A Load Balancing Parallel Method for Frequent Pattern Mining on Multi-core Cluster

Lan Vu, Gita Alaghband
University of Colorado Denver,
{lan.vu, gita.alaghband}@ucdenver.edu

ABSTRACT
In this paper, we present a new parallel method named SDFEM that enables frequent pattern mining (FPM) on cluster with multiple multi-core compute nodes to provide high performance. SDFEM is distinguished from previous parallel FPM works due to incorporating three advanced features to provide high mining performance for large-scale data analytic applications. First, SDFEM combines both shared memory and distributed memory computational models to leverage benefits of shared memory within a node in cluster. Second, it employs a multi-strategy load balancing approach to address the most challenging issue of parallel FPM to balance the mining workload among all cores of the cluster. Finally, its self-adaptive mining solution with the capability of dynamically adjusting to the characteristics of the database to perform efficiently on different data types either sparse or dense. For performance evaluation, we implement SDFEM using a hybrid model of OpenMP and MPI in which OpenMP is for the shared memory model and MPI is for message passing. SDFEM has been tested on a cluster of multiple 12-core shared memory compute nodes. Our experimental results on real databases show that performance of SDFEM is up to 329.5% faster than the parallel FPM approach that uses only distributed memory model with message passing (i.e. using pure MPI). In addition, SDFEM can achieve up to 45.4 – 64.8 speedup on 120 cores (i.e. 10 compute nodes and 12 cores per node).

Author Keywords
Frequent pattern mining; multi-core cluster; high performance computing; load balancing; database.

HPC 2015, April 12 - 15, 2015, Alexandria, VA, USA

INTRODUCTION

Frequent pattern mining (FPM) is a crucial component of data mining used to find various types of relationships among variables in large databases such as associations [1], correlations [2], causality [3], sequential patterns [4], episodes [5] and partial periodicity [6]. It has many practical applications such as market analysis, biomedical and computational biology, web mining, decision support, telecommunications alarm diagnosis and prediction, and network intrusion detection [7, 8]. With growth of big data in numerous fields such as business, social media, life science, medicine, etc., applying high performance computing (HPC) using large cluster computers for FPM are essential. These machines provide massive computing and memory resources making them ideal for big data analysis. Development of high performance methods for FPM requires platform-specific design to efficiently leverage the specific platform's powerful resources.

Motivation

Our study of parallelizing FPM for large-scale data mining applications on multi-core clusters addresses three critical problems that have not been thoroughly investigated in previous studies. Solving these problems are challenging because parallel FPM usually involves multiple reduction steps, large synchronization cost and unpredictable workload for load balancing.

First, most current HPC machines are clusters consisting of many multi-core nodes whose memory is shared among cores within a node but is not directly accessible from other nodes. Recent studies have shown that a hybrid parallel programming method that applies both shared and distributed memory programming models for cluster architecture delivers better performance than parallel methods using only distributed memory model [9–13]. However, most FPM methods designed for cluster use "shared nothing" parallel model; communication among parallel processes is, therefore, done by message passing [8, 14–27]. As a result, benefits of shared memory available within each node are ignored.

Second, load balancing is highly critical for parallel FPM in cluster computing environment. FPM incurs high message passing communication cost due to large variation and amount of data communications; making load balancing a great challenge. Irregular and imbalanced computation loads may result in sharp degradation of the overall performance [19]. An efficient workload balancing solution is critical for FPM scalability on cluster architectures.

Finally, in our prior study, we have developed a sequential frequent pattern mining method that can dynamically adjust to the characteristics of the database at runtime as the pattern mining proceeds and outperforms most well-known sequential methods both on sparse and dense databases [28–31]. Applying this mining approach for parallel FPM is essential to provide high performance for this mining task on sparse and dense data.

Contribution

We present a novel parallel FPM algorithm, SDFEM, to address the above-mentioned issues. SDFEM efficiently adapts to the architecture of multi-core clusters and maximizes utilization of the available computing resources. SDFEM is distinguished from prior work due to the following features: (1) exploits the use of shared memory within a node of the cluster, (2) applies multi-level load balancing and (3) uses a self-adaptive mining approach based on data characteristics. Highlights of our contributions include:

1) SDFEM algorithm, a hybrid parallel method utilizing both shared and distributed memory programming models that performs communication within-node via shared memory and between-node via message passing. Using shared memory inter-process communication cuts down the communication cost which is quite high for message passing communication among parallel processes and reduces load balancing overhead. SDFEM also employs the mining based on data characteristics approach for faster FPM performance on both sparse and dense data.

2) A multi-level load balancing approach that uses four different strategies: dynamic job scheduling and work sharing for load balancing among cores within a node, and cyclic job scheduling and work stealing for load balancing among nodes in the cluster. This load balancing method is designed based on the data communication features of the hybrid mining model to minimize the overhead of the load balancer as well as to enhance the scalability of FPM.

3) Implementation of the algorithms using MPI (message passing interface) and OpenMP (shared memory), and performance evaluation using real-world datasets to demonstrate the efficiency of the proposed method.

BACKGROUND

Problem Statement

The FPM problem is defined as follows: Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all distinct items in the transactional database *D*. The *support* of an *itemset* α (a set of items) is the percentage of transactions containing α in *D*. A *k-itemset* α, which consists of *k* items from *I*, is frequent if α*'s support* is larger or equal to *minsup*, where *minsup* is a user-specified minimum support threshold. Given a database *D* and a *minsup*, FPM searches for the complete set of frequent itemsets in *D*. For example, given the database in Table 1 and *minsup*=20%, the frequent 1-itemsets include *a, b, c, d* and *e* while *f* is infrequent because the *support* of f is only 11%. Similarly, *ab, ac, ad, ae, bc, bd, cd, ce, de* are frequent 2-itemsets and *abc, abd, ace, ade* are the frequent 3-itemsets.

Table 1. Sample dataset with *minsup* = 20%

Transaction ID	Items	Sorted Frequent Items
1	b,d,a	a,b,d
2	c,b,d	b,c,d
3	c,d,a,e	a,c,d,e
4	d,a,e	a,d,e
5	c,b,a	a,b,c
6	c,b,a	a,b,c
7	f	
8	b,d,a	a,b,d
9	c,b,a,e,f	a,b,c,e

Related Works

A number of parallel methods have been developed for distributed memory systems. However, these methods do not take advantage of the shared memory within a node because they apply distributed memory computing model and eliminate the fact that cores within a node share same memory space. Most methods succeed in reducing data communication and increasing data independence among parallel processes but suffer from load imbalance since their load balancing strategy is heavily based on data partitioning. As a result, they may not scale well in cases like mining with small *minsup* which results in very large number of frequent patterns (the smaller the *minsup*, the larger the number of produced frequent patterns).

Pramudiono et al. [32] proposed a parallel shared nothing FPM method based on FP-growth, a well-known sequential FPM method. It partitions data equally among nodes to construct local FP-trees and deploys a model similar to MapReduce to map conditional pattern bases from a send process to a receive process. The method utilizes a characteristic called "path depth" to determine the size limit of conditional pattern bases to balance the workload among processes. Work balance is maintained by randomly selecting a process for work distribution. This approach is efficient when a good selection of path depth is made. The load balancing strategy with random selection of process is similar to one of load balance strategies in our proposed method. Yu et al. presented a parallel FPM solution for a

homogeneous PC cluster [19] based on FP-growth [25]. It uses a sampling technique for load balancing. However, they used synthetic data for experiments and reported poor performance. The method proposed by Tanbeer et al. [18], also based on FP-growth, required a simple database scan using a Parallel Pattern Tree. This study did not describe a load balancing strategy.

For mining dense databases, Sucahyo et. al [33] proposed a parallel method based on a sequential FPM method called Eclat [34]. The database is partitioned into projections, one for each item. Each projection, whose size depends on data characteristics, is stored on the local disk of a node in the cluster. Load balancing is done by distributing the projections to nodes in a round-robin fashion. All nodes follow the same order for deciding the destination node for the conditioned pattern bases, so there is a potential for blocking [32]. Similar to the method by Pramudiono [32], O¨zkural et al. presented a parallel solution using vertical data layout [27] and applied a top-down data partitioning scheme in such a way that entire database could be divided into parts with some replications so that they could be mined independently. The data were partitioned to minimize replications and maintain storage balance and computational load. Similar to the FP-growth based methods [18], [25], [32], the benefit of shared memory in multi-core clusters was ignored. Nevertheless, for dense data, they obtained good performance

Another well-known sequential method is Apriori [1]. DPA (Distributed Parallel Apriori) proposed by Yu et al. [19], is one of the few FPM methods that parallelize Apriori. Because DPA uses a breadth first mining, it is easier to maintain load balancing than in methods using depth first mining strategy. However, this approach requires multiple database scans and suffers from large synchronization overhead because of multiple iterations of the mining loop. Furthermore, Apriori usually has lower performance than most other mining methods; its parallel version (DPA) exhibits a similar poor performance level.

In summary, most existing methods supply their own strategies for data partitioning and job scheduling to balance workload and minimize communication among parallel processes. However, none of them considers the situation where load balance cannot be obtained via work partitioning, particularly when mining with very small *minsup*. This together with the three issues described in previous section motivate the development of a new parallel FPM method.

SDFEM ALGORITHM

Overview

SDFEM performs FPM by deploying a group of parallel processes and mapping each process to a multi-core node. Each process creates a group of shared memory threads, mapping each thread to a core. Threads of the same process collaborate to construct the process's projected XFP-tree. The XFP-tree is a new extension of the FP-tree [29]. The XFP-tree is a prefix tree storing compact mining data in memory. On each process, the XFP-tree is built from transactions projected to a group of items assigned to its process. The XFP-tree data structure uses shared memory and allows for some node replication but keeps the total count constant. This enhances parallelism among threads in that construction of the XFP-tree requires minimal synchronisation among threads and it is shared among the shared memory threads. Each thread uses the XFP-tree to generate its own frequent patterns [29], with each thread applying both FP-tree and bit vectors. SDFEM finally aggregates frequent patterns generated by all threads for the final output. SDFEM combines features of both distributed memory and shared memory programming models where between-node communication is done using message passing and within-node communication is done using shared memory. Figure 1 illustrates the execution model of SDFEM.

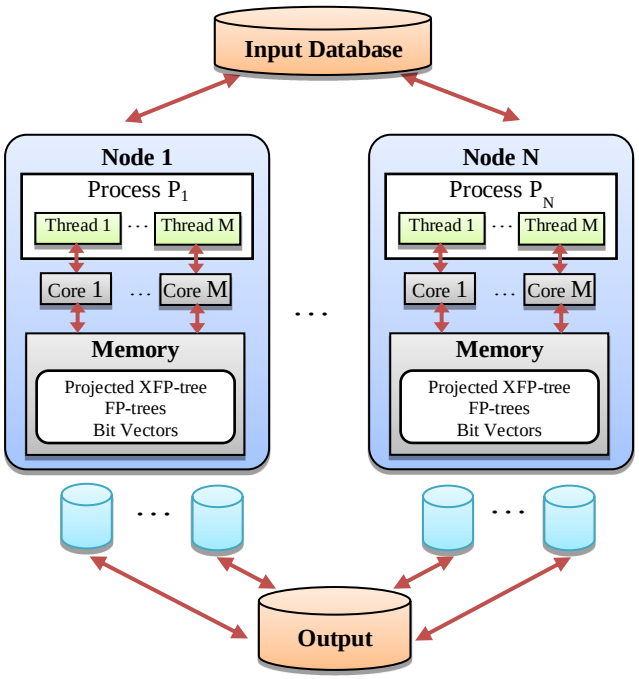

Figure 1: Overview execution model of SDFEM

SDFEM model can significantly reduce the overhead of data communication and allow more efficient load balancing. We develop a multi-level load balancing method for SDFEM using four different techniques to increase the CPU utilization and enhance performance. SDFEM performs FPM in two stages: parallel projected XFP-tree construction stage and parallel frequent pattern generation stage.

Parallel Projected XFP-tree Construction Stage

The process constructs a local projected XFP-tree from its data partition. In the first database scan, data is partitioned equally into M*N parts and each part is assigned to a thread where N is the number of processes and M is the number of threads per process. The process and its threads collaborate to compute the global count of all items by reducing the local count lists into a global one. They then identify the frequent items and sort them in frequency descending order. Figure 2 illustrates this compute task for the sample dataset in Table 1 with the execution model of 2 processes and 2 threads per process.

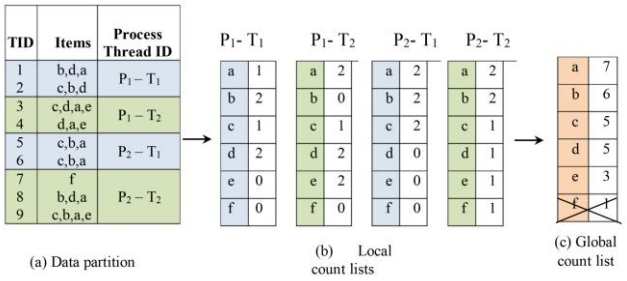

Figure 2: Computation of the global count by all processes (P = process; T= thread)

SDFEM then distributes the frequent items to processes in cyclic fashion [35]. For example, if we have two processes, P1 and P2, and a list of frequent items a, b, c, d, e, then P1 will mine all frequent patterns ending with item a, c, e and P2 will mine all frequent patterns ending with b, d. In the second database scan, each process is responsible for the entire database; each thread reads a 1/M of database and filters transactions containing the assigned frequent items to construct a local FP-tree (Figure 3) and connect them into projected XFP-tree (Figure 4). This tree also ensures that each process can work independently. The cyclic scheduling balances the data size of each tree.

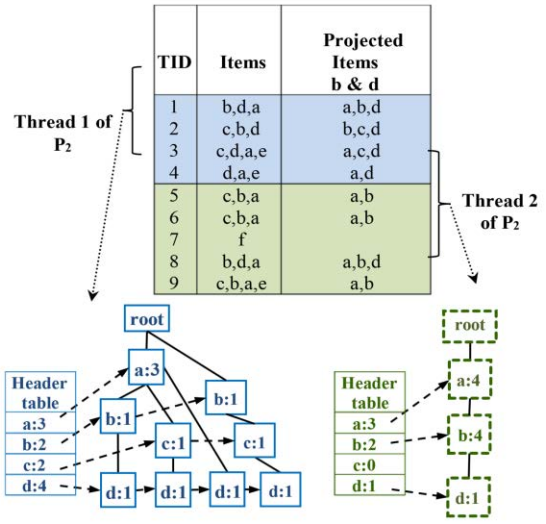

Figure 3: Construction of local FP-trees by each thread of P_2

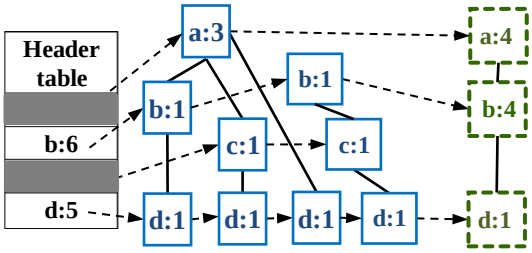

Figure 4: The project XFP-tree constructed by Process P_2

Parallel Frequent Pattern Generation Stage

After the first stage, each process has a projected XFP-tree in its memory. The process starts to independently generate frequent patterns using a multi-strategy mining approach inherited and improved from our prior sequential algorithms whose efficiency on both sparse and dense data has been shown in [29]. This new multi-strategy mining process uses ParallelMinePattern, MineFPTree, MineBitVector and LoadBalancing. The frequent pattern generation model for each thread of a process is illustrated in Figure 5.

