A Fast Parallel Implementation of Apriori Algorithm on AiMOS using MPI, CUDA, and Parallel I/O

Md. Shamim Hussain†
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York
hussam4@rpi.edu

Nafis Neehal
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York
neehan@rpi.edu

Sharmishtha Dutta
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York
duttas@rpi.edu

ABSTRACT

For this project, we implemented the Apriori algorithm for Frequent pattern mining on the DCS cluster of AiMOS using CUDA and MPI. The most computationally expensive part of the Apriori algorithm is scanning over the whole dataset at each level. This process can be massively parallelized since it requires performing independent subset operations on different datapoints for different candidate patterns. To implement this algorithm in CUDA used the bitset representations of the itemsets which allowed us to efficiently perform operations like subset and set intersection using simple bitwise operations which are very fast on both CPU and GPU. Each thread on the GPU is assigned to perform subset operation for each candidate pattern, allowing for simultaneous scanning of the datasets for all candidate patterns. On the advent of multiple MPI ranks and GPUs on AiMOS, we partitioned the dataset and allocated different parts to different MPI ranks. After computing support within each rank, the total support was calculated by adding up supports found by other ranks – which was performed by the MPI all-reduce operation. To facilitate the reading of large datasets, the data file is read in sequential blocks parallelly by each MPI rank – using MPI read. To make the output file write more efficient MPI write was used.

We have verified the scalability of this implementation both for massively parallel systems and for large datasets by extensive experiments. We have evaluated the performance of different parts of the program and identified potential limiting factors. The current implementation which can be further improved, still achieves admirable results on benchmark datasets.

KEYWORDS

Frequent Pattern Mining, Apriori Algorithm, Parallel Computing, MPI, CUDA, Parallel I/O

1 Introduction

1.1 Frequent Pattern Mining

Frequent Pattern Mining is an interesting problem in many applications where the idea is to determine the frequency of two or more objects of interest co-occur. If one finds the frequent objects, they can further analyze the patterns and determine the association rule that exists in the dataset they are working on. Association rules are statements about the statistical relevance of whether some objects frequently co-occur or conditionally occur.

Frequent pattern mining aims at finding patterns that occur frequently in a dataset of multiple transactions; and also, their corresponding frequency - or "support". The support of an itemset in a dataset is defined as the number of transactions in which a given set of items appear as a subset. A frequent pattern is defined as an itemset that has a support greater than a given threshold value (called minimum support or *minsup*).

Frequent Pattern Mining, specifically Association Rule Mining became very popular in 1993 when the Apriori algorithm was published (has more than 22000 citations according to Google Scholar as of May 2020) by Agarwal et al [21]. Since then, there have been numerous studies on this area and many algorithms have also been proposed.

Some of the most popular frequent pattern mining algorithms are Equivalence CLAss Transformation (ECLAT) algorithm [4], Rapid Association Rule mining (RARM) [5], FPGrowth algorithm etc. While considering parallel implementations of these algorithms several factors come into play. ECLAT requires the allocation and release of virtual memory within threads. FPGrowth requires an irregular memory access pattern. Therefore, GPUs cannot be used most efficiently in FPGrowth. RARM is difficult to implement in an interactive system. Among these, only Apriori algorithm provides an opportunity to employ uniform parallelism by using the best resources of AiMOS supercomputer. Also, it performs excellently in larger datasets. Therefore, we have chosen to implement Apriori algorithm for this project.

1.2 Apriori Algorithm

The Apriori algorithm is a breadth-first (level-wise) approach to finding frequent itemsets which utilizes two facts - if an itemset is frequent, all possible subsets of the itemset must also be frequent and if an itemset is not frequent, any superset of that itemset cannot be frequent. At level 1 it looks for all frequent patterns of length 1.

It scans the whole dataset once per level. At each level, it takes four steps. The first step is to generate a candidate itemset based on the frequent itemsets found in the previous level.

Of the four stages of the Apriori algorithm, the most computationally expensive one is the scanning of the whole dataset for support computation at each level/iteration. This process also has the potential to be massively parallelized

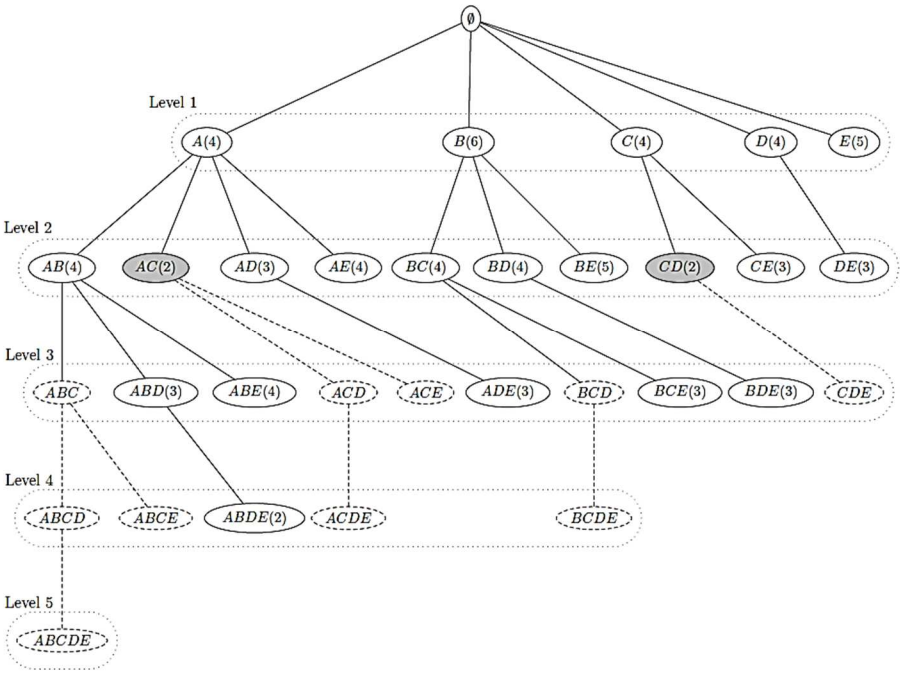

Fig 1. Apriori Algorithm [22]

The second step is to prune those itemsets that have infrequent subsets. As the third step, the supports of the valid candidate itemsets are computed by scanning over the dataset. Finally, the candidate itemsets whose supports are less than the minimum supports are removed, and the rest are saved as frequent itemsets.

1.3 Motivation

We wanted to implement a data mining algorithm in GPU. In most of these algorithms the threads are divergent and perform different tasks. Such algorithms are not suitable for GPU as they reduce the instruction throughput and performance. GPUs are most suitable in performing vectorized operations where all threads perform mostly the same operation on different parts of the data. Additionally, we looked for a bandwidth-bound problem (which is dependent on how fast the read is). These problems are also most suitable for a GPU implementation - as GPUs have high memory bandwidths. Algorithms like ECLAT and FP Tree, although parallelizable – perform vastly different operations in different threads and/or have very irregular memory accesses – which is not suitable for a GPU implementation.

for two reasons – firstly, it requires the same operation, i.e. subset identification on each data point which can be performed by multiple threads independently in parallel, and secondly, it requires reading the dataset in a regular manner which can maximize bandwidth. If the subset identification operation can be done efficiently this problem becomes completely bandwidth bound. For these reasons, we have found this algorithm suitable for implementation with CUDA on GPU and also with MPI across multiple processes and nodes.

1.4 Key Contributions

We aimed at an implementation of the Apriori algorithm which achieves a high level of efficiency on the DCS subsystem AiMOS system – which has multiple GPUs per node – so our focus was to get the most out of the compute capability of the GPUs. To achieve this, we have taken some optimization steps – as discussed below.

We have assigned the costliest ‘support computation’ part of the original sequential algorithm to the GPU threads. All threads have been assigned the same task – identification of subsets in different data points of the dataset.

To make the subset identification even more efficient and simplify the problem even further, we have adopted bitset representation of the itemsets – where if an item is present its corresponding bit is set. This has also facilitated the simplification of the candidate generation portion of the problem.

Our implementation included the use of CUDA on GPU, and MPI and parallel I/O. MPI was used to communicate among processes – each of which in turn performs computations both serially and in parallel on the GPU. MPI I/O was used to enable parallel read of potentially large datasets which are not possible to be loaded into the memory, and to write back numerous mined frequent patterns to the disk efficiently.

