International Journal of Computer Applications (0975 - 8887)
Volume 186 - No.36, August 2024

Robust Frequent Patterns Mining Algorithms on Parallel Systems

Sameh Abdulah
Department of Computer Science
Faculty of Computers and Information
Menoufia University

Walid Atwa
Department of Computer Science
Faculty of Computers and Information
Menoufia University

ABSTRACT

Data Mining (DM) algorithms have become increasingly prevalent in analyzing vast amounts of data generated in scientific fields like instrument and simulation data and in areas such as social networks and financial transactions. The availability of High-Performance Computing (HPC) systems has made parallel implementations of these algorithms commonplace. However, these systems, which were designed with data movement constraints in mind, often experience faults in computing devices, resulting in permanent process or node failures. This paper presents fault-tolerant parallel algorithms that enable checkpointing and recovery in memory for frequent pattern mining algorithms. Long-running data-intensive applications typically utilize the Message Passing Interface (MPI). Therefore, we tackle the challenge of fault tolerance in MPI-based applications by leveraging internal algorithm features and using MPI one-sided communication technology. Although this paper focuses on the FP-Growth frequent mining algorithm, we anticipate that the proposed approaches will serve as a foundation for designing fault-tolerant DM algorithms in general, given the effectiveness of the proposed implementations. Our evaluation demonstrates that MPI one-sided communication can act as a robust support system for efficient memory-based fault tolerance in parallel algorithms, even when compared to existing parallel programming models like Hadoop and Spark. To evaluate our fault-tolerant (FT) algorithms, we conduct tests on a large-scale InfiniBand cluster using several extensive datasets, employing up to 2K cores. Our evaluation reveals excellent efficiency in checkpointing and recovery compared to the disk-based approach. Furthermore, we observe an average speed-up of 20X for the FP-Growth algorithm compared to Spark. This establishes that a well-designed algorithm can easily surpass a solution based on a general fault-tolerant programming model.

General Terms

Message Passing Interface (MPI), One-side communication, fault tolerance

Keywords

MPI one-sided communication MPI-based applications fault tolerance algorithms FP-Growth Frequent mining algorithm.

1. INTRODUCTION

Current High-Performance Computing (HPC) systems can deliver immense computational power to support large-scale applications, and their computational capacity is increasing rapidly. For example, according to the latest Top500 list, two systems have surpassed the exaflop threshold, achieving 10^{18} floating-point operations per second. These systems are Frontier at Oak Ridge National Laboratory (ORNL) and Aurora at Argonne National Laboratory (ANL) [1]. This milestone will be achieved merely a decade after introducing the first petascale supercomputer, capable of performing 10^{15} floating-point operations per second. However, Supercomputing faces frequent hardware failures and high energy consumption, with system failures occurring within hours due to numerous cores. Research is now directed toward developing fault-tolerant technologies to address these issues.

In parallel systems, programming models serve as a means to develop many parallel applications in various domains. Several programming models, such as Hadoop [21] and Spark [24], have emphasized the significance of fault tolerance as a critical design consideration. Hadoop achieves fault tolerance by incorporating multiple replicas of data structures in permanent storage. The data is divided into chunks, each replicated on multiple nodes. The default replication factor is three, meaning each block is stored on three different nodes. This ensures that data can still be accessed from other nodes if one node fails. However, this approach introduces a substantial I/O overhead on the critical path and increases the required storage. On the other hand, Spark tackles this limitation by relying on Resilient Distributed Datasets (RDDs), enabling in-memory replication for fault tolerance. However, with large datasets, in-memory replication becomes impractical. Within the context of general-purpose programming systems, new techniques have emerged to address the issue of fault tolerance. One notable method is Scalable Checkpoint Restart (SCR) [8, 18, 19], which offers in-memory checkpointing for multi-level hierarchical file systems through non-blocking methods. SCR also permits the utilization of additional main memory to facilitate in-memory checkpointing/recovery. Similarly, researchers have proposed programming model/runtime extensions for Charm++ [1, 13] and X10 [9] to enhance fault tolerance support. While these approaches offer non-blocking checkpointing, they come with an inherent increase

[1] www.top500.org

in memory requirements as spare memory is needed for checkpointing purposes.

Analyzing substantial data volumes at each stage presents challenges related to space usage and I/O overhead. Algorithm-based Fault Tolerance (ABFT) emerges as a promising approach to address these issues. ABFT is a technique for enhancing system reliability and robustness by directly integrating fault tolerance mechanisms into the algorithms. This approach is particularly useful in high-performance computing, distributed systems, and critical applications where traditional hardware or system-level fault tolerance might be insufficient or inefficient. ABFT focuses on checkpointing critical data essential for recovering an algorithm in case of system failure, selectively excluding unnecessary data. Implementing an ABFT algorithm requires an in-depth understanding of the specific internal characteristics of the algorithm, making it generally applicable only to that particular algorithm rather than easily extendable to a group of algorithms [12]. Several ABFT algorithms have been proposed in the literature for different kinds of applications [5, 7, 22, 26].

Message Passing Interface (MPI) is frequently used in data-intensive HPC applications to build robust applications for HPC systems [10]. The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to function on various parallel computing architectures. It allows processes to communicate with each other by sending and receiving messages, making it a fundamental tool for developing parallel applications in high-performance computing (HPC) environments. This paper addresses the challenge of fault tolerance, specifically in MPI-based applications, focusing on well-known FP-growth pattern mining algorithms. Existing literature has seen the development of various fault tolerance (FT) algorithms to support data mining algorithms with robust fault tolerance techniques. Examples like VB-FT mine and FTP-mine provide internal recovery mechanisms for hardware failures, especially in frequent pattern mining [15]. However, these algorithms often prioritize fault tolerance over efficiency, impacting performance. Some efforts have also added fault tolerance support to the Apriori frequent pattern algorithm, although it is less commonly used in large-scale applications [23].

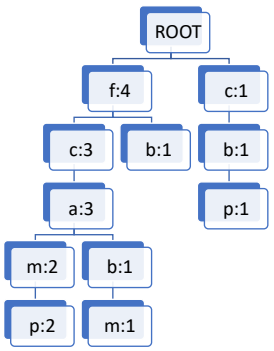
Our primary objective in this study is to overcome the limitations of disk I/O bottleneck and minimize the overhead associated with checkpointing. We propose leveraging the memory instead of relying on disk operations to achieve this. Our approach revolves around the utilization of Algorithmic Based Fault Tolerance (ABFT) techniques [26], coupled with advanced MPI features, specifically one-sided communication (MPI-RMA), where a process to directly access the memory of another process without the involvement of the remote process. By incorporating these elements, we aim to construct robust fault tolerance systems for the targeted algorithm.

2. PRELIMINARIES

This study is dedicated to developing parallel fault-tolerant solutions for the FP-Growth frequent pattern mining algorithm. FP-Growth is a critical algorithm in data mining, widely used for discovering frequent patterns in large datasets. When it comes to parallel execution and large datasets, ensuring fault tolerance in the FP-Growth algorithm is essential to maintain reliability and efficiency, especially in distributed computing environments where node failures and communication errors can disrupt the mining process. In the following subsections, we provide a concise overview of the algorithms to facilitate a better understanding of their functionality.

2.1 FP-Growth Algorithm

Frequent itemsets are defined as itemsets with frequencies exceeding a user-defined threshold value, denoted as θ [20]. To extract such items from a given dataset, various Frequent Pattern Mining (FPM) algorithms have been introduced in the literature, including Apriori [3], Eclat [25], GenMax [11], and FP-Growth [6] algorithms. FP-Growth is commonly employed among these algorithms in large-scale domains because it can retrieve all the frequent itemsets with just two passes over the dataset. FP-Growth relies on a compressed structure known as the FP-Tree (FPT), allowing for efficient in-memory processing of large datasets while maintaining a smaller memory footprint than other existing algorithms. In the first pass of the FP-Growth algorithm, frequent items are identified. An FP-Tree (FPT) is generated in the second pass to represent the frequent itemsets that meet the specified threshold θ. Figure 1 illustrates an example of the FP-Tree generation process. The left subfigure shows an example of an FP-Tree, while the right subfigure demonstrates how to extract the frequent itemsets from the given FP-Tree. Specifically, creating the FP-Tree during the second pass is the most time-consuming part of the computation. Thus, this study focuses on developing a fault-tolerant approach for the FP-Tree creation step, as the extended execution time increases the likelihood of encountering faults.



Transaction	Ordered by Freq. items
f,a,c,d,g,l,m,p	f,c,a,m,p
a,b,c,f,l,m,o	f,c,a,b,m
b,f,h,j,o	f,b
b,c,k,s,p	c,b,p
a,f,c,e,l,p,m,n	f,c,a,m,p

Fig. 1: An illustrative depiction of the FP-Tree generation phase within the FP-Growth algorithm.