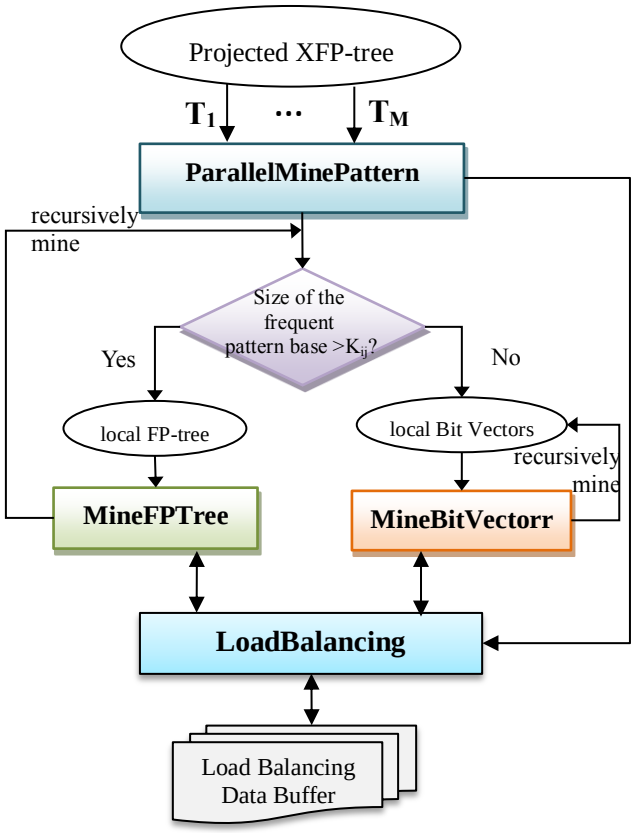

Figure 5: The mining model of SDFEM by a thread (T) within a process

ParallelMinePattern initializes the frequent pattern generation and manages the work distribution among parallel threads using dynamic job scheduling. Each thread uses this task to obtain a frequent item α in the header table of the shared projected XFP-tree. The thread invokes either *MineFPTree* or *MineBitVector* to generate the frequent patterns based on data characteristics [29].

MineFPTree is one of the two mining strategies in SDFEM. It generates frequent patterns by concatenating the suffix pattern of the previous step with each item α of the input FP-tree. Then, it constructs a child FP-tree for every item α using a data subset that is extracted from the input FP-tree and consists of sets of frequent items co-occurring with the suffix pattern. The new tree is then used as the input of this recursive mining task. This mining approach has been shown to perform well on sparse databases [36, 37, 38]. *MineFPTree* can switch to *MineBitVector* when it detects that the current data subset is dense by using a K_{ij} threshold values (one for each thread and it is computed using the method presented in [29]).

MineBitVector is the second mining strategy. It generates frequent patterns by concatenating the suffix pattern with each item of the input bit vector. It then joins pairs of bit vectors using a logical AND operation and computes their *support* using the weight vector to specify new frequent patterns. The resulting bit vectors are used as the input of *MineBitVector* to find longer frequent patterns. The mining process continues in a recursive manner until all frequent pattern are found. The efficiency of using the vertical data format on dense data has been shown in [8, 34, 39, 40]. *MineBitVector* is distinguished from the previous works because it uses a compact form of bit vectors where the compactness is presented in a weight vector.

LoadBalancing is another advanced feature of SDFEM for efficient deployment of FPM on a cluster environment because the workload associated with each item or itemset varies depending on the *minsup* input value. Hence, all threads use *LoadBalancing* to maintain workload balance during the mining process as described in more details in the next section.

MULTI-LEVEL LOAD BALANCING OF SDFEM

SDFEM is designed with two levels of parallelism: thread parallelism on shared memory multicores, and process parallelism on cluster nodes, which require minimizing communication for scalability. Therefore, load balancing in SDFEM also includes two levels: within-node load balancing for threads and between-node load balancing for processes. Balancing workload in a parallel task can be done implicitly with job scheduling and explicitly with load balancing techniques such as work sharing or work stealing. To maximize workload balance, we apply four load balancing techniques in SDFEM (Table 2). For simplicity, we implement a single data structure called load balancing data buffer (LBDB) used by LoadBalancing for both within-node and between-node load balancing purposes.

Table 2: Load balancing techniques applied in SDFEM

	Within-node load balancing	Between-node load balancing
Implicit techniques	Dynamic Job Scheduling	Static Cyclic Scheduling
Explicit techniques	Work Sharing	Work Stealing

Within-node Load Balancing

SDFEM applies dynamic job scheduling and work sharing to maintain the workload balance and optimal CPU utilization among the threads of the same process.

Dynamic job scheduling: threads of the same process dynamically obtain the next available item from the header table of the projected XFP-tree and complete mining all frequent patterns ending with this item. In OpenMP, this is implemented by simply defining a dynamic directive for the parallel loop.

Work sharing: dynamic job scheduling is efficient. It however does not ensure load balance for cases where number of frequent items is considerably small, dense databases for example. A load balancer is added to each process; it maintains a load balancing data buffer holding shared data subsets. Busy threads within a process share their workload. Available threads seek additional work from this buffer. Because this data structure is shared among threads of the same process, all threads within a process can easily update it by using critical section or lock to ensure data integrity. Figure 6 illustrates the load balancing using work sharing.

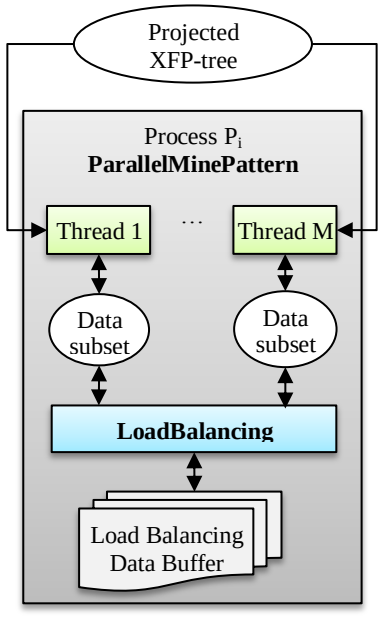

Figure 6: Within-node load balancing with work sharing

During the frequent pattern generation stage, threads periodically check this data buffer via *LoadBalancing*. If it is empty or if the number of data subsets is smaller than a certain limit Max_{LBDB}, threads will add their newly generated data subsets to the buffer, which are either FP-tree or Bit Vectors depending on the mining strategy being used (i.e. *MineFPTree* or *BitVector*). When a thread completes mining its data, it increments its counter *TC* by one where *TC* is a variable used to count number of threads completing mining assigned by the dynamic job scheduler). It then checks the data buffer to take a data subset and recursively mines this data subset. If the buffer is empty, the thread will wait until new data subsets are added, or until process status changes to *terminating*. This load balancing method ensures that threads of the same process remain busy until they all complete generating frequent patterns for the process's data partition. To minimize memory usage and the overhead of maintaining data buffer, we keep the number of buffer entries as small as possible. For example, in our experiments, the maximum number of buffer entries was set to the total number of threads of all processes.

Between-node Load Balancing

SDFEM applies static cyclic job scheduling and work stealing techniques to balance workload among the processes.

Cyclic job scheduling: due to the large amount of processed data, dynamic job scheduling may result in huge communication overhead and bottleneck. We apply static job scheduling to distribute work among processes because it is simple, has practically no overhead and does not require communication and synchronization optimization. In the first stage of projected XFP-tree construction after the frequent items are found and sorted in frequency descending order, they are distributed to processes in a cyclic fashion as illustrated in Stage 1. Each process filters database using the assigned items to construct XFP-tree.

Work stealing: in most cases when the number of frequent items are large, cyclic job scheduling is good for initial work partition among parallel processes. We apply work stealing to maintain better load balance, especially for cases where mining workload is associated with significantly varied frequent items or the number of frequent patterns is small. Based on work stealing techniques, idle processes actively look for busy processes to request more work. Since only idle processes attempt to communicate, the amount of communication is reduced and the overhead is well tolerated compared to idle time of processes without work [21]. Both work sharing and work stealing use the same data buffer. On starting the frequent pattern generation, each process keeps a process status list whose elements indicate the status of all processes (i.e. *working*: a process is still generating frequent patterns from its pre-scheduled data, *balancing*: a process completed its work portion and is requesting more work from a remote process, *terminating*: a process completed its work and there is no *working* process to request for more work). A process's status is initialized as *working*. If a process P_k completes generating all frequent patterns from its projected XFP-tree (i.e. a *balancing* process) it will pick up a victim process P_h among the *working* processes and sends a job request to P_h. Figure 7 depicts the load balancing model with work stealing between P_k and P_h.

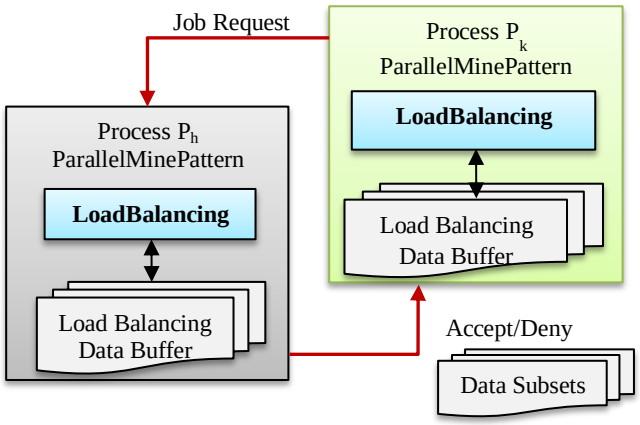

Figure 7: Between-node load balancing with work stealing

Although work stealing technique requires communication among processes, its overhead is always small because of the following reasons. First, SDFEM employs the hybrid programming model that create fewer parallel processes (i.e. usually equal to the number of nodes in the cluster). Because the number of processes involved in load balancing is small and data communication is decentralized due to work stealing, the overhead of load balancing is much smaller than that of parallel programming with pure message passing. Second, in SDFEM, only one thread, the master thread, of each process participates in work stealing while the other threads continue its mining work.

PERFORMANCE EVALUATION

Experimental Setup

Datasets: The five real datasets used for our experiments: two sparse, one moderate and two dense databases are obtained from the FIMI Repository [41], a well-known repository for FPM. The database features are reported in Table 3.

Table 3: Experimental datasets of SDFEM

Dataset	Type	# of Items	Average Length	# of Trans.
Chess	Dense	76	37	3196
Pumsb	Dense	2113	74	49046
Accidents	Moderate	468	33.8	340183
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Software: In our experiments, we use OpenMP and OpenMPI to implement SDFEM; g++ (OpenMP) and mpic++ (OpenMPI) for compilation.

Hardware: We use the cluster at http://pds.ucdenver.edu consisting of several Altus 1702 machines where each node is equipped with dual AMD Opteron 2427 processor, 2.2GHz, 24GB memory and 160 GB hard drive. The interconnection among nodes is Infiniband. Benchmark experiments of SDFEM with up to 120 cores (i.e. 10 nodes of our cluster) were conducted. The operating system is CentOS 5.3, a Linux-based distribution.

Execution Time

We demonstrate the performance of SDFEM by measuring its execution time for various number of cores of the test cluster on five datasets. The sequential test mode was done on 1 core. The parallel test was done by running the program on varying number of nodes from 1 to 10, each node runs up to 12 cores, providing a range of 12 to 120 threads or cores. Experimental results of SDFEM with varying number of cores are shown in Figure 8.

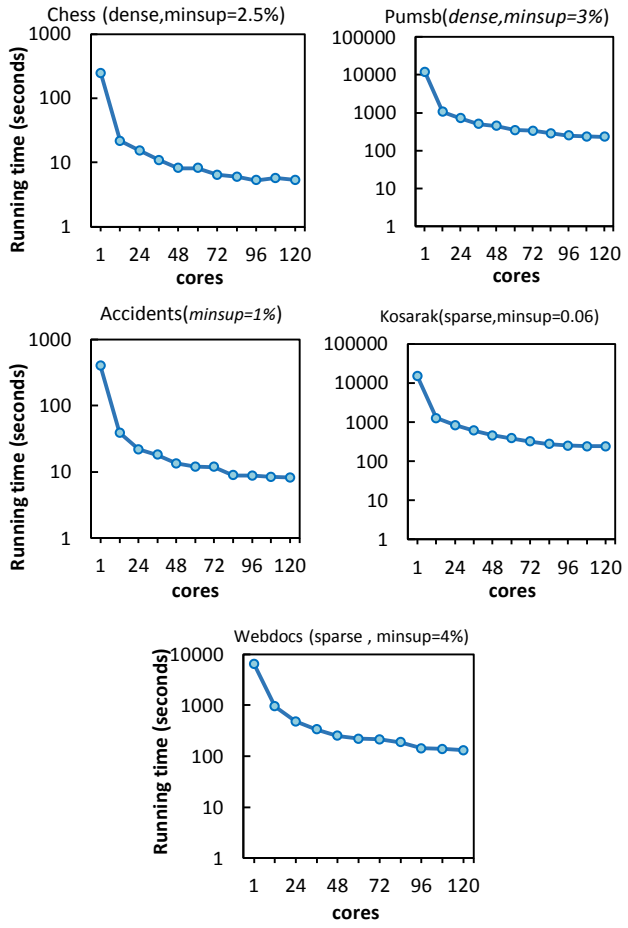

Figure 8: Running time of SDFEM (from 1 to 120 cores)

The results show significant reduction in execution time is obtained for all test cases. In the experiments, SDFEM reduces the mining time on Webdocs databases from 6460 seconds on 1 core to 130 seconds on 120 cores which saves 97.98% of the time required by sequential execution (i.e. 97.98% = (6460-130)/6460*100%). The time of SDFEM on Kosarak database is cut down 98.4% (from 15234 seconds on 1 core to 238 seconds on 120 cores). Similarly, the percentage of execution time savings for Chess, Pumsb and Accidents are 97.79%, 98.08%, 97.95% respectively. This performance improvement comes from sharing mining workload for large number of cores and reducing the amount of data that each parallel process/thread has to handle. Performance gains of SDFEM is consistent for both dense and sparse databases.

Speedup

To evaluate scalability of SDFEM to the size of cluster, we compute its speed up by dividing the sequential time of SDFEM by the parallel execution time (i.e. for 12, 24, 36,…120 cores) and present the results in Figure 9.

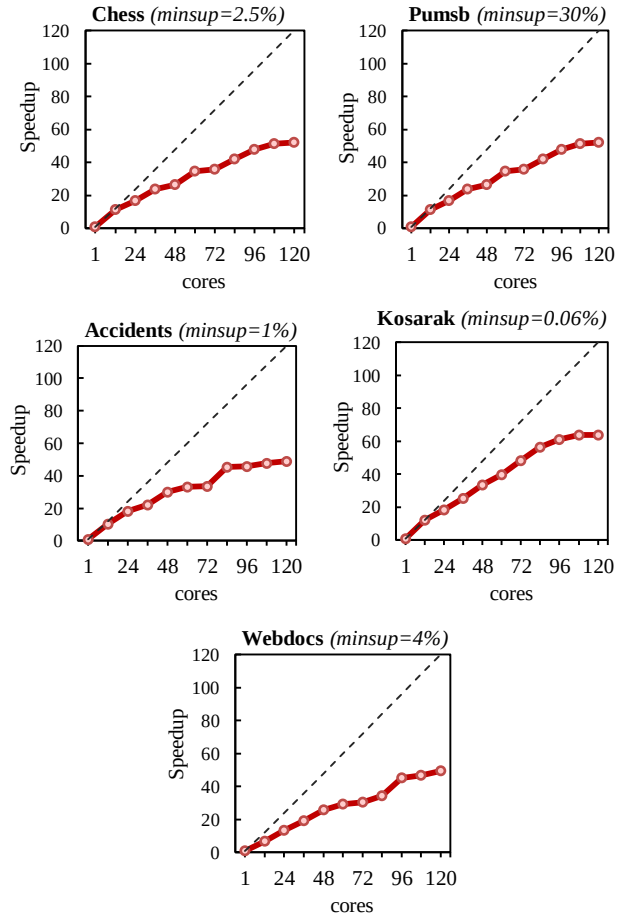

Figure 9: Speedup of SDFEM (from 1 to 120 cores)

We can see that for most cases, speed up increases when the number of cores is increased. Speed up values of five datasets on 120 cores are 45.4 (Chess), 52.1 (Pumsb), 64.8

(Kosarak), 48.7 (Accidents), 49.6 (Webdocs). Many factors limit scalability of most parallel FPM methods like synchronization, load balancing and data communication overheads or limitations of test hardware like serial I/O. It is important to note that, as the number of nodes (multiples of 12 in plots of Figure 9) is increased, higher speedups are obtained which shows SDFEM scales well for lager machines.

Impact of Hybrid MPI-OpenMP Programming Model

Application of hybrid MPI-OpenMP programming model is an important feature of SDFEM, distinguishing it from related work. We study the impact of this programming model in comparison with the traditional pure MPI model. For this purpose, SDFEM has been benchmarked using 5 compute nodes with 60 cores in total in two scenarios:

Scenario 1- 1 process per core and total processes = 60

Scenario 2- 12 threads per process, 1 process per node and total threads = 60

In Scenario 1, SDFEM performs exactly like a traditional MPI program. In Scenario 2, SDFEM applies the hybrid programming model presented in Section 3. The comparison results in Table 4 shows clear evidence that using hybrid model can significantly improve the performance. Compared to pure MPI, the hybrid mode with 12 threads per process enhances from 78.6% up to 329.5% of mining performance. For Kosarak, SDFEM with the hybrid model runs much faster than its pure MPI version (474 seconds vs. 2036 seconds).

Table 4: Time comparision of pure MPI vs. hybrid MPI-OpenMP (60 cores)

Datasets	**MinSup**	**Scenario 1** Pure MPI (1) (sec.)	**Scenario 2** Hybrid (2) (sec.)	**Performance Improvement** (4)= ((2)-(1))*100/(2)
Chess	2.50%	42	11	281.8%
Pumsb	30%	128	68	88.2%
Accidents	1%	25	14	78.6%
Kosarak	0.06%	2036	474	329.5%
Webdocs	4%	436	220	98.2%

Impact of Different Load Balancing Techniques

Another important factor impacting FPM performance is load balancing. We evaluate the efficiency of the four load balancing techniques applied in SDFEM by implementing 4 different versions of SDFEM where each integrates a combination of different techniques as listed in Table 5 and benchmarking them using 120 cores (12 threads/cores per process, 1 process per nodes). In Table 5, the underlined values indicate load-balancing techniques applied in SDFEM. All load balancing techniques are applied in SDFEM-V4 while fewer are used in the others: SDFEM-V1 (1 technique), SDFEM-V2 (2 techniques) and SDFEM-V3 (3 techniques). The test results presented in Table 6 show that SDFEM-V4 runs much faster the other three versions, showing clearly the importance of load balancing techniques. For example, for Kosarak dataset, SDFEM_V4 runs 8.7 times faster than SDFEM_V1 (8.7=2153/248), 8.1 times faster than SDFEM_V2 (8.1=2020/248), and 1.2 times faster than SDFEM_V3 (1.2=304/248).

Table 5: Four versions of SDFEM with different load balancing techniques

Techniques	**SDFEM-V1**	**SDFEM-V2**	**SDFEM-V3**	**SDFEM-V4**
Within-node job scheduling	Static (cyclic)	Dynamic	Dynamic	Dynamic
Work Sharing	N/A	N/A	Yes	Yes
Between-node job scheduling	Static (cyclic)	Static (cyclic)	Static (cyclic)	Static (cyclic)
Work Stealing	N/A	N/A	N/A	Yes

Table 6: Running time of four versions of SDFEM

Datasets	**MinSup**	**SDFEM-V1** (sec.)	**SDFEM-V2** (sec.)	**SDFEM-V3** (sec.)	**SDFEM-V4** (sec.)
Chess	2.5 %	42.2	42.3	7.5	5.2
Pumsb	30 %	3153	3152	341	252
Accidents	1 %	44	27	9.7	8.1
Kosarak	0.06 %	2153	2020	304	248
Webdocs	4 %	637	316	130	129

CONCLUSION

We present SDFEM, a novel parallel FPM for multi-core clusters, as a high performance FPM solution for large-scale applications. SDFEM has three main features which have not been investigated by prior parallel FPM work. They include (1) use of hybrid programming model to leverage benefits of shared memory and enhance the mining performance; (2) application of multiple load balancing techniques to achieve high performance and scalability; and (3) utilization of data characteristics-based mining approach that we developed to perform efficiently on different types of data. Our performance evaluation has shown that in our test cases, SDFEM results in savings of 97.79% - 98.4 % compared to sequential time. SDFEM on 120 cores of our cluster runs 45.4 – 64.8 times faster than its sequential version for the test datasets. The execution time of SDFEM are over the complete program execution and includes I/O time and the cost of performing multiple reduction steps and balancing workload.

REFERENCES

1. Agrawal, R. and Srikant, R. Fast Algorithms For Mining Association Rules In Large Databases. In *Proc. 20th International Conference on Very Large Data Bases* (1994), 487-499.
2. Brin, S., Motwani, R. and Silverstein, C. Beyond Market Baskets: Generalizing Association Rules To Correlations. In *Proc. the 1997 ACM SIGMOD international conference on Management of data* (1997), 265-276.
3. Silverstein, C., Brin, S., Motwani, R. and Ullman, J. Scalable Techniques for Mining Causal Structures. *Data Mining and Knowledge Discovery* (2000), vol. 4, no. 2-3, 163-192.
4. Agrawal, R. and Srikant, R. Mining Sequential Patterns. In *Proc. the Eleventh International Conference on Data Engineering* (1995), 3-14.
5. Mannila, H., Toivonen, H. and Verkamo, A. Inkeri. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery* (1997), vol. 1, no. 3, 259-289.
6. Han, J., Dong, G. and Yin, Y. Efficient Mining of Partial Periodic Patterns in Time Series Database. In *Proc. the 15th International Conference on Data Engineering* (1999), 106-115.
7. Han, J., Cheng, H., Xin, D. and Yan, X. Frequent Pattern Mining: Current Status And Future Directions. *Data Mining and Knowledge Discovery* (2007), vol. 15, no. 1, 55-86.
8. Burdick, D., Calimlim, M., Flannick, J., Gehrke, J. and Yiu, T. MAFIA: A Maximal Frequent Itemset Algorithm. *IEEE Transactions on Knowledge and Data Engineering* (2005), vol. 17, no. 11, 1490-1504.
9. Chorley, M. J. and Walker, D. W. Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-core Clusters. *Journal of Computational Science* (2010), vol. 1, no. 3, 168-174.
10. Rabenseifner,R., Hager, G. and Jost, G. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. in *Proc. the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (2009), 427-436.
11. He, Y. and Ding, C. H. MPI and OpenMP Paradigms on Cluster of SMP Architectures: the Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition. *Supercomputing, ACM/IEEE 2002 Conference* (2002), 1-14.
12. Wu, X. and Taylor, V. Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-scale Multicore Supercomputers. *SIGMETRICS Perform. Eval. Rev.* (2011), vol. 38, no. 4, 56-62.
13. Hager, G., Jost, G. and Rabenseifner, R. Communication Characteristics And Hybrid MPI/OpenMP Parallel Programming On Clusters Of Multi-Core SMP Nodes. in *Proc. Cray User Group Conference* (2009), p.5455.
14. Liu, L., Li, E., Zhang, Y. and Tang, Z. Optimization of Frequent Itemset Mining on Multiple-Core Processor. in *Proc. the 33rd international conference on Very large data bases* (2007), 1275-1285.
15. Zaki, M. Parallel and Distributed Association Mining: A Survey. *IEEE Concurrency* (1999), vol. 7, no. 4, 14-25.
16. Garg, R. and Mishra, P. K. Some Observations of Sequential, Parallel and Distributed Association Rule Mining Algorithms. in *Proc. the 2009 International Conference on Computer and Automation Engineering* (2009), 336-342.
17. Moonesinghe, H. D. K., Chung, M. and Tan, P. Fast Parallel Mining of Frequent Itemsets. in *Michigan State University*.
18. Tanbeer, S. K., Ahmed, C. F. and Jeong, B.-S. Parallel and Distributed Frequent Pattern Mining in Large Databases. in *Proc. the 2009 11th IEEE International Conference on High Performance Computing and Communications* (2009),407-414.
19. Yu, K.-M., Zhou, J., Hong, T.-P., and Zhou, J.-L. A Load-Balanced Distributed Parallel Mining Algorithm. *Expert Systems with Applications* (2010), vol. 37, no. 3, 2459-2464.
20. El-hajj, M. and Zaïane, O. R. Parallel Leap: Large-scale Maximal Pattern Mining In A Distributed Environment. in *Proc. 12th International Conference on Parallel and Distributed Systems* (2006).
21. Dinan, J., Olivier, S., Sabin, Prins, G. J., Sadayappan, P., and Tseng, C.-W. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. *Parallel and Distributed Processing Symposium, International* (2007), vol. 0, p. 391.
22. Yu, K.-M., Zhou, J. and Hsiao, W. Load Balancing Approach Parallel Algorithm for Frequent Pattern Mining. *Parallel Computing Technologies* (2007), vol. 4671, V. Malyshkin, Ed., Springer Berlin Heidelberg, 623-631.
23. Manaskasemsak, B., Benjamas, N., Rungsawang, A., Surarerks, A. and Uthayopas, P. Parallel Association Rule Mining Based On FI-Growth Algorithm. in *Proc. 2007 International Conference on Parallel and Distributed Systems* (2007), 1-8.
24. Ramaiah, B. Janaki, Reddy, A. Rama Mohan, and

Kumari, M. Kamala. Parallel Privacy Preserving Association Rule Mining on PC Clusters. *IEEE International Advance Computing Conference* (2009), 1538-1542.

25. Yu, K.-M., and Zhou, J. Parallel TID-based Frequent Pattern Mining Algorithm on a PC Cluster and Grid Computing System. *Expert Syst. Appl.* (2010), vol. 37, no. 3, 2486-2494.
26. Tseng, F. S., Kuo, Y.-H., and Huang, Y.-M. Toward Boosting Distributed Association Rule Mining By Data De-clustering. *Information Sciences* (2010), vol. 180, no. 22, 4263-4289.
27. Ozkural, E., Ucar, B., and Aykanat, C. Parallel Frequent Item Set Mining with Selective Item Replication. *IEEE Transactions on Parallel and Distributed Systems* (2011), vol. 22, no. 10, 1632-1640.
28. Vu, L. and Alaghband, G. A Fast Algorithm Combining FP-Tree and TID-List for Frequent Pattern Mining. In *Proc. the 2011 International Conference on Information and Knowledge Engineering* (2011).
29. Vu, L. and Alaghband, G. Mining Frequent Patterns Based on Data Characteristics. In *Proc. the 2012 International Conference on Information and Knowledge Engineering* (2012).
30. Vu, L. and Alaghband, G. An Efficient Approach for Mining Association Rules from Sparse and Dense Databases. In *Proc. the 2014 International Conference on Information and Knowledge Management, IEEE,*, (2014).
31. Vu, L. and Alaghband, G. Efficient Algorithms for Mining Frequent Patterns from Sparse and Dense Databases. *Intelligent Systems* (2014).
32. Pramudiono, I., and Kitsuregawa, M. Parallel FP-growth on PC Cluster. in *Proc. the 7th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining* (2003).
33. Sucahyo, Y., Gopalan, R. and Rudra, A. Efficiently Mining Frequent Patterns from Dense Datasets Using a Cluster of Computers. in *AI 2003: Advances in Artificial Intelligence*, vol. 2903, T. Gedeon and L. Fung, Eds., Springer Berlin Heidelberg,(2003), 233-244.
34. Zaki, M., Parthasarathy, S., Ogihara, M. and Li, W. New Algorithms for Fast Discovery of Association Rules. in *Proc. the 3rd International conference on Knowledge Discovery and Data Mining* (1997).
35. Jordan, L. E. and Alaghband, G. Fundamentals of Parallel Processing, Prentice Hall Professional Technical Reference (2002).
36. Han, J. Pei, J. and Yin, Y. Mining Frequent Patterns Without Candidate Generation. in *Proc. the 2000 ACM SIGMOD international conference on Management of data* (2000).
37. Grahne, G. and Zhu, J. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *Proc. the 2003 Workshop on Frequent Pattern Mining Implementations* (2003).
38. Racz, B. nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree. In *Proc. the 2004 Workshop on Frequent Pattern Mining Implementations* (2004).
39. Shporer, S. AIM2: Improved Implementation of AIM," in *Proc. the 2004 Workshop on Frequent Itemset Mining Implementations* (2004).
40. Schmidt-Thieme, L. Algorithmic Features of Eclat. In *Proceedings of the 2004 Workshop on Frequent Itemset Mining Implementations* (2004).
41. Frequent Itemset Mining Implementations Repository. In *Workshop on Frequent Itemset Mining Implementation*, 2003-2004.