The vectorized implementation of support computation in the GPU achieves a massive boost for the Apriori algorithm. In the best-case scenario – when overhead from other steps is low, it can achieve linear scaling with the number of GPU threads.

1.5 Paper Outline

The paper is divided into the following sections – Section 2 describes the related works to our work in detail. Section 3 will discuss the background of the Apriori algorithm in depth. Section 4 will illustrate our methodology in detail. Section 5 will contain detailed experimental details and result analysis. Section 6 will summarize and enumerate the key points of our work again. And finally, Section 7 will deliver the concluding remarks and references and necessary appendices (if any) will follow.

2 Literature Review

2.1 Frequent Pattern Mining (FPM) algorithms

The concept of pattern mining in the form of market basket analysis using association rules mining was first proposed by Agrawal et al. (1993). This concept used transactional databases and other data repositories to extract association's casual structures, interesting correlations, or frequent patterns. The Apriori algorithm [21] is the most widely used in the history of association rule mining that uses an efficient candidate generation process, such that large Itemset generated at k level is used to generate candidates at k+1 level. On the other hand, it scans the database multiple times as long as large frequent Itemsets are generated. Apriori TID generates candidate Itemset before the database is scanned with the help of the Apriori-gen function. Database is scanned only the first time to count support, rather than scanning the database it scans candidate Itemset. This variation of Apriori performs well at a higher level whereas the conventional Apriori performs better at lower levels [2]. Apriori Hybrid is a combination of both the Apriori and Apriori TID. It uses Apriori TID in later passes of the database as it outperforms at high levels and Apriori in the first few passes of the database.

Equivalence CLAss Transformation (**ECLAT**) algorithm [4] proposed by Zaki et al (1993) uses vertical database format whereas in Apriori and RARM horizontal data format (*TranscationId*, Items) has been used, in which transaction ids are explicitly listed. While in vertical data format (Items, *TransactionId*) Items with their list of transactions are maintained. ECLAT algorithm with set intersection property uses the depth-first search algorithm [3]. ECLAT requires the virtual memory to perform the transformation.

Rapid Association Rule mining (**RARM**) proposed in [5] generates Large 1- Itemset and large 2-Itemset by using a tree Structure called SOTrieIT and without scanning database. It also avoids complex candidate generation processes for large 1-Itemset and Large 2-Itemset that was the main bottleneck in the Apriori Algorithm. RARM is an algorithm that avoids complex candidate generation process. By using a novel tree data structure known as Support-Ordered Trie Itemset (SOTrieIT), RARM accelerates the mining process even at low support thresholds [5]. RARM is difficult to use in interactive system mining. It is also difficult to use in incremental Mining.

Another accomplishment in the development of association rule mining and frequent pattern mining is FPGrowth Algorithm. Efficiency of FP-Growth is based on three salient features: (1) A divide-and-conquer approach is used to extract small patterns by decomposing the mining problem into a set of smaller problems in conditional databases, which consequently reduces the search space (2) FP-Growth algorithm avoid the complex Candidate Itemset generation process for a large number of candidate Itemsets, and (3) To avoid expensive and repetitive database scan, the database is compressed in a highly summarized, much smaller data structure called FP tree [6]. FP-Tree is expensive to build and consumes more memory.

The performance of any algorithm can be estimated by the number of required database scan to extract patterns. The storage consumption of different algorithms can be assessed for their utilization of memory to generate less candidate Itemset or avoid candidate Itemset generation process [7]. One disadvantage of the Apriori algorithm (sequential) is that it generates too many Candidate Itemsets. Also, it needs too many passes over the database. Thus, it requires large memory space. But when one makes a parallel implementation, these shortcomings can be overcome, and an excellent performance speedup is achieved. Also, this algorithm uses large itemset property

which makes it an idea to use in large datasets. It is easy to understand and implement.

2.2 Parallel Apriori Implementation

Chiang et. al. [8] developed a parallel version of the Apriori algorithm and that was developed on Bodon et.al.'s implementation. Their implementation was a partition based Apriori algorithm that partitions a transaction database into N partitions and then distributes the N partitions to N nodes where each node computes its local candidate k-itemsets from its partition The implementation was tested on the SGI® Altix® system [9], namely Cobalt that consists of 1,024 Intel® Itanium® 2 processors running the Linux® operating system, 3 terabytes of globally accessible memory, and 370 terabytes of SGI® Infinite Storage that serves as the Center's shared file system. There was no significant result comparison, mentioned, except only one work of Borgelt et.al.

Kun-Ming Yu et. al. [9] in their paper presented the Weighted Distributed Parallel Apriori algorithm (WDPA) as a solution for prior problems related to imperfect load balancing in parallel implementation of Apriori. In the proposed method, a database had only to be scanned once because metadata was stored in TID tables. This approach also considered the TID count. Therefore, WDPA improved load-balancing as well as reduces the idle time of processors. The experimental results in this study showed that the running time of WDPA was significantly faster than that of previous methods. In some cases, WDPA only used about 2% of the time used in previous methods. WDPA was implemented along with the algorithm proposed by Chiang et.al [8]. The program was executed in a PC cluster with 16 computing processors. CPU AMD Athlon Processor 2200+, Memory 1GB DDR Ram, Network 100 Mbps interconnection network, Disk 80GB IDE H.D., *Software Environment* O.S. ReadHat Linux 7.3, Library MPICH2 1.0.3. Only compared with Chiang's algorithm [8], did not mention others. WDPA is nearly 120 times faster than Ye's algorithm in the 16 processors case. Experimental results show that WDPA achieved higher speedups than previous works in the case of high data volume and low support.

Algorithm proposed by Shah2009 et.al.[10] determines the no. of running processes and divides the load equally to maximize the system performance and its efficiency. The proposed algorithm uses a simple principle of allowing "redundant computations in parallel on otherwise idle processors to avoid communication". The parallel versions were implemented using the MPICH2 library functions under the Windows environment. These experiments are conducted on a 3.0 GHz, duo core processor systems with 2GB of RAM, dual-core processor running n processes. The experiments conducted show that the parallel algorithm scales. They analyzed the performance of the algorithm over n processors on a commodity cluster of machines. The extensive experiments carried out show that the parallel algorithm scales well to the number of processors and improves on the efficiency by effective load balancing.

In the paper by Ning Li et. al. [11] the authors have implemented a parallel Apriori (PApriori) algorithm based on MapReduce, which is a framework for processing huge datasets on certain kinds of distributable problems using many computers (nodes). Results demonstrated that the proposed algorithm could scale well and efficiently process large datasets on commodity hardware. PApriori algorithm needs one kind of MapReduce job. The map function performs the procedure of counting each occurrence of potential candidates of size k and thus the map stage realizes the occurrences counting for all the potential candidates in a parallel way. Then, the reduced function performs the procedure of summing the occurrences counts. For each round of the iteration, such a job is carried out to implement the occurrences computing for potential candidates of size k. It was run on a cluster of 10 computers, six with four 2.8GHz cores and 4GB memory, the rest four with two 2.8GHz cores and 4GB memory. Hadoop version 0.20.2 and Java 1.5.0_14 are used as the MapReduce system for all the experiments. Experiments were carried out 10 times to obtain stable values for each data point. For size up - the program is more efficient as the database size is increased. For Speedup - PApriori algorithm can deal with large datasets efficiently than smaller ones. For scaleup - PApriori algorithm scales well, the scaleup falls short as the database and multiprocessor sizes increase. It always maintains a higher than 78% scalability for dataset1 and 80% for T10I4D100K.

As presented in Verma et. al and Yahya et.al's work [12-13], Yahya et.al's [13] experimentation was done with Windows 7 (64-bit) with Hadoop version 0.20.0 and JDK version 1.6.31 having sample dataset T1014D100k produced by IBM quest synthetic data generator. Multiple experiments with variation in minimum support count as well as length of frequent itemset were performed during execution. Results shown that one-phase algorithm was incompetent, however, two-phase and K-phase algorithms both were having comparable results. As the algorithm is producing equivalent results comparable to K-phase algorithms still it is generating a large number of partial frequent itemsets which lead to additional computational time in each node moreover synchronization and communication overhead to mapper and reducers during phase-2, which makes phase-2 more intricate.

In Yang et.al's work [14], as described in Verma et. al and Yang et.al, [13-14], the execution the author of [14]'s experimentation was achieved over Hadoop version 0.20.0 with nine systems having 2.60 GH configurations. Sample datasets were generated from the IBM data generator. The algorithm deploys speed-up as criteria to test the performance. However, the algorithm works speedily with inordinate datasets as compared to small datasets, and execution time is comparably less. The conclusion drawn from this study was, parallel algorithms with speed-up parameter working quite efficiently as compared to other algorithms furthermore this algorithm is computationally faster as it exchanges count rather than data.

2.3 Parallel Apriori Implementation on GPU

The authors of [15] have studied GPGPU platforms for Apriori-based data mining algorithms. Based on Apriori, they have proposed an algorithm named "HSApriori" that uses the parallel processing nature of modern GPU. They have studied both *tidset* and bitset representations of dataset and identified the bitset representation is suitable for parallel processing. They have described the mechanism for support counting in the GPU computing environment where multiple threads are used. The algorithm is tested using a prototype application. They used datasets obtained from the UCI machine learning repository along with a synthetic dataset that was taken from IBM Almaden Quest Research Group. Their results revealed that the speed-up ratio is substantially more with HSApriori than the approach presented in [16]

The authors of [17] have experimented by creating a huge size transactional database with the help of random number generation. The algorithm proposed in [17] is based on two phases. Firstly, they have implemented the Apriori ARM on the map-reduce model on Hadoop. And tested it with different sets of itemsets. In the second phase, parallel Apriori (which is GPU-based) was implemented on the map-reduce model. After the two phases, the authors compared the difference between the GPU based and non-GPU based algorithms. Their method could achieve up to a maximum of 18x speedup as they use GPU's in the cluster. They highlight that attaching a GPU to every node in a Hadoop cluster adds extra hardware charge and should be omitted for better performance.

The purpose of [18] is to implement an association rule mining algorithm using the Nvidia CUDA framework for general-purpose computing on GPU. They have performed their experiment on Nvidia 9500 GT GPU with CPU Intel Q9550 2.83Ghz and Nvidia GTX 285 GPU with CPU Intel E7400 2.8Ghz. They have used bitmap-based structure as it efficiently computes the frequent itemsets by using bitmap AND (&) operation which is very fast. Also, the storage required to represent Item is one bit per Item. If an itemset contains Item#1 and Item#3 than the respective data is represented as a two bytes integer. To deal with a very large number of itemsets and to enhance the execution of a single thread, each thread has the responsibility to compute the support of multiple itemsets instead of single itemset. This is implemented by first calculating the start and end index on which a single thread has the responsibility to compute support. Then a for loop is implemented to computer confidence of selected itemsets. In the next step it uses __syncthreads() to synchronize all threads within a single block to compute respective support count. Finally, it computes the aggregated support of a single rule for which a single block is responsible. The authors have showed their program can be 4X to 20X faster than solving the same problems on CPU based implementations.

In [19], Silvestri et. al. presented a many-core parallel version of a state-of-the-art FIM algorithm, DCI, whose sequential version resulted, in most of the tested datasets, better than FP-Growth, one of the most efficient algorithms for FIM. They proposed a couple of parallelization strategies for Graphics Processing Units (GPU) suitable for different resource availability, and they presented the results of several experiments conducted on real-world and synthetic datasets. In this work they presented the GP-GPU version of a state-of-the-art FIM algorithm, DCI, whose sequential version resulted in orders of magnitude better than the well-known Apriori algorithm. Also, for most of the tested datasets, DCI resulted better than FP-Growth, a famous divide & conquer FIM algorithm. It used simple static data structures and permitted a lot of data parallelism (involving bitwise operations) to be exploited and because of that might be a good candidate for a GPU porting. In this paper the authors focused on a pair of parallelization strategies. The former uses a simple map-reduce paradigm to realize in parallel collective logical operations between bitmaps with a final *bit count*: we call this technique transaction-wise, since each bitmap records the presence/absence of a given item in all the transactions of the dataset. The latter adopts a nested data-parallelism, where blocks of (candidate) itemsets are assigned to distinct GPU's multiprocessors for computing their supports (number of occurrence of each candidate itemset in the dataset transactions), where each GPU's multiprocessor in turn adopts a map-reduce paradigm like in the previous approach. We call this strategy candidate-wise, and its exploitation affects a crucial feature of DCI: the management of the special cache used to store intermediate results and save work. The experiments were

executed on T40I10D100K benchmark dataset amongst others, and a server equipped with an Intel Core2 Quad CPU @ 2.66GHz, 8 GB of RAM, and an NVIDIA GTX275 GPU featuring: 30 multiprocessors (240 cores) @ 1.4 GHz, 896MB device memory, and CUDA device capability 1.3. Finally, in this paper they introduced a parallel algorithm, GPU DVI, which exploits GPUs to compute frequent itemsets, that is to find the subsets of items that are contained in at least a given fraction of a collection of transactions (i.e., sets of items).

In [20], Yun Li et. al. proposed an algorithm for mining closed frequent itemsets based on a new vertical data structure. Especially when dealing with large datasets, the proposed algorithm can obtain high-speed computing with the help of a graphics processing unit. The experimental results show that their proposed algorithm requires much less computation time than other related methods. This paper proposed a closed frequent itemset mining algorithm, called MVCG. The MVCG algorithm uses a multi-layer vertical data structure. In the proposed algorithm, unfixed-length bit vectors are used to represent the items. When dealing with large datasets, their algorithm could obtain a high speed with the help of GPU, especially on large-size sparse datasets. The experimental results showed that their proposed algorithm requires much less computation time than other related methods. All experiments are carried out on a small cluster composed of 14 IBM blade servers. Computational nodes are connected by gigabit switches and MPI standard library is used for communications between nodes. NVIDIA's GPU GTX-970 with 4G of DDR5 graphical memory is used. Open CL library fits the specification of Open CL 1.2. The C++ is employed for programming. Experiments were run on the simulated dataset by IBM QDG – 100K and 1000k transactions.

3 Apriori Algorithm

Apriori is an algorithm for frequent itemset mining and association rule learning over relational databases [22]. It proceeds by identifying the frequent individual items in the database and extending them to larger and larger item sets if those item sets appear sufficiently often in the database.

Let, X, Y ⊆ I be any two itemsets. If X ⊆ Y, then sup(X) ≥ sup(Y), which leads to the following two observations: i) if X is frequent, then any subset Y⊆ X is also frequent, and ii) if X is not frequent, then any superset Y ⊇ X cannot be frequent.

A brute force approach to pattern mining would enumerate all possible itemsets in its quest to determine the frequent ones. This results in a lot of wasteful computation since many of the candidates may not be frequent. The Apriori algorithm utilizes the two properties of frequent itemsets to significantly improve the brute-force approach. It employs a level-wise or breadth-first exploration of the itemset search space, and prunes all supersets of any infrequent candidate, since no superset of an infrequent itemset can be frequent. It also avoids generating any candidate that has an infrequent subset. In addition to improving the candidate generation step via itemset pruning, the Apriori method, also significantly improves the I/O complexity. Instead of counting the support for a single itemset, it explores the prefix tree in a breadth-first manner and computes the support of all the candidates of size k that comprise level k in the prefix tree shown in Fig 1.

```
   APRIORI (D, 𝓘, minsup):
 1 ℱ ← ∅
 2 C(1) ← {∅} // Initial prefix tree with single items
 3 foreach i ∈ 𝓘 do Add i as child of ∅ in C(1) with sup(i) ← 0
 4 k ← 1 // k denotes the level
 5 while C(k) ≠ ∅ do
 6     COMPUTESUPPORT (C(k), D)
 7     foreach leaf X ∈ C(k) do
 8         if sup(X) ≥ minsup then ℱ ← ℱ ∪ {(X, sup(X))}
 9         else remove X from C(k)
10     C(k+1) ← EXTENDPREFIXTREE (C(k))
11     k ← k + 1
12 return ℱ(k)

   COMPUTESUPPORT (C(k), D):
13 foreach ⟨t, i(t)⟩ ∈ D do
14     foreach k-subset X ⊆ i(t) do
15         if X ∈ C(k) then sup(X) ← sup(X) + 1

   EXTENDPREFIXTREE (C(k)):
16 foreach leaf Xa ∈ C(k) do
17     foreach leaf Xb ∈ SIBLING(Xa), such that b > a do
18         Xab ← Xa ∪ Xb
           // prune candidate if there are any infrequent subsets
19         if Xj ∈ C(k), for all Xj ⊂ Xab, such that |Xj| = |Xab| − 1 then
20             Add Xab as child of Xa with sup(Xab) ← 0
21     if no extensions from Xa then remove Xa from C(k)
22 return C(k)
```

Fig 2. Apriori Algorithm Pseudocode [22]

Fig 2. shows the pseudo-code for the Apriori method. Let $C^{(k)}$ denote the prefix tree comprising all the candidate k-itemsets. The method begins by inserting the single items into an initially empty prefix tree to populate $C^{(1)}$. The while loop (Lines 5-11) first computes the support for the current set of candidates at level k via the *ComputeSupport* procedure that generates k-subsets of each transaction in the database D, and for each such subset, it increments the support of the corresponding candidate in $C^{(k)}$ if it exists. This way, the database is scanned only once per level, and the supports for all candidate k-itemsets are incremented during that scan.

Next, we remove any infrequent candidate (Line 9). The leaves of the prefix tree that survive comprise the set of frequent k-itemsets $F^{(k)}$ which are used to generate the candidate (k + 1)-itemsets for the next level (Line 10). The *ExtendPrefixTree* procedure employs a prefix-based extension for candidate generation. Given two frequent k-itemsets X_a and X_b with a common (k-1) length prefix, i.e., given two sibling leaf nodes with a common parent, we generate the (k +1)-length candidate $X_{ab} = X_a \cup X_b$. This candidate is retained only if it has no infrequent subset. Finally, if a k-itemset X_a has no extension, it is pruned from the prefix tree, so that in $C^{(k)}$ all leaves are at level k. If new candidates were added, the whole process is repeated for the next level. This process continues until no new candidates are added.

The worst-case computational complexity of the Apriori algorithm is $O(|I| \cdot |D| \cdot 2|I|)$, since all itemsets may be frequent. In practice, due to the pruning of the search space the cost is much lower. However, in terms of I/O cost Apriori requires O(|I|) database scans, as opposed to the O(2|I|) scans in brute-force. In practice, it requires only a *len* number of database scans, where *len* is the length of the longest frequent itemset.

4 Methodology

4.1 Bitset Representation

To implement the subset operation efficiently by bitwise operations we represented the itemsets (both frequent patterns and transactions) by their bitset representations. In this representation, each transaction is allocated a fixed chunk of bytes. Each bit in the chunk represents either the presence or absence of an item. We took 1=presence, and 0=absence. The order of the bits is assigned according to the "little-endian" convention – where the first byte represents items 0 to 8, the second one from 8 to 16, and so on.

To convert the datasets into binary representations, we used Python with NumPy – where we used a NumPy array of bits which are packed together by the np.packbits() function and later written as binary files.

4.2 Data Representation In memory

The patterns, i.e. itemsets can be easily stored as a C++ vector of char (characters). However, to seamlessly switch to raw memory representation (for CUDA kernel calls) from C++ vectors without any memory rearrangement, we needed to make sure that the patterns remain contiguous in the memory. To ensure this we used two classes – Dataset keeps the memory representation contiguous, while also providing access to individual patterns via the second class, Itemset which provides an interface for individual patterns without requiring a copy of memory.

4.3 Apriori Algorithm Implementation

We first made sure to implement user interfaces such as bitwise operations, identification of siblings, etc. to the itemset class. For example, the set union operation is simply equivalent to the bitwise or operation.

The first two steps of candidate generation are straight-forward, and we used specific and efficient procedures to perform them, whereas the generalized step for the N'th step requires sibling identification and more involved steps. So, we kept a separate procedure to perform that.

4.4 Computation of Support

The subset operation of the bitsets can be done with a very simple biwise operation –

$$(X \subseteq Y) \equiv !\,(X \& \sim Y)$$

We use a CUDA kernel to compute support of each pattern in parallel on a chunk of data transactions. Each thread computes support for a particular pattern.

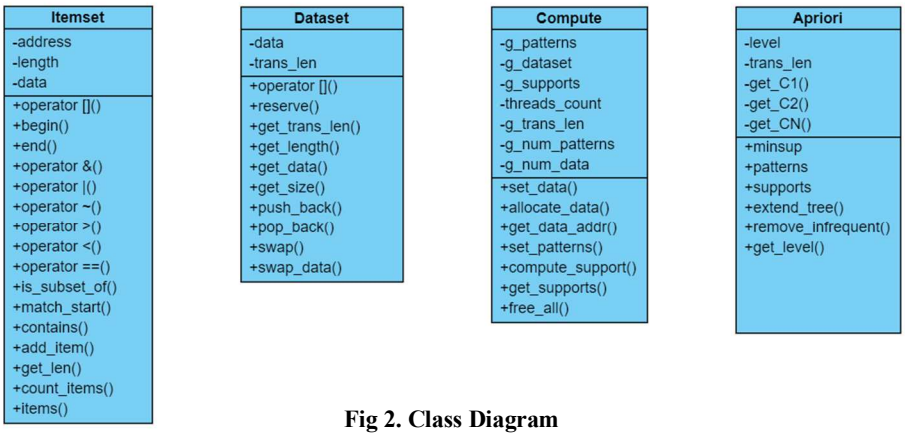

Fig 2. Class Diagram

4.5 MPI Communication (Allreduce)

After each MPI Rank computes support locally on its chunks of input data, it must communicate with other ranks to get the total support for the whole dataset. We performed an MPI_Allreduce with summation operation on the supports in place – to sum up, the supports for each rank.

4.6 MPI I/O

Larger datasets are either impossible or very expensive to load into memory at once (as a whole). Instead, we used MPI_Read_at to read sequential blocks of data into memory for each MPI ranks and computed support onto them. As will be shown later, this reading pattern is very efficient on AiMOS and does not significantly reduce the bandwidth of the support computation process. To make this process more efficient we read the patterns directly into CUDA "managed" memory – to avoid unnecessary repetition.

The program outputs two files, one containing mined frequent patterns – written also in little-endian bitset representation and a supports file of unsigned integer values that contain the supports in the same order as the patterns. To write outputs in parallel each MPI rank takes a sequential chunk of pattern/support and uses MPI_Write_at to write it into proper output position.

4.7 Program Workflow

The program starts by initializing MPI, determining the rank of the process, and setting the CUDA device (i.e. GPU) according to rank. Next, we allocate enough data into GPU Managed memory to read chunks of an input data files. Next, we move onto initializing the Apriori algorithm by extending the prefix tree, i.e. generating the first batch of candidates. Then we allocate memory for support calculation and set the supports to zero. We repetitively read chunks of inputs by MPI Read, compute supports on them and increase the computed support accordingly.

When we reach the end of the file, we communicate the calculated supports in each process by MPI_Allreduce. Then we prune the infrequent patterns and write the frequent patterns and supports in chunks to the corresponding output files by MPI Write. If we have two or more patterns, we again generate new candidates and continue the algorithm. Otherwise we clean up the memories, close the opened files, finalize MPI and end the program.

5 Experimental Results and Discussions

5.1 Dataset Details

We wanted to observe the performance of our implementation on datasets of different lengths. Additionally, we also wanted to observe the effect of varying transaction lengths on the performance of our implementation. For this, we used 3 different base datasets, i.e. MNIST_25, MNIST_98, and T10I4D100K.
MNIST_98 is the binarized and linearized version of MNIST training database containing 60,000 handwritten digits where each set bit represents a white pixel which can