2.2 Challenges in Large-Scale Fault-Tolerant Programming Models

Recently, there has been a significant rise in the popularity of large-scale and FT programming models like Hadoop and Spark [14]. Programming models rely on a single assignment, where every variable mutation is tracked, stored (in the memory or permanent storage of another node), and replayed in the event of a fault. For instance, when inserting an element into an existing FP-Tree, each modification or mutation must be locally recorded and eventually persisted to permanent storage. By exploring the implications of this framework for an algorithm like FP-Tree, in many cases, saving a new FP-Tree version on disk occurs at the end of the Reduce phase in the MapReduce programming model. However, this does not provide a significant advantage in two-phase algorithms like FP-Growth, where most computation is in the second phase. An alternative is to split the calculation into multiple MapReduce steps, taking checkpoints after each Reduce phase. However, this

approach increases total execution time, as saving a new version requires disk writes or using memory from neighboring nodes when designed for distributed memory systems. Since the Reduce phase is blocking, this leads to significant overhead and degrades overall performance. Ideally, a scalable algorithm should prioritize native execution for optimal performance while minimizing checkpointing costs through non-blocking methods.

2.3 Leveraging MPI-RMA for Fault-Tolerant Data Mining

This study leverages the Message Passing Interface (MPI), a mainstay on supercomputers and clusters that is now gaining traction in cloud environments. MPI-RMA provides a powerful and efficient mechanism for one-sided communication in parallel and distributed computing environments. Allowing direct memory access across processes can lead to significant performance improvements and more flexible communication patterns. Understanding and leveraging MPI-RMA can be crucial for developing high-performance parallel applications. Despite past criticisms regarding its fault tolerance capabilities, recent advancements and literature suggest MPI now adequately handles permanent process faults. Furthermore, the introduction of MPI One-sided communication primitives, also known as MPI-Remote Memory Access (MPI-RMA), provides valuable means for integrating communication and computation.
MPI-RMA operations fall into three categories: window creation, RMA operations (like `MPI_Get` and `MPI_Put`), and synchronization. These operations allow asynchronous access to specific memory regions during "epochs" - periods between the opening and closing of a window. Modern MPI updates have brought in dynamic windows, which enable the flexible allocation of memory across remote processes, a feature advantageous for tasks that involve work-stealing and enhancing fault tolerance. We aim to harness these MPI capabilities to develop fault-tolerant data mining algorithms.

3. PROPOSED FAULT TOLERANT ALGORITHM

This section explores strategies for designing a fault-tolerant (FT) FP-Growth algorithm in a fail-stop scenario. We prioritize continued execution over re-spawning new processes on spare nodes due to its simplicity and smoother recovery process, reducing reliance on additional software components. In this study, we introduce an Asynchronous Memory-based Fault Tolerant (AMFT) approach to provide fault tolerance support for the FP-Growth algorithm.
Algorithm 1 outlines the key steps of the parallel FP-Growth algorithm, which serves as the baseline for designing fault-tolerant FP-Growth algorithms. The process begins by distributing the input database transactions across all $|P|$ processes (Line 3). Each process independently constructs its own local FP-Tree. In this step, each process (p_i) scans its assigned transactions to calculate the frequency of each item (Line 4). To obtain the global frequency of items, an all-to-all reduction operation is performed using `MPI_Allreduce`, with a time complexity of $\log(|P|)$ (Line 5). Following this reduction, only the items that meet or exceed the support threshold are retained. Each process p_i then constructs a local FP-Tree ($L.Tree$) using its transactions that contain at least one frequent item (Line 6). These local FP-Trees are subsequently merged into a global FP-Tree ($G.Tree$) through a ring communication algorithm (Line 7). Finally, the frequent itemsets ($FreqItemSet$) are extracted from this global FP-Tree (Line 8).

Algorithm 1 : Parallel FP-Growth Algorithm

1: **Input:** Global Transactions S, Support threshold θ
2: **Output:** Set of frequent itemsets
3: L.Trans ← getLocalTrans(S)
4: L.FreqList ← findLocalFreqItems($L.Trans$, θ)
5: G.FreqList ← MPI_Allreduce(L.FreqList)
6: L.Tree ← generateLocalFPTree($L.Trans$, $G.FreqList$)
7: G.Tree ← generateGlobalFPTree($L.Tree$)
8: FreqItemSet ← miningGFPTree($G.Tree$)

To simplify the understanding of the following subsections, Table 1 provides the symbols used to represent the time and space complexity of our proposed FT algorithm.

Table 1. : Symbols used for time-space complexity modeling.

Name	Symbol
Process Set	$P = \{p_0, \ldots, p_{\|P\|-1}\}$
Transaction Database	$T = \{t_0, \ldots, t_{\|T\|-1}\}$
Average Local Transaction Size	t_{avg}
Minimum Support Threshold	θ
Local FP-Tree Set	$S = \{s_0, \ldots, s_{\|P\|-1}\}$
Average Local FP-Tree Size	s_{avg}
Average Time to Merge Two FP-Trees	m
Number of Checkpoints	C
Disk Access Bandwidth	l
Network Bandwidth	b

3.1 Asynchronous Memory-based Fault Tolerant (AMFT) FP-Growth Algorithm

Generating FP-trees is the most demanding part of the FP-growth algorithm, with each process building an individual FP-tree from its transaction subset. To safeguard against failures, these FP-trees must be saved at intervals on permanent storage. Traditional disk checkpointing, while common, is not as efficient as our proposed fault-tolerant FP-Growth algorithm that mainly relies on MPI-RMA technology for in-memory storage of these crucial structures, enhancing both efficiency and fault tolerance. We introduce the Asynchronous Memory-based Fault Tolerance (AMFT) approach, which checkspoints in memory without increasing space complexity. It uses the memory allocated to processed transactions to save intermediate states, avoiding the need for process synchronization during checkpointing. By employing non-blocking MPI primitives, we overlap checkpointing of FP-trees with transaction processing, thus achieving efficient and asynchronous fault tolerance that minimally affects performance. This overlapping strategy enables the concurrent execution of communication and computation in distributed memory systems, which helps accelerate the checkpointing process and conceal its complexity. We summarize the checkpointing and recovery algorithms for the FP-Growth algorithm in the following subsections.

3.1.1 Checkpointing Algorithm. In a system with a set of processes P, assume two processes, p_i and p_{target}. Within our checkpointing framework, p_i saves its checkpoint data onto p_{target}. To achieve a one-sided checkpointing mechanism, p_i must ensure its checkpoint is compact enough to fit into the space allocated by the processed transactions in p_{target}, which optimizes checkpointing efficiency and reduces overhead. The AMFT method enables this

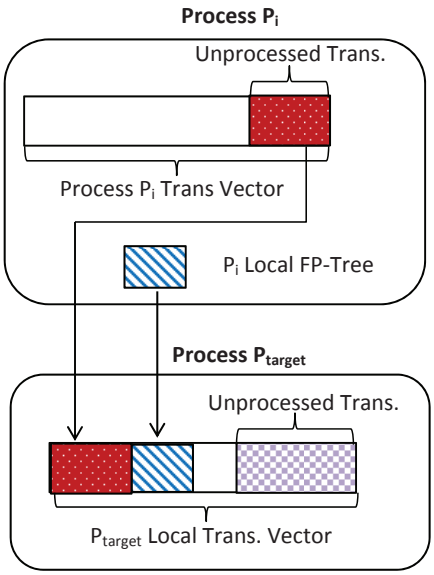


Fig. 2: A snapshot of an AMFT FP-Growth algorithm checkpoint: blue shows the local FP-tree of the process P_i, red indicates the unprocessed transactions of the process P_i, and purple highlights the unprocessed transactions of P_{target}.

using an atomic variable created through MPI-RMA, accessible by other processes for read and update actions. p_{target} has the sole task of atomically updating this variable to reflect the space available for checkpointing, which does not require interaction with other processes. When p_i is ready to checkpoint its FP-Tree, it atomically checks the variable of p_{target} variable to confirm available space, facilitating a smooth, communication-free memory resource coordination for checkpointing. In extreme cases, p_i might have to verify the space until it has sufficient periodically, but this is a rare occurrence. Normally, p_i proceeds with the checkpoint using `MPI_Put`. This protocol includes saving the local FP-Tree and potentially unprocessed transactions from p_i to memory of p_{target}. Checkpointing transactions just once improves recovery by allowing immediate access to the data of the failed process from checkpoint memory, bypassing slower disk reads. This boosts recovery speed and overall system efficiency. Figure 2 depicts the AMFT checkpointing method, showcasing two scenarios. In one, only the local FP-Tree of a process p_i is checkpointed into the transaction space of p_{target}. In the other, both the remaining transactions and the local FP-Tree of p_i are checkpointed into the memory of p_{target}, space permitting. These instances highlight how the AMFT checkpointing approach can adapt to various data-saving needs.
The simplicity of the AMFT checkpointing algorithm is a critical factor in its effectiveness. By relying on MPI-RMA on high-performance interconnects like InfiniBand, Slingshot, and NVLink, we anticipate AMFT as an almost optimal checkpointing solution for large-scale FP-Growth algorithms. As anticipated, each process only needs to initiate the communication for the checkpointing process, resulting in an expected time complexity of approximately,

$$O\left(\frac{|T|}{\log|P|\cdot C}\right)$$

, as per the LogGP communication performance model [4]. This time complexity demonstrates the efficiency and scalability of the AMFT checkpointing algorithm in handling large-scale datasets.

3.1.2 Recovery Algorithm. To illustrate the FP-Growth recovery algorithm in the case of a failure, we consider the following scenario. If p_{target} is the process chosen for recovery, also referred to as p_{rec}, then upon a failure in process p_i, p_{rec} undertakes the following recovery sequence: It integrates the checkpointed FP-Tree pf the process p_i with its own; should an in-memory checkpoint exist locally, p_{rec} reassigns unfinished transactions of p_i to a subset of working processes, often $\log|P|$ in number; if no in-memory checkpoint, then all operational processes join forces to retrieve the unfinished transactions of p_i from the disk. Adopting this recovery strategy enables the system to rebound from process failures using in-memory checkpoints alongside parallel disk retrieval, thereby maintaining the operation of the FP-Growth algorithm. In the direst situation, recovery entails reading the full transaction set from the disk simultaneously, which has a time complexity of

$$O\left(\frac{|T|}{|P|\cdot(|P|-1)\cdot l}\right)$$

The recomputation is then carried out by $\log|P|$ processes, resulting in a complexity of

$$O\left(\frac{|T|}{|P|\cdot\log|P|}\right)$$

It is worth mentioning that disk reads are often unnecessary, especially if a fault arises late in the FP-Tree construction phase, leading to a substantially swifter recovery than the worst-case analysis suggests, thereby greatly diminishing the time needed for system restoration.

4. PERFORMANCE EVALUATION

This section provides a detailed performance evaluation of the proposed fault-tolerant system for the FP-Growth algorithm discussed in Section 3. We analyze the overhead associated with checkpointing and recovery operations and compare our MPI-RMA-based approaches with the built-in fault-tolerant support of Spark and Hadoop to demonstrate their efficacy.

4.1 Experimental Testbed

For performance testing, we conducted evaluations on 6,400 Dell PowerEdge servers, each equipped with 32GB of RAM and dual 8-core Intel Xeon E5 (Sandy Bridge) processors. We rely on the MVAPICH2-3.7 MPI library, optimized for RDMA, leveraging high-speed InfiniBand interconnects. Additionally, we apply aggressive optimizations using the Intel compiler to enhance performance. Our study simulates permanent process failures by arbitrarily selecting a point during execution to "fail" a process, effectively rendering it offline. We focus on simulating a single process failure, the most common failure mode in fault-tolerant systems.
We evaluate the proposed asynchronous checkpointing/recovery algorithm against two other techniques: Disk-based Fault Tolerant (DFT) and Synchronous Memory-based Fault Tolerant (SMFT). In the FP-Growth algorithm, both database transactions and intermediate FP-Trees are crucial for recovery. The DFT approach periodically saves intermediate FP-Trees to disk, which limits scalability due to its reliance on disk-based checkpointing, making memory-

based alternatives more appealing. SMFT overcomes these limitations by maintaining constant space complexity and utilizing non-blocking MPI primitives for efficient checkpointing and recovery. It avoids disk I/O, minimizing overhead by distributing transactions from failed processes to active ones and using parallel disk access when necessary. However, SMFT has two main drawbacks: it requires synchronization between processes to share checkpoint information and incurs overhead from de-allocating and reallocating memory for checkpointing.

4.2 Overhead of FP-Growth Fault Tolerance

To assess our fault-tolerant FP-Growth algorithms, we use the IBM Quest dataset generator [2], known for creating realistic, large-scale synthetic datasets resembling real-world transaction patterns [16, 17]. We work with two synthetic datasets: one with 100 million transactions and another with 200 million transactions. Each transaction contains 15-20 items, using 1,000 unique item IDs.

4.2.1 Checkpointing Overhead Evaluation. Table 2 shows the checkpointing overhead in fault-tolerant FP-growth algorithms—DFT, SMFT, and AMFT—on datasets with 100M and 200M transactions at support thresholds of 0.03 and 0.05. We assess these algorithms against a fault-tolerant baseline, noting that both the baseline and our algorithms (DFT, SMFT, and AMFT) scale efficiently, achieving super-linear speedup at 256-512 processes due to better cache utilization. At a 0.05 support threshold, processing time for some frequent itemsets is under 50 seconds. Small tests with 256 processes handling 100M transactions show DFT and SMFT slowing down by 67% and 31%, respectively, with AMFT at a lower 21%, attributed to increased FP-tree sizes and fewer workers. This is because MPI-RMA is not efficient for large data transfers. Conversely, AMFT running on 2048 cores with 100M transactions has a mere 5% overhead due to checkpointing. When the support threshold is reduced, AMFT's slowdown stays within 4-6%, whereas DFT varies between 10-20%, depending on the number of processes.

4.2.2 Recovery Overhead Evaluation. To assess the impact on recovery time following deliberate process failures, we instigate process crashes after 80% of the dataset transactions have been processed. This provides an even basis for comparing the recovery performance of DFT, SMFT, and AMFT approaches. We concentrate on how much faster AMFT and SMFT can recover from a single failure relative to DFT. The data illustrated in Figure 3 indicate that with a 0.05 support threshold and a 100M dataset, AMFT registers a recovery speedup of 1.41X on average, and SMFT records 1.36X. When tested on a larger, 200M synthetic dataset, AMFT and SMFT decrease the total execution time by 1.59X and 1.55X, respectively. In Figures 3b and 3d, with a support threshold of 0.03, we note that the recovered FP-Tree size increases, impacting DFT performance compared to SMFT and AMFT. Using a 100M dataset, AMFT achieves a 1.46X recovery speedup over DFT, and SMFT achieves 1.39X. With a 200M dataset, AMFT speeds up algorithm execution with recovery by 1.68X, and SMFT achieves a 1.51X speedup.
Figure 3 offers significant findings, indicating that the SMFT and AMFT algorithms enable quicker FP-Growth recovery than DFT. With a lower support threshold, such as $\theta = 0.05$, the smaller size of checkpointed and recovered FP-Trees highlights the synchronization costs in SMFT, with AMFT showing superior performance. As the threshold increases to $\theta = 0.03$, leading to larger FP-Trees, the relative impact of synchronization declines, making the speed advantages of SMFT and AMFT over DFT less pronounced. Furthermore, these algorithms excel with larger datasets, like those with 200M transactions, as the expanded FP-Trees enhance speedups. DFT, in contrast, incurs longer checkpointing and recovery times from disk. Notably, a super-linear speedup is observed with a higher threshold of $\theta = 0.3$ when expanding from 256 to 512 cores, a benefit primarily due to better cache usage.

4.3 Comparison Against Spark

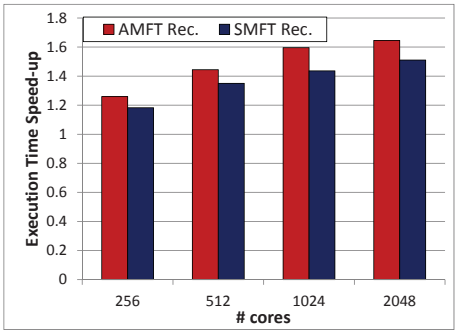
We compare the overhead of handling failures in MPI-based implementations, which typically surpass MapReduce ones. For benchmarking, we use Spark's MLlib, which includes FP-Growth implementations, as our comparison standard. Figure 4 shows the AMFT algorithm's performance relative to Spark. AMFT is faster than Spark, averaging 20X quicker for $\theta = 0.01$ and 8.6X for $\theta = 0.03$ when there are no failures. This increase in speed, especially at the lower $\theta = 0.01$ threshold, is attributed to the larger FP-Trees that need checkpointing. AMFT also scales better during checkpointing compared to Spark. It periodically checkpoints only the FP-Trees and a small set of transactions, which remain compact even as the core count increases. Spark, however, uses the RDD mechanism that replicates FP-Trees and transactions in memory, with replication overhead growing as more cores are added. In failure scenarios, AMFT achieves an average speed-up of 15.3X for $\theta = 0.01$ and 8.34X for $\theta = 0.03$ over Spark. Both AMFT and Spark's performances benefit from a larger core count and/or smaller support thresholds like $\theta = 0.03$, since they result in a smaller FP-Tree to recover.

5. CONCLUSION

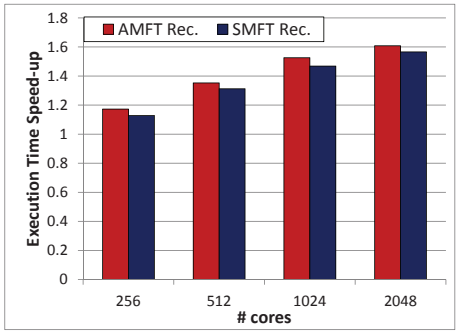
This paper presents a comprehensive study on designing and evaluating fault-tolerant solutions for the FP-Growth algorithm, specifically tailored for high-performance computing (HPC) environments. Leveraging the capabilities of MPI one-sided communication (MPI-RMA), we developed the Asynchronous Memory-based Fault Tolerant (AMFT) FP-Growth algorithm, which significantly enhances efficiency and robustness compared to traditional disk-based checkpointing methods.
Our key findings are: first, the AMFT algorithm achieves significant speedups in checkpointing and recovery processes, with up to 20x improvement over Spark for lower support thresholds. Second, by focusing on a single process failure, we effectively demonstrated the robustness and efficiency of our fault-tolerant mechanisms, which are crucial for maintaining the reliability of data mining operations in distributed environments. Third, our approach minimizes the overhead associated with fault tolerance by utilizing in-memory checkpointing and non-blocking MPI primitives, ensuring that system performance remains high despite failures. The detailed performance evaluation highlights the advantages of integrating fault tolerance directly into the algorithmic design, particularly through the use of MPI one-sided communication. This integration allows for seamless memory management and efficient resource utilization, making it a viable solution for large-scale data mining applications. Our tests show that AMFT reduces overhead and speeds up performance by up to 20X for $\theta = 0.01$ and 8.6X for $\theta = 0.03$ over Spark.
In conclusion, our research demonstrates that leveraging MPI-RMA and algorithm-based fault tolerance techniques can significantly enhance the performance and reliability of parallel data mining algorithms. The AMFT FP-Growth algorithm serves as a robust model for future developments in fault-tolerant computing, offering valuable insights and methodologies that can be applied to a wide

Table 2. : Baseline FP-Growth, DFT, AMFT, and SMFT systems execution time without failures.

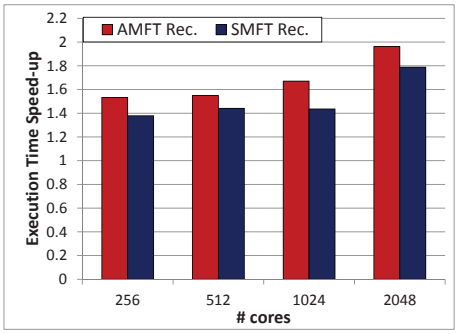
# Cores	Sup.	Baseline FP-growth		DFT		SMFT		AMFT	
		100M	200M	100M	200M	100M	200M	100M	200M
256	0.03	1628.79	5751.64	1950.73	7799.13	1805.55	6455.58	1727.93	6350.59
	0.05	35.94	79.15	50.74	133.67	47.09	111.45	43.71	102.18
512	0.03	549.94	1811.63	634.03	2280.31	603.63	1993.09	581.10	1980.59
	0.05	14.68	36.25	24.56	57.31	19.05	50.31	17.39	45.53
1024	0.03	319.32	838.91	363.30	1025.34	338.66	935.59	333.60	915.59
	0.05	7.06	19.32	10.88	29.56	9.03	27.30	7.92	22.96
2048	0.03	238.88	531.87	263.09	625.30	254.80	578.08	248.95	558.09
	0.05	5.32	10.91	6.78	14.80	6.16	14.20	5.59	11.99



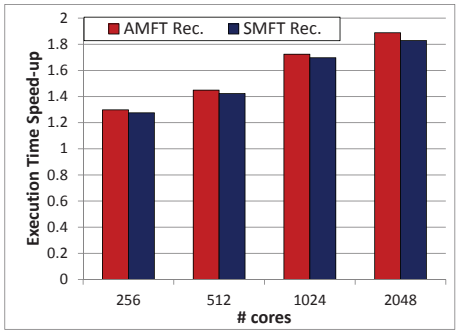
(a) 100M Trans. θ=0.05



(b) 100M Trans. θ=0.03

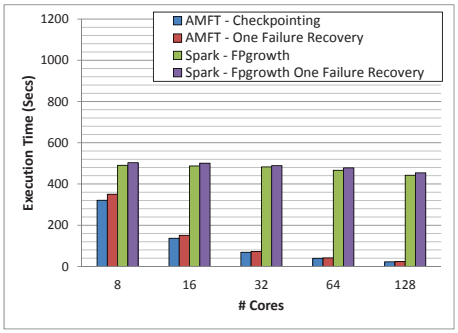


(c) 200M Trans. θ=0.05

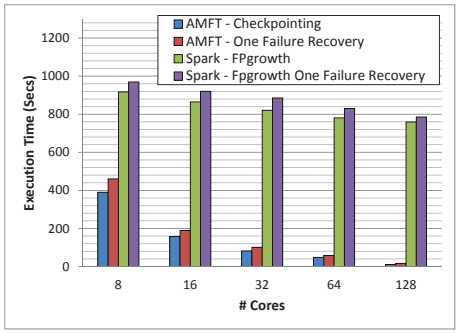


(d) 200M Trans. θ=0.03

Fig. 3: SMFT and AMFT recovery speedup compared to DFT approach with different number of transactions, support threshold, and cores.



(a) 500K Trans. θ=0.03



(b) 500K Trans. θ=0.01

Fig. 4: Spark and MPI-based (AMFT) with Different Support Threshold θ and using 500K Synthetic Dataset

range of data-intensive applications. Future work will focus on further optimizing these mechanisms and exploring their applicability to other data mining algorithms and HPC systems.

6. REFERENCES

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, et al. Parallel programming with migratable objects: Charm++ in practice. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658. IEEE, 2014.

[2] R Agrawal and R Srikant. Quest synthetic data generator. ibm almaden research center, san jose, california, 2009.

[3] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM, 1993.

[4] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, 1995.

[5] Chong Bao and Shancong Zhang. Algorithm-based fault tolerance for discrete wavelet transform implemented on gpus. *Journal of systems architecture*, 108:101823, 2020.

[6] Christian Borgelt. An implementation of the fp-growth algorithm. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 1–5. ACM, 2005.

[7] Claus Braun, Sebastian Halder, and Hans Joachim Wunderlich. A-abft: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 443–454. IEEE, 2014.

[8] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K Panda, Hari Subramoni, Jérôme Vienne, and Gene Cooperman. System-level scalable checkpoint-restart for petascale computing. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 932–941. IEEE, 2016.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[10] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[11] Karam Gouda and Mohammed J Zaki. Genmax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3):223–242, 2005.

[12] Upama Kabir and Dhrubajyoti Goswami. An abft scheme based on communication characteristics. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 515–523. IEEE, 2016.

[13] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[14] Khushboo Kalia and Neeraj Gupta. Analysis of hadoop mapreduce scheduling in heterogeneous environment. *Ain Shams Engineering Journal*, 12(1):1101–1110, 2021.

[15] Jia-Ling Koh and Pei-Wy Yo. An efficient approach for mining fault-tolerant frequent patterns based on bit vector representations. In *DASFAA*, volume 5, pages 568–575. Springer, 2005.

[16] Jay Lewis, Ryan G Benton, David Bourrie, and Jennifer Lavergne. Enhancing itemset tree rules and performance. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1143–1150. IEEE, 2019.

[17] Wei-Tee Lin and Chih-Ping Chu. Determining the appropriate number of nodes for fast mining of frequent patterns in distributed computing environments. *International Journal of Parallel, Emergent and Distributed Systems*, (ahead-of-print):1–13, 2014.

[18] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[19] Faisal Shahzad, Markus Wittmann, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. A survey of checkpoint/restart techniques on distributed memory systems. *Parallel Processing Letters*, 23(04):1340011, 2013.

[20] Mai Shawkat, Mahmoud Badawi, Sally El-ghamrawy, Reham Arnous, and Ali El-desoky. An optimized fp-growth algorithm for discovery of association rules. *The Journal of Supercomputing*, pages 1–28, 2022.

[21] Kamalpreet Singh and Ravinder Kaur. Hadoop: addressing challenges of big data. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 686–689. IEEE, 2014.

[22] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 31–42, 2016.

[23] Cheng Yang, Usama Fayyad, and Paul S Bradley. Efficient discovery of error-tolerant frequent itemsets in high dimensions. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 194–203. ACM, 2001.

[24] Matei Zaharia and et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

[25] Mohammed Javeed Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

[26] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1677–1689, 2020.