp. 583–590.

[22] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. D. Nguyen, Y.-K. Chen, and P. Dubey, "Cache-conscious frequent pattern mining on a modern processor," in *VLDB*, 2005, pp. 577–588.

[23] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005, pp. 519–538.

[24] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal on High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[25] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, August 1998.

[26] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: Parallel programming for data-driven applications," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11, 2011, pp. 235–244.

[27] SPARK, "Lightning Fast Cluster Computing."

[28] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided," in *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13)*, Nov. 2013, pp. 53:1–53:12.

[29] M. Besta and T. Hoefler, "Fault Tolerance for Remote Memory Access Programming Models," in *Proceedings of the 23rd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*, Jun. 2014.

[30] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, "Upc++: A pgas extension for c++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 1105–1114.

[31] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation," Santa Barbara, CA, USA, Tech. Rep., 1995.

[32] K.-M. Yu and S.-H. Wu, "An efficient load balancing multi-core frequent patterns mining algorithm," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, Nov 2011, pp. 1408–1412.

[33] G. Buehrer, S. Parthasarathy, and A. Ghoting, "Out-of-core frequent pattern mining on a commodity pc," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '06, 2006, pp. 86–95.