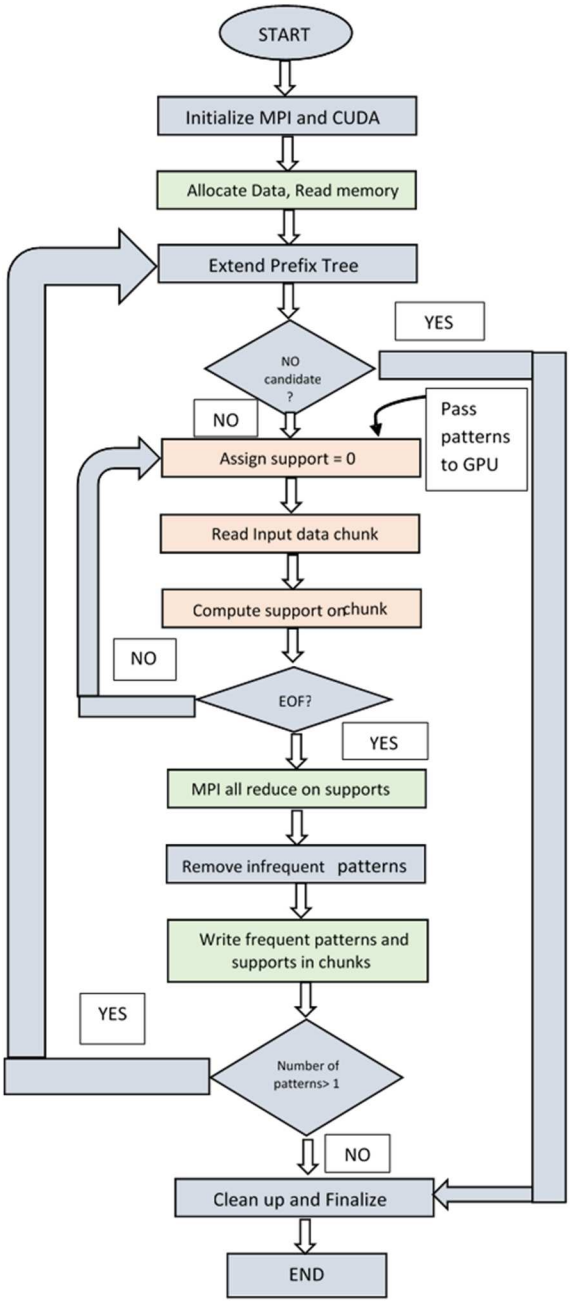

Fig 4. Program Workflow Diagram

be interpreted as "occurrence" of a particular pixel and each 28x28 = 768-pixel image is interpreted as a transaction. We need 98 bytes of data to represent each transaction in the bitset format. MNIST_25 is the reduced resolution version of the 14x14 = 196-pixel images requiring 25 bytes of bitset representation.

T40I4D100K is a renowned synthetic database generated using the IBM Quest generator. This database is widely used to evaluate various frequent and correlated pattern mining algorithms. It contains 100 thousand transactions of one thousand items. For our purposes we converted it to bitset representation with 125 bytes per transaction.

To experiment on longer a dataset, we generated a longer version of the MNIST_25 dataset by repeating each transaction 200 times to produce the MNIST_rep_200_25 dataset which contains 12 Million transactions of 196 items.

5.2 Setup Configuration

Our experimental setup consists of several components/modules. For implementation, we have used C++ for the host portion and CUDA C++ for the device code. We have used mpic++ or g++ and nvcc for collective C++ and CUDA code compilation, respectively. Our code was run on the IBM DCS Supercomputer – AiMOS. Also, we have used 3 modules i.e. gcc/7.4.0/1, Spectrum-MPI, and CUDA. We have used Slurm as a resource manager which is compatible with IBM Spectrum MPI which we used.

AiMOS contains System contains 268 nodes, each with 2x IBM Power 9 processors clocked at 3.15 GHz. Each processor contains 20 cores with 4 hardware threads (160 logical processors per node), 512 GiB RAM, and 1.6 TB Samsung NVMe Flash. Within the total cluster, 16 nodes contain 4x NVIDIA Tesla V100 GPUs with 16 GiB of memory each and 252 nodes each contain 6x NVIDIA Tesla V100 GPUs with 32 GiB of memory each. GPU uses a SM_70 architecture. We have used the high-resolution (512 MHz) built-in cycle counter available in POWER 9 architecture to measure the time.

5.3 Experimental Analysis – scalability of different components

Our program has four components from the standpoint of parallelizability – (i) the support computation portion of the code runs independently completely in parallel manner in different GPUs (when available). Thus, we expected this part to scale linearly with the number of available GPUs. (ii) The candidate generation component is serial and stays fixed for with increased number of GPUs/MPI Ranks. (iii) The MPI Communication i.e. Allreduce is O(log n) where n is the number of MPI ranks, but we expect it to remain almost constant or increase slightly with the number of MPI ranks. However, if the number of candidates is higher, it needs to communicate a larger chunk of memory (containing support values) and thus incurs more overhead. (iv) Due to the sequential read/write of blocks of memory via MPI I/O, the parallel I/O component of our program is quite efficient. However, we expect a slight increase in I/O overhead with an increased number of MPI ranks.

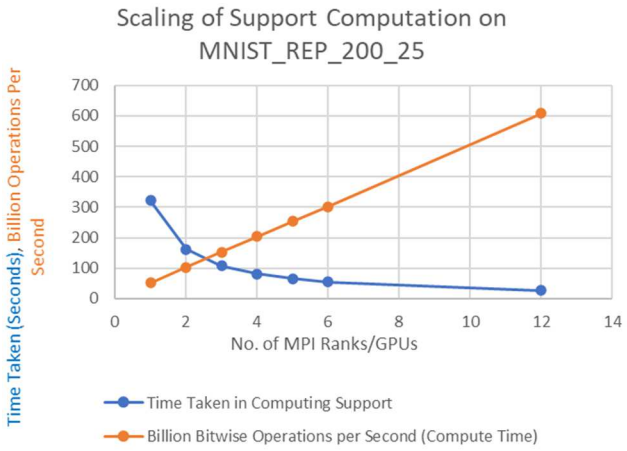

Fig 5. Scaling of Support Computation time on MNIST_REP_200_25

To compare the scalability of these components we consider two different scenarios – on a larger dataset for a moderate minimum support value (i.e. moderately high number of candidates) the support computation component dominates the total run time. So, we experimented on the MNIST_REP_200_25 dataset which contains 12 million transactions. We wanted to observe how the total number of available MPI Ranks/GPUs affect the time taken for different components. The result of the experiment is presented in Fig 5-7.

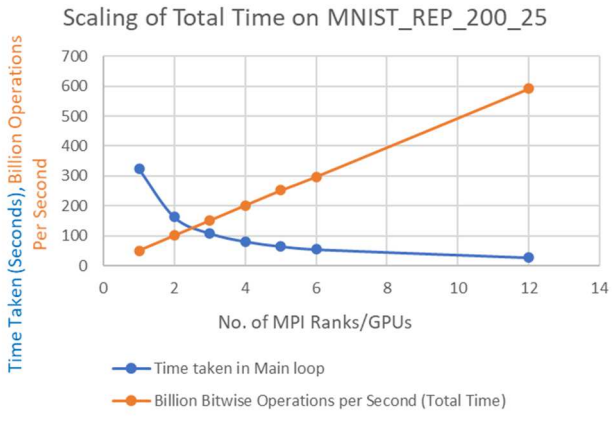

Fig 6. Scaling of Total Time on MNIST_REP_200_25

As we can see from Fig 5. the total time taken for compute support reduces inversely with an increasing number of MPI Ranks/GPU while the number of bitwise operations linearly increases. This shows that this component of the program has a very strong scaling. From Table 1. we see that the support computation part dominates the total time taken. So, we also observe a similar pattern in Fig 6. while which shows the total time taken for overall the overall Apriori algorithm and corresponding bitwise operations/second on this dataset.

Scaling Behavior for A Larger Dataset (MNIST_rep_200_25) for moderate Value of Support (Moderate No. of Mined Patterns = 55024)				
No of MPI Ranks/GPUs	Time taken in Main loop	Time Taken in Computing Support	MPI Overhead Time (in seconds)	Apriori Candidate generation overhead
1	323.892	322.201	0.139243	1.551757
2	162.893	161.343	0.28716	1.26284
3	108.927	107.671	0.545615	0.710385
4	81.7063	80.9816	0.044489	0.680211
5	65.5645	64.9444	0.0463291	0.5737709
6	55.4445	54.5585	0.040945	0.845055
12	27.8858	27.1353	0.185097	0.565403

Table 1. Scaling Behavior for MNIST_REP_200_25 for moderate value of support

In Fig 7. Shows how the MPI Communication overhead varies with an increasing number of MPI Ranks. As we can see from the figure, MPI communication overhead time constitutes only a small portion of the total run time and remained almost constant with very negligible fluctuations. As can be seen from Table I – the serial component of the program, i.e. Apriori candidate generation overhead constitutes only a small portion of the total run time.

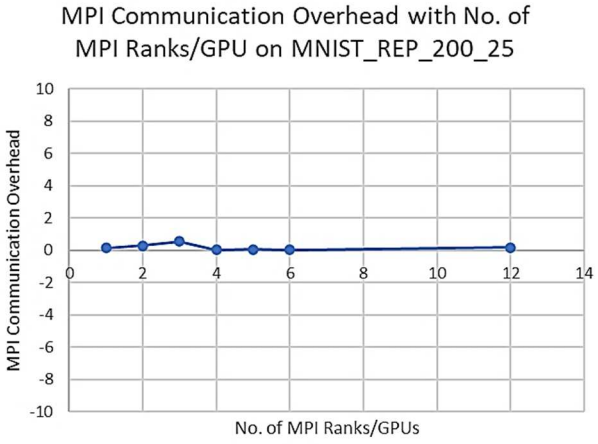

Fig 7. MPI Communication Overhead with No. of MPI Ranks on MNIST_REP_200_25

Now, we move on to a different scenario – where we have a smaller dataset with very low minimum support – which leads to a large number of frequent pattern candidates. This makes the tree extension part of the Apriori algorithm and other overheads, such as CPU to GPU transfers, more expensive – which in our case runs serially. This is apparent from our experiment on the MNIST_25 dataset. The results are presented in Fig. 8-10 and a Table 2. In Table 2, we see that a large overhead occurs for the Apriori Candidate generation.

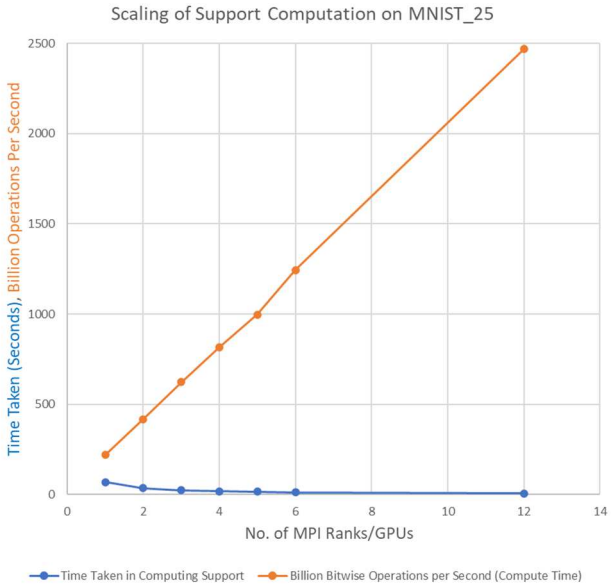

Fig 8. Scaling of Support Computation time on MNIST_25

Although we can see that the support computation part of the program still scales linearly with the number of GPUs from Fig 8. However, the total time no longer scales linearly as seen in Fig 9. The total time taken on the overall Apriori algorithm shows a flattened trend for a higher number of MPI ranks due to Amdahl's law.

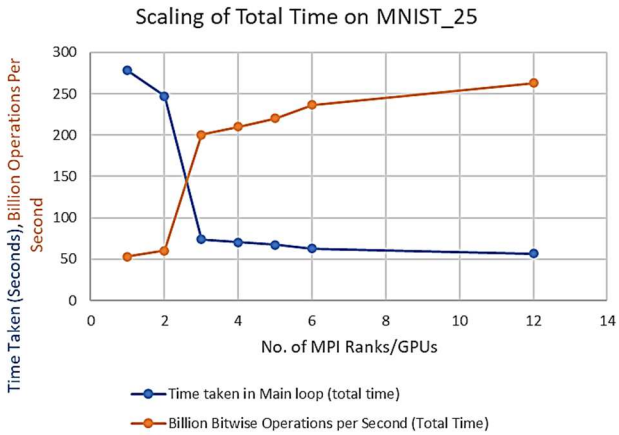

Fig 9. Scaling of Total Time for Apriori Implementation on MNIST_25

In Fig 10., the MPI Communication overhead for MNIST_25 dataset against increasing No. of MPI was presented. It is noteworthy here that overhead had a significant increase when the number of MPI ranks were increased from 6 to 12. This is because at this point, we move on to two different nodes form one (we have 6 ranks per node). Communication of a large number of support values across nodes causes bigger overhead.

Scaling Behavior for A Smaller Dataset (mnist_25) for low Value of Support (Higher No. of Mined Patterns = 9880591)				
No of MPI Ranks/GPUs	Time taken in Main loop (total time)	Time Taken in Computing Support	MPI Overhead Time in seconds	Apriori Candidate generation overhead
1	278.051	67.8514	0.702413	209.497187
2	247.165	35.5442	6.60874	205.01206
3	73.9669	23.7727	0.116305	50.077895
4	70.5993	18.1405	2.0046	50.4542
5	67.2598	14.86644	0.35773	52.03563
6	62.5849	11.9213	0.36173	50.30187
12	56.3967	6.0041	10.5976	39.795

Table 2. Scaling Behavior for MNIST_25 for moderate value of support

Finally, we wanted to verify if the scaling trend is consistent across datasets of different transactions byte lengths. The results are presented in Tables 3-5. We see that all datasets show similar trends for different portions of the programs.

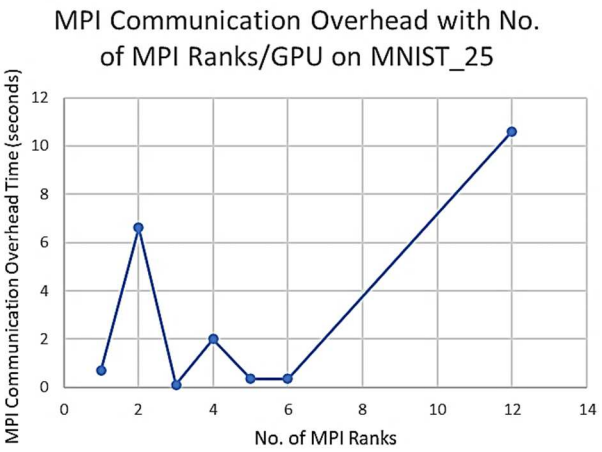

Fig 10. MPI Communication Overhead with No. of MPI Ranks on MNIST_25

Table showing Compute scalability [Compute Support Time in second] for 3 different datasets in 3 different MPI Rank Configurations			
Number of MPI Ranks	Dataset 1 - T40I4D100K	Dataset 2 - MNIST 98	Dataset 3 - MNIST 25
1 node 1 MPI Rank	558.647	186.717	9.5777
1 Node, 6 MPI Rank	93.9234	29.7684	1.70103
1 Node, 12 MPI Ranks	46.6788	15.5973	0.862226

Table 3. Compute Scalability for 3 different datasets in 3 different MPI Rank Configurations

Table showing MPI Overhead Time (in second) for 3 different datasets in 2 MPI Rank Configurations			
Number of MPI Ranks	Dataset 1 - T40I4D100K	Dataset 2 - MNIST 98	Dataset 3 - MNIST 25
1 Node, 6 MPI Rank	0.814253	0.834752	0.291591
1 Node, 12 MPI Ranks	2.48809	3.1133	12.6482

Table 4. MPI Overhead for 3 different datasets in 2 MPI Rank Configurations

Table showing Candidate Generation Overhead (in second) for 3 different datasets in 3 different MPI Rank Configurations			
Number of MPI Ranks	Dataset 1 - T40I4D100K	Dataset 2 - MNIST 98	Dataset 3 - MNIST 25
1 node 1 MPI Rank	6.062	3.118	6.2182
1 Node, 6 MPI Rank	2.900247	2.378548	6.183439
1 Node, 12 MPI Ranks	2.74281	2.4246	6.259574

Table 5. Candidate Generation Overhead for 3 different datasets for 3 different MPI Rank configurations

5.4 Experimental Analysis – GPU Thread Configuration and Computational Performance

To find the best thread configuration for the CUDA kernel launch we experimented on the three base datasets with different GPU threads. We see that from Fig 11. the time for computing support does not change drastically for different thread configurations. However, we did notice an improvement for a higher number of threads. We achieved the best performance for 1024 threads which is the highest number of threads on the V100 GPUs.

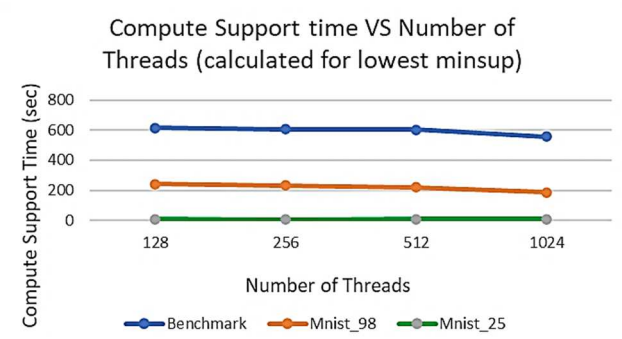

Fig 11. Compute Support time against Number of Threads for 3 base datasets

Next, we wanted to experiment on the utilization of the computational capacity of the GPU for different problems. We assigned each thread to a different pattern for support computation. So, when the number of patterns is larger, the performance reaches its peak.

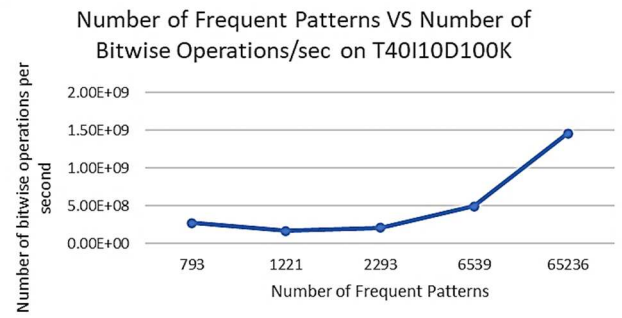

Fig 12. Number of Bitwise operations per second against the number of frequent patterns on T40I10D100K

For a robust observation we ran our experiment on two different datasets of different transaction byte lengths. In Fig 12. And Fig 13. number of bitwise operations per second was plotted against the number of frequent patterns on T40I10D100K and MNIST_25 dataset, respectively.

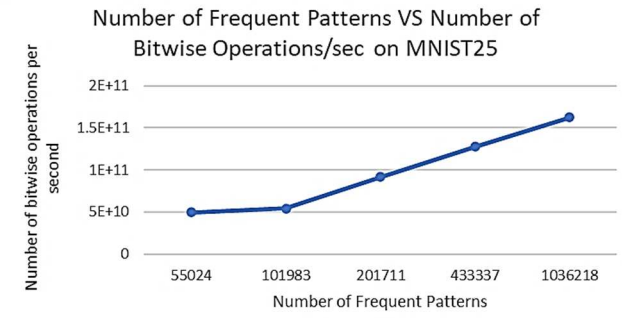

Fig 13. Number of Bitwise operations per second against the number of frequent patterns on MNIST_25

Both have a similar trend of going upwards which verifies our assumption.

5.5 Experimental Analysis - Effect of Dataset Length on Computational Efficiency

The biggest portion of computation occurs for the calculation of support. We parallelized support computation over the patterns. We did not perform any parallelization for data points. So, we expect support computation time per data point to remain almost constant. The performance only increases if we have a higher number of GPUs.

Compute Support Time (in Microseconds) per Data length		
Configurations	Smaller Dataset - MNIST 25	Larger Dataset - MNIST_rep_200_25
1 Node 1 MPI Ranks	27.8015	26.85008333
1 Node 2 MPI Ranks	14.65416667	13.44525
1 Node 3 MPI Ranks	10.21098333	8.972583333
1 Node 4 MPI Ranks	12.96666667	6.748466667
1 Node 5 MPI Ranks	6.30845	5.412033333
1 Node 6 MPI Ranks	28.3505	4.546541667
1 Node 12 MPI Ranks	14.37043333	2.261275

Table 6. Compute Support time Per Data length

As in most cases total runtime for a larger dataset is dominated by support computation which stays constant per data point and the total runtime per data point for a larger dataset is lower than that of a smaller dataset. These observations are established by experimental results presented in Table 6 and Table 7.

Total Apriori Execution Time (in Microseconds) per Data length		
Configurations	Smaller Dataset - MNIST 25	Larger Dataset - MNIST_rep_200_25
1 Node 1 MPI Ranks	37.72516667	26.991
1 Node 2 MPI Ranks	22.087	13.57441667
1 Node 3 MPI Ranks	17.5685	9.07725
1 Node 4 MPI Ranks	15.5742	6.808858333
1 Node 5 MPI Ranks	14.92835	5.463708333
1 Node 6 MPI Ranks	136.2676667	4.620375
1 Node 12 MPI Ranks	329.5	2.323816667

Table 7. Total Apriori Execution Time per Data Length

5.6 Experimental Analysis – Parallel I/O Efficiency

We measured the read and write efficiency using MPI I/O for different Node and MPI Task configuration and different sizes of the read and write blocks. We see an

increase in read performance with increasing block size. The speed also increases with an increasing number of MPI tasks within a single node. However, when we moved on to two nodes, either we observe a flattening of the read speed or decrease in case of the largest block size experimented on i.e. 1 Megabyte. We expect that block size of 512 Kilobytes, or 1 Megabyte would be suitable for most cases.

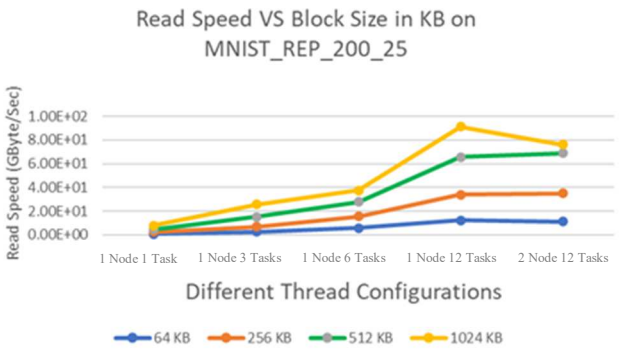

Fig 14. Read Speed VS Block Size on MNIST_REP_200_25

We observed that if the number of nodes and size of the read/write block is increased then the efficiency reduces. The results for read performance are presented in Fig 14. and Table 8.

Table showing Read Speed (in GBytes/sec) for varying block size and in different configurations					
Block Size	1 Node 1 Task	1 Node 3 Tasks	1 Node 6 Tasks	1 Node 12 Tasks	2 Nodes 12 Tasks
64 KB	0.75204	2.67076	5.76201	12.4050	11.3573
256 KB	1.81511	4.1550	9.96984	21.7002	23.7149
512 KB	2.12113	8.26520	12.0719	31.6014	33.8002
1024 KB	3.47389	10.4486	9.6747	25.4092	7.210845

Table 8. Read Speed for varying block size and in different configurations

The results for write performance are presented in Fig 15. and Table 9. We see that the larger block size of 1024 KB has an edge over smaller ones for a lower number of MPI ranks. However, for a higher number of MPI ranks, we did not see any significant difference in performance for different block sizes.

5.7 Experimental Analysis – Performance Improvement with CUDA and MPI

To verify the improvement over baseline performance on a single CPU running the Apriori algorithm in serial, we stripped our implementation of both MPI and CUDA. Then we added MPI to the baseline code and evaluated performance improvement for multiple MPI ranks running on the same node and also on different nodes.

Table showing Write Speed (in GBytes/sec) for varying block size and in different configurations					
Block Size	1 Node 1 Task	1 Node 3 Tasks	1 Node 6 Tasks	1 Node 12 Tasks	2 Nodes 12 Tasks
64 KB	9.342	17.493	52.066	8.384	8.602
256 KB	15.863	19.997	25.155	6.695	14.465
512 KB	17.223	4.398	21.465	14.496	12.966
1024 KB	120.693	221.807	23.774	12.230	47.397

Table 9. Write Speed for varying block size and in different configurations

Again, we added CUDA functionality to our code to perform support computation on the GPU and evaluated the performance improvement. Then, we moved on to our final implementation with both CUDA and MPI which utilizes multiple MPI ranks running on multiple GPUs.

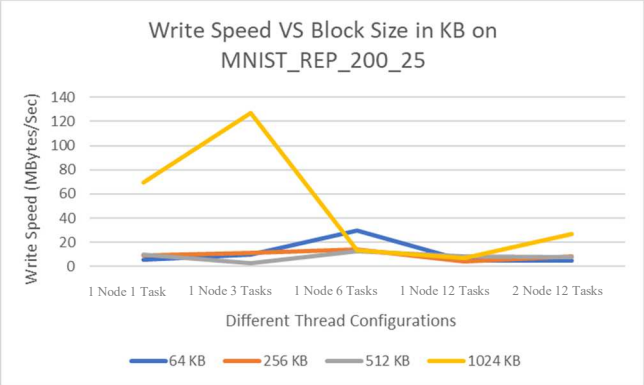

Fig 15. Write Speed VS Block Size on MNIST_REP_200_25

We can see from the results presented in Table 10. that addition of MPI improves over the baseline performance of the serial implementation linearly with an increasing number of MPI ranks. We see consistent improvement with an increased number of MPI ranks regardless of whether they reside on the same node or not.

Table showing performance without CUDA on MNIST 25 dataset with minimum support 4000		
Configurations	Overall Apriori Execution Time (sec)	Compute Support Time (sec)
No MPI No CUDA	79.6804	79.6252
1 Node, 6 Tasks per node	14.1924	13.9441
1 Node, 12 Tasks per node	7.23752	7.03066
2 Nodes, 6 Tasks per node	7.21152	6.96482
2 Nodes, 12 Tasks per node	3.70512	3.51786

Table 10. Performance without CUDA on MNIST_25 dataset with minimum support 4000

From Table 11. we can see that performing support computation on the GPU by CUDA implementation drastically reduces the runtime. Next, when we add MPI support to the GPU implementation to utilize multiple GPUs we see that the performance improves almost linearly with the number of GPUs. However, in this case, running multiple MPI tasks sharing the same GPU does not necessarily lead to an improvement of performance as can be seen for the case of 2 Nodes and 12 Tasks per node.

Table showing performance with CUDA on MNIST 25 dataset with minimum support 4000		
Configurations	Overall Apriori Execution Time (sec)	Compute Support Time (sec)
No MPI with CUDA	0.801511	0.772986
1 Node, 6 Tasks per node	0.222461	0.149666
1 Node, 12 Tasks per node	0.197863	0.131464
2 Nodes, 6 Tasks per node	0.115643	0.074859
2 Nodes, 12 Tasks per node	0.1686	0.056779

Table 11. Performance with CUDA on MNIST_25 dataset with minimum support 4000

6 Summary of Results

From our experimental results we see that support computation component of our implementation scales linearly with available GPUs. The computation time per data length stays almost constant for each datapoint. As in most practical purposes, the runtime of the Apriori algorithm is dominated by the support computation step, we conclude that our implementation should be scalable to large datasets containing numerous transactions. As the candidate generation component of the algorithm is still done serially the total runtime is only limited by this part in a massively parallel system. We did not observe any significant overhead MPI communication which only serves to communicate the computed support values among the ranks. Also, our algorithm is not limited by I/O performance as overhead for I/O is much lower compared to the actual algorithm runtime.

7 Future Work

The performance of our algorithm is limited by the serial portion of our implementation i.e. Apriori Candidate Generation process. However, we think that this portion can also be parallelized, but that would require a more involved approach. As for support computation, we only parallelized the computation process over multiple candidate patterns. However, this process can also be parallelized for data points and later a reduction operation can be performed on the GPU to compute the support. We leave this as a future work. We are using MPI_Allreduce() to communicate calculated support values among all MPI ranks. However, for the Apriori algorithm to proceed, we don't need the exact support values rather whether the support value is greater or equal to the minimum support value. We expect that the MPI communication overhead can be reduced if only the result of this comparison can be communicated rather than the actual values. However, as the MPI communication overhead was already low compared to the other components, we did not investigate this further. We may explore this further in our future research works.

8 Conclusion

In this paper we have investigated an implementation of the Apriori algorithm tailored more specifically for the DCS cluster of AiMOS, utilizing MPI, CUDA, and parallel I/O. We focused on the most computationally expensive part of this algorithm – the support computation which was performed by highly efficient bitwise operations on the GPU. Then we devised the implementation of the algorithm for parallel support computation on different MPI ranks and necessary reduction among ranks. To read and write large datasets in parallel efficiently we used MPI I/O which performs I/O in sequential blocks. By experimental results we show that the support computation process scales linearly with available resources. Also, the runtime of the algorithm scales sub-linearly with the size of the dataset. Along with efficient I/O, these two behaviors allow this implementation to be applicable for big datasets containing millions of transactions in the advent of a massively parallel system e.g. AiMOS.

Starting with our motivation behind choosing this algorithm for the project, we move on to present a review of the relevant research works. Then we describe how the Apriori algorithm works originally and sequentially. Later, we present our implementation techniques, parallelization of the algorithm, how we employed CUDA, MPI, and parallel I/O, dataset descriptions, setup configuration, and other relevant information in detail. Then we presented the experimental results, their implications, and the future direction of this research.

REFERENCES

[1] Imielienskin T. and Swami A. Agrawal R., "Mining Association Rules Between set of items in large databases," in Management of Data, 1993, p. 9.

[2] H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri R. Agrawal, "Fast Discovery of Association Rules," in Advances in Knowledge Discovery and Data Mining, 1996, pp. 307-328

[3] Borgelt C., "Efficient Implementations of Apriori and Eclat," in 1st IEEE ICDM Workshop on Frequent Item Set, 2003, p. 9.

[4] Srinivasan Parthasarathy, and Wei Li Mohammed Javeed Zaki, "A Localized Algorithm for Parallel Association Mining," in In 9th ACM Symp. Parallel Algorithms & Architectures. , 1997.

[5] WeeKeong, YewKwong Amitabha Das, "Rapid Association Rule Mining," in Information and Knowledge Management, Atlanta, Georgia, 2001, pp. 474-481

[6] Jian Pei, Jiawei Han, "Mining Frequent patterns without candidate generation," in SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data, New York, NY, USA, 2000, pp. 1-12.

[7] Nasreen, S., Azam, M.A., Shehzad, K., Naeem, U. and Ghazanfar, M.A., 2014. Frequent pattern mining algorithms for finding associated frequent patterns for data streams: a survey. Procedia Computer Science, 37, pp.109-116.

[8] Ye, Y., & Chiang, C. C. (2006). A parallel apriori algorithm for frequent itemsets mining. Proceedings - Fourth International Conference on Software Engineering Research, Management and Applications, SERA 2006, 87–94. https://doi.org/10.1109/SERA.2006.6

[9] Yu, K. M., & Zhou, J. L. (2008). A weighted load-balancing parallel apriori algorithm for association rule mining. 2008 IEEE International Conference on Granular Computing, GRC 2008, 756–761. https://doi.org/10.1109/GRC.2008.4664768

[10] Shah, K. D., & Mahajan, S. (2009). Maximizing the efficiency of parallel Apriori algorithm. ARTCom 2009 - International Conference on Advances in Recent Technologies in Communication and Computing, 107–109. https://doi.org/10.1109/ARTCom.2009.73

[11] Li, N., Zeng, L., He, Q., & Shi, Z. (2012). Parallel implementation of the apriori algorithm based on MapReduce. Proceedings - 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD 2012, 236–241. https://doi.org/10.1109/SNPD.2012.31

[12] Verma, N., & Singh, J. (2017, October 2). A comprehensive review from sequential association computing to Hadoop-MapReduce parallel computing in a retail scenario. Journal of Management Analytics, Vol. 4, pp. 359–392. https://doi.org/10.1080/23270012.2017.1373261

[13] Yahya, O., Hegazy, O., & Ezat, E. (2012). An efficient implementation of the apriori algorithm based on the Hadoop-MapReduce model. International Journal of Reviews in Computing, 12, 59–67

[14] Yang, X. Y., Liu, Z., & Fu, Y. (2010, June). Mapreduce is a programming model for the association rules algorithm on Hadoop. In Information sciences and interaction sciences (ICIS), 2010 3rd international conference, China (pp. 99–102). IEEE

[15] Albert, D. W., Fayaz, K., & Babu, D. V. (2014). HSApriori: high-speed association rule mining using apriori based algorithm for GPU. *Int. J. of Multidisciplinary and Current research*.

[16] Borgelt, C. (2003, November). Efficient implementations of apriori and eclat. In *FIMI'03: Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations*.

[17] Tiwary, M., Sahoo, A. K., & Misra, R. (2014, December). Efficient implementation of the apriori algorithm on HDFS using GPU. In *2014 International Conference on High-Performance Computing and Applications (ICHPCA)* (pp. 1-7). IEEE

[18] Adil, S. H., & Qamar, S. (2009, October). Implementation of association rule mining using CUDA. In *2009 International Conference on Emerging Technologies* (pp. 332-336). IEEE.

[19] Silvestri, C., & Orlando, S. (2012, February). GPU DVI: Exploiting GPUs in frequent itemset mining. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (pp. 416-425). IEEE.

[20] Li, Y., Xu, J., Yuan, Y. H., & Chen, L. (2017). A new closed frequent itemset mining algorithm based on GPU and improved vertical structure. *Concurrency and Computation: Practice and Experience*, *29*(6), e3904.

[21] Agrawal, R., & Srikant, R. (n.d.). *Fast Algorithms for Mining Association Rules*.

[22] Zaki, M. J., & Meira, Jr, W. (2014). Data Mining and Analysis. In *Data Mining and Analysis*. https://doi.org/10.1017/cbo9780511810114