

MR-FIMNA: An Efficient N-Lists-Based Algorithm for Mining Frequent Itemsets via the Hybrid Parallel

Hao-Yu Gu¹, Bin-Bin Guo¹, Ke Gong², Chi Zhang³,
Neelakandan Chandrasekaran¹, and De-Cheng Miao^{1*}

¹ School of Information Engineering, Shaoguan University,
Shaoguan 512000, China

Ghy2022@sgu.edu.cn, gbb197750@163.com, 2264226916@qq.com, deansgu@163.com

² China United Network Communications Group Co., Ltd. Jiangxi Branch,
Nanchang 330029, China
gongk12@chinaunicom.cn

³ School of Information Engineering, Jiangxi University of Science and Technology,
Ganzhou 341000, China
781287727@qq.com

Received 18 April 2025; Revised 25 April 2025; Accepted 26 April 2025

Abstract. Frequent itemset mining (FIM), with their compound correlation structure and powerful association mining capabilities, have been successfully used in retail, fast selling, e-commerce, finance and other fields, and has shown great advantages. However, with the increasing scale of data and the expectation of the response time, FIM faces three complex challenges in a big data environment: inefficient parallelism, inefficient merge performance and redundant search. To solve these three problems, this paper proposes an optimization parallel FIM algorithm (MR-FIMNA) in the MapReduce framework. Firstly, a grouping technique based on greedy strategy of 0-1 knapsack (GM-GSK) is developed in the stage of grouping frequent 1-itemset to diminish the limitations initiated by clusters load balance in the parallel algorithm. Then, a previously abandon strategy is proposed in the stage of mining frequent itemsets in parallel to improve the merge performance of *N-list* structure and a pruning strategy of equivalent superset is proposed to avoid redundant searches during data mining. The MR-FIMNA algorithm was compared with other algorithms on four datasets, namely HIGGS, Adult, Susy and HTRU2. The results of experimental show that the MR-FIMNA algorithm gains a good-performing speed-up ratio and take fewer computing resources and memory usage in a big data environment.

Keywords: frequent itemset mining, MapReduce, N-list, set-enumeration tree, load balancing

1 Introduction

With the rapid advancement and evolution of internet technology in recent years, various services and applications have generated unprecedented volumes of data, often referred to as big data. This data is characterized by its volume, velocity, and variety, posing significant challenges for traditional data processing techniques. To address these challenges, data mining methods [1] have been widely adopted to extract meaningful patterns, trends, and insights from large and complex data sets. Among these methods, frequent itemset mining (FIM) has emerged as a fundamental and widely used technique for discovering recurring patterns in transactional data. Other related data mining techniques include clustering methods for group structure discovery [2], robust regression models for handling complex data relationships [3], feature selection approaches for high-dimensional data analysis [4], ensemble-based classification techniques [5], and fuzzy rule-based systems for intelligent decision-making. Frequent itemset mining focuses on identifying sets of items that frequently co-occur in a given data set, which is essential for applications such as market basket analysis, recommendation systems, and customer behavior analysis. The significance of FIM lies in its ability to uncover hidden relationships and dependencies within data, enabling organizations to make informed decisions and optimize their operations. Traditional

* Corresponding Author

FIM algorithms, such as Apriori [6], FP-Growth [7], and Eclat [8], have been the cornerstone of this field. The Apriori algorithm, for instance, employs a breadth-first search strategy to generate frequent itemsets by iteratively pruning the search space based on the anti-monotonicity property. FP-Growth, on the other hand, utilizes a tree-based structure to compress the data and improve mining efficiency, while Eclat adopts a depth-first search approach with a vertical data representation to reduce computational overhead. Despite their effectiveness, traditional FIM algorithms face significant limitations in the context of big data. The exponential growth of data volume, velocity, and variety has exposed the computational and scalability constraints of these methods. Processing massive data sets on single machines often results in prolonged execution times, high memory consumption, and limited scalability. For example, the Apriori algorithm suffers from multiple database scans and a large number of candidate itemsets, making it inefficient for large-scale data. Similarly, FP-Growth and Eclat, while more efficient than Apriori, still struggle to handle data sets that exceed the memory capacity of a single machine. These challenges hinder the ability to extract valuable insights from large-scale data in a timely and efficient manner.

In order to enhance the computational capacity for processing big data and overcome the limitations imposed by the memory constraints of traditional systems, Google Inc. introduced the MapReduce programming model [9]. MapReduce is a highly scalable parallel computing framework designed for distributed processing of large-scale data sets. Its core strength lies in its ability to divide complex tasks into smaller sub-tasks, which are then processed in parallel across a cluster of machines. This approach not only improves computational efficiency but also enables the handling of data sets that exceed the memory capacity of a single machine. The MapReduce framework consists of two key phases: the Map phase, which processes and partitions the input data into intermediate key-value pairs, and the Reduce phase, which aggregates and summarizes the results. This model has become a cornerstone in big data processing due to its fault tolerance, scalability, and ease of use.

Given the inherent advantages of the MapReduce framework, researchers have developed numerous algorithms based on this model to address the challenges of big data analysis. For instance, in 2016, Vasoya and Koli [10] proposed a parallel Apriori algorithm based on the MapReduce framework. This algorithm leverages the iterative nature of the Apriori approach, which generates candidate itemsets of increasing length in each iteration and prunes those that do not meet the minimum support threshold. By adapting this approach to the MapReduce model, the authors enabled the algorithm to process large data sets in a distributed manner. The Map phase is responsible for counting the frequency of itemsets, while the Reduce phase aggregates the results and identifies frequent itemsets. This parallelization significantly reduces the computational burden and allows the algorithm to handle data sets that are orders of magnitude larger than those manageable by traditional methods.

However, despite its advancements, the MapReduce-based Apriori algorithm still faces three significant limitations: 1) Inefficient Parallelism: While MapReduce inherently supports parallel processing, the algorithm's iterative nature and dependency between iterations often lead to suboptimal resource utilization. Tasks cannot be fully parallelized due to synchronization requirements, resulting in idle nodes and underutilized hardware during certain phases of execution. 2) Inefficient Merge Performance: The Reduce phase, which aggregates intermediate results from the Map phase, suffers from high communication and data shuffling overhead. Merging partial results from distributed nodes becomes a bottleneck, especially when dealing with large-scale data sets, as the process consumes significant time and network bandwidth. 3) Redundant Search: The algorithm generates and evaluates a large number of candidate itemsets, many of which are redundant or do not meet the minimum support threshold. This repetitive search process not only wastes computational resources but also increases the overall execution time, making the algorithm less efficient for processing massive data sets. These limitations highlight the need for further optimization and innovation to enhance the scalability and efficiency of MapReduce-based Apriori algorithms in big data environments.

In response to the challenges outlined above, this paper introduces a parallel FIM algorithm, MR-FIMNA, which integrates the MapReduce distributed computing framework with advanced grouping and pruning techniques. By leveraging the parallel processing capabilities of MapReduce, the algorithm significantly enhances the efficiency of frequent itemset mining (FIM) in large-scale datasets. Furthermore, the framework's scalability effectively addresses the challenges of data volume and computational complexity, enabling faster and more resource-efficient mining processes. This method paves the way for more efficient, robust, and scalable association rule mining systems in big data environments. The core contributions of the MR-FIMNA algorithm include the following three aspects:

(1) Grouping Technique Based on Greedy Strategy of 0-1 Knapsack (GM-GSK)

An efficient grouping technique, GM-GSK, is proposed to address the challenge of inefficient parallelism in frequent itemset mining. By leveraging the greedy strategy of the 0-1 knapsack problem, GM-GSK dynamically balances the workload across distributed nodes, ensuring optimal resource utilization. This technique minimizes the limitations caused by cluster load imbalance, significantly improving the parallel efficiency of the algorithm.

Furthermore, GM-GSK reduces redundant computations and enhances the overall scalability of the mining process, making it suitable for large-scale datasets.

(2) Previously Abandoned Strategy for Merge Performance Optimization (PAS-MPO)

A previously abandoned strategy, PAS-MPO, is introduced to tackle the issue of inefficient merge performance in the MapReduce framework. This strategy optimizes the merging of intermediate results by reducing the communication overhead and data shuffling between the Map and Reduce phases. By restructuring the merge process and eliminating unnecessary computations, PAS-MPO significantly accelerates the aggregation of frequent itemsets. This improvement ensures faster and more efficient processing of large-scale datasets, addressing one of the critical bottlenecks in traditional MapReduce-based FIM algorithms.

(3) Pruning Strategy of Equivalent Superset (PSES)

A pruning strategy of equivalent supersets, PSES, is proposed to eliminate redundant searches during the frequent itemset mining process. By identifying and discarding supersets that do not contribute to the final results, PSES reduces the computational complexity and memory usage of the algorithm. This strategy not only minimizes redundant search operations but also enhances the overall efficiency and accuracy of the mining process. Combined with the parallel capabilities of MapReduce, PSES ensures that the algorithm can handle large-scale datasets with improved speed and reduced resource consumption.

The remainder of the paper will provide a comprehensive exploration of the proposed MR-FIMNA algorithm and its contributions. Section 2 reviews current approaches to parallelizing frequent itemset mining (FIM) and critically examines their limitations in big data environments. Section 3 introduces the PPC-Tree structure, which serves as a foundational concept for optimizing the mining process. Section 4 discusses the technical aspects of the MR-FIMNA algorithm, focusing on its innovative strategies for grouping, merging, and pruning to enhance frequent itemset mining efficiency. Section 5 presents a detailed analysis of the experimental results, comparing the performance of the algorithm with state-of-the-art methods and evaluating its scalability and efficiency on large-scale datasets.

2 Related Work

Frequent Itemset Mining (FIM) is a fundamental technique in data mining, widely used for discovering meaningful patterns and associations in large datasets. It is known for its ability to identify recurring item combinations, efficient rule generation, and scalability in transactional data analysis. These characteristics make FIM particularly effective in applications such as market basket analysis, recommendation systems, and customer behavior analysis. For instance, Vasoya and Koli [10] proposed an improved Apriori algorithm that integrates the MapReduce framework with a dynamic pruning strategy. This model utilizes parallel processing to distribute the mining tasks across multiple nodes and employs a candidate reduction mechanism to minimize redundant computations. Experimental results demonstrate that the model significantly outperforms traditional methods in terms of computational efficiency and scalability for large-scale datasets. However, a critical limitation of this approach is its inefficient parallelism.

To address this issue, Shaikh et al. [11] proposed the DIAFM (Distributed Incremental Approximate Frequent Itemset Mining) algorithm, which introduces a distributed incremental processing framework that focuses only on new data increments, avoiding the need to reprocess historical data. This approach significantly reduces computational overhead and improves runtime efficiency. Additionally, DIAFM employs a shard-based approximate mining strategy, dividing the dataset into multiple shards for parallel processing, ensuring a balance between efficiency and accuracy. The algorithm also leverages the MapReduce framework for optimized load balancing and network communication, making it highly scalable in distributed environments such as Hadoop. Despite these advancements, another significant challenge remains: inefficient merge performance.

Building on the distributed approach, Wu et al. [12] developed the MR-HUPSPM (MapReduce-based High Utility Probability Sequence Pattern Mining) algorithm to efficiently mine uncertain sequence patterns from large-scale datasets. Unlike traditional single-machine algorithms, MR-HUPSPM utilizes the Hadoop platform to distribute tasks across multiple nodes, significantly improving computational efficiency. The algorithm introduces two pruning strategies during the initialization phase to eliminate unpromising candidates, thereby reducing the search space and enhancing performance. Moreover, MR-HUPSPM addresses the challenges of processing uncertain data, such as those collected by IoT sensors, by incorporating a probabilistic model. Experimental results demonstrate its superior performance in terms of runtime and memory consumption, particularly for large datasets. Further extending the capabilities of distributed pattern mining, Wu et al. [13] proposed the SFUP-MR

(Skyline Frequent-Utility Patterns Mining via MapReduce) algorithm, which mines frequent high-utility patterns from large-scale datasets using the MapReduce framework. Unlike traditional FIM and HUIM (High Utility Itemset Mining) methods, SFUP-MR integrates a Skyline framework to simultaneously consider frequency and utility dimensions, enabling more comprehensive decision-making. The algorithm employs three pruning strategies—sub-tree utility, local utility, and Utility-Max array—to efficiently reduce the candidate search space. Additionally, SFUP-MR introduces a logical search graph structure tailored for big data environments, replacing traditional tree and list structures to enhance scalability. Experimental validation confirms its superior performance in handling large datasets, outperforming traditional single-machine algorithms like SFUPMiner and SFUI-UF. However, both MR-HUPSPM and SFUP-MR still face a critical limitation: redundant search, where the algorithms generate and evaluate a large number of candidate patterns that do not contribute to the final results, leading to unnecessary computational overhead and reduced efficiency.

3 Preliminary

Deng [14] presented the PrePost algorithm for fast mining frequent itemsets in 2012. He used the N-list structure to store frequent itemsets and achieved good performance while mining frequent itemsets. The related definitions and characteristics of PrePost are mentioned below:

Definition 1 (N-list [15] of frequent itemsets) For any *PPC-Tree*, The N-list of a frequent itemsets is the PP-code of all the item nodes registered in the PPC-tree. The PP-codes are sorted in ascending order of their pre-node values.

Definition 2 (The ancestor-offspring relationship of PP-codes) For any two nodes R_1 and R_2 , R_2 is a descendant of R_1 , if and only if:

$$R_1.pre - node > R_2.pre - node \quad (1)$$

$$R_1.post - node > R_2.post - node \quad (2)$$

Definition 3 (N-list of k-itemsets) Given any two frequent (k-1)-itemsets with same prefix regard as XA and XB , and their corresponding N-list structures are:

$$N - list(XA) = \{(a_{11}, b_{11}, c_{11}), (a_{12}, b_{12}, c_{12}), \dots, (a_{1m}, b_{1m}, c_{1m})\} \quad (3)$$

$$N - list(XB) = \{(a_{21}, b_{21}, c_{21}), (a_{22}, b_{22}, c_{22}), \dots, (a_{2m}, b_{2m}, c_{2m})\} \quad (4)$$

a_{ij} is the pre-order traversal of the sequence generated by ascending joins. b_{ij} is the post-order traversal of the sequence generated by descending joins. c_{ij} is the number of records on each item. The N-list of the k-itemsets XAB , is defined as follows:

For any $(a_{1p}, b_{1p}, c_{1p}) \in N-list(XA)$, $(1 \leq p \leq m)$, $(a_{2q}, b_{2q}, c_{2q}) \in N-list(XB)$, $(1 \leq q \leq n)$ if condition $a_{1p} < a_{2q}$, $b_{1p} > b_{2q}$ is satisfied, put (a_{1p}, b_{1p}, c_{2q}) in N-list of XAB , and we get initial N-list.

Traverse *N-list* of XAB , merge the same *PP-code* between *pre-node* and *post-node*. Then we can get final N-list.

Definition 4 (The support for frequent itemsets) Given the N-list of frequent itemsets X , which is denoted by $\{(a_1, b_1, c_1), (a_2, b_2, c_2), \dots, (a_m, b_m, c_m)\}$, the support of the item X is $\sum_{i=1}^m c_i$.

Definition 5 (relation [16]) For i_1 and i_2 , which are two frequent items in L_1 , $i_1 \prec i_2$ if and only if i_1 is prior to i_2 in L_1 .

4 Algorithm MR-FIMNA

Here, the MR-FIMNA algorithm is described and analyzed in detail. Our algorithm involves 3 stages: frequent 1-itemsets obtaining, frequent 1-itemsets grouping, frequent itemsets mining in parallel. In the first stage, for obtaining the F -list from the database in parallel, we execute a MapReduce task and use a method like World Count. In the second stage, the GM - GSK is presented to divide the calculate node uniformly during data partitioning, and we design a load estimation function named $LCEF$ to compute the load of every item in frequent 1-itemsets, and then group F -list evenly to generate the G -list (grouping list). In the last stage, the frequent itemsets will be mined in the PPC-subtree which is created based on F -list and G -list.

4.1 Frequent 1-itemset Obtaining

In order to produce F -list with frequent 1-itemsets, we need to execute a MapReduce task, which needs to go through the four stages of Split, Map, Combine and Reduce. In Split phase, we split the data into file blocks of the equivalent size and save them in HDFS. In Map phase, these data will be plotted as key-value pairs. In the Combine phase, these key-value pairs which has the same key will be merged locally to reduce data processing costs. In reduction phase, global key-value pairs are combined with equivalent key which will lead to support and individual time. In the end, the F -list is generated through minimum support ($Min_Support$).

Then we show the pseudo code of the frequent 1-itemsets algorithm, which is implemented as follows:

Algorithm 1. Frequent 1-itemsets obtaining

Inputs: Database DB , $min_support$;
Output: frequent 1-itemsets F -list ;
a): **Procedure:** Mapper ($key, value$) = T_i
for each $item$ in T_i **do**
 if $item$ not null **then**
 output $\langle key = item, value = 1 \rangle$
 end if
end for
Procedure: Reducer ($key, values$) = $\langle sup(key)_1, sup(key)_2, \dots, sup(key)_m \rangle$
Support = 0
 $F_1 = \emptyset$
for each sup in values **do**
 support += sup
end for
if (support $\geq min_support$) **do**
 $F_1 = F_1 \cup \langle key = item, value = support \rangle$
end if
Sorted (F_1) //sort F_1 by value

4.2 Frequent 1-itemsets Grouping

At this stage, the steps of GK-GSK are described in detail. And we designed a load capacity estimation function ($LCEF$) to compute the load of individual item in frequent 1-itemsets. The $LCEF$ function defined as follows:

Theorem 1 ($LCEF$) For any given frequent items, it's support degree and position in F -list is defined as $count$ and loc respectively. The $LCEF$ can be defined as:

$$LCEF (item) = \min \{ count, 2^{loc-1} \} \quad (5)$$

Proof: For a given frequent item, the number of nodes in the PCC -Tree equals the length of N -list. It is obvious that the current item is supported by most of the nodes of PPC-tree and the number of these nodes in individual item in the tree corresponds to its situation in F-list sequence. In the worst case, there are at most 2^{loc-1} paths when the $loc-1$ items are randomly combined before i . The result of the merge has a corresponding path

in the PPC-Tree, and the path also contains item. Thus, the length of each N -list in the F -list is greater than the support of current term and less than 2^{loc-1} .

Based on Property 1, the idea of GM - GSK strategy is described as follows as below:

1) Use equation (2) to calculate the result of $LCEF(item)$ in F -list, then sort F -list by $LCEF(item)$ to get the L -list;

2) Read those items in L -list and add them to G based on the following rules. First, we calculate the average load of all items marked as avg , if avg is smaller than g_load (the current item load), put current item in a group with lower load and delete this item from L -list. Next, we traverse the L -list in reverse order, if g_load is satisfied with Formula (6), put this item into the current group. Then, we put this item into the next group. Finally, the current term is deleted in the L -list.

3) Store the G -list in $HDFS$ to guarantee accessibility to G -list results for every node in the cluster.

$$abs(item.load + g_load - avg) < abs(g_load - avg) \quad (6)$$

The GM - GSK grouping method algorithm is implemented as follows:

Algorithm 2. The process of GM - GSK

Inputs: Frequent 1-itemsets F -list, Number of clusters G ;

Output: G -list;

a): **Procedure:** Calculating the load of each item in F -list

```

1:    $L$ -list  $\leftarrow \emptyset$ 
2:   For each item in  $F$ -list do
3:     Compute  $item\_load$  by Eq. (5)
4:      $L$ -list  $\leftarrow L$ -list  $\cup$   $key = item, value = item\_load$ 
5:   End for
6:    $Sorted(L$ -list)//sort  $L$ -list in non-decreasing order by  $value$ 
7:   Return  $L$ -list

```

b): **Procedure:** Calculate the average load for all items avg

```

1:    $sum\_load = 0$ 
2:   For each  $\langle key, value \rangle$  in  $L$ -list do
3:      $sum\_load += value$ 
4:   End for

```

c): **Procedure:** Grouping F -list uniformly

d): **Setp1:** Grouping items in L -list whose load is greater than avg

1: **Function** getPartition1(L -list, G , avg)

```

2:    $G$ -list  $\leftarrow \emptyset$ 
3:    $i = 0$ 
4:   while  $i < G$  and  $L$ -list  $\neq$  null do
5:      $G$ -list [ $i$ ]  $\leftarrow \emptyset$ 
6:      $curNode \leftarrow (L$ -list)[0]
7:     if  $curNode.value \geq avg$  do
8:        $G$ -list [ $i$ ].add( $curNode$ )
9:        $L$ -list.remove( $curNode$ )
10:     $G$ -list.add( $G$ -list [ $i$ ])
11:     $i = i + 1$ 
12:   else
13:     Break
14:   End if
15:   End while

```

e): **Setp2:** Grouping items in L -list whose load is smaller than avg

1: **Function** getPartition2(L -list, $remain_G$, G -list)

```

2:   For (int  $i = 0$ ;  $i < remain\_G$ ;  $i ++$ ) do
3:      $G$ -list [ $i$ ]  $\leftarrow \emptyset$ 

```

```

4:     groupSum = 0
5:     For (int j = L-list.size(); j > 0; j--) do
6:         curNode ← (L-list)[0]
7:         If abs(groupSum+curNode.value-avg) < abs(g_load-avg) do
8:             groupSum+=curNode.value
9:             G-list [j].add(curNode)
10:            L-list.remove(curNode)
11:        End if
12:    End for
13:    G-list.add(G-list[i])
14: End for

```

4.3 Frequent Itemsets Mining in Parallel

At the last stage, we divided the F -list into multiple blocks evenly through the GM - GSK strategy to get the G -list. Next, we map the divided dataset to each node according to the G -list. The PPC-subtree will be created by these plotted data to mine frequent itemsets. The process needs to go through two phases of Map and Reduce:

Map Phase. The main task of this phase is to map the divided dataset to each computing node according to G -list, the detailed process is: Firstly, the MapReduce task read the G -list and F -list from HDFS. Secondly, the algorithm makes the items in G -list as *key*, makes the G-number as *values* to establish the $HTable$. Then, the task traverses the *items* in reverse order, and form *key-value* pairs whose *gid* is greater than the current *item*. At the same time, the *key-value* pairs that the *value* is the same as *gid* will be removed to evade the question that the same *item* may be mapped multiple times to the same node. After that, the task will put the *key-value* pairs to the Reduce node. The detailed process is shown in Algorithm 3. The algorithm is implemented as follows:

Algorithm 3. The Map step of mining frequent items

Inputs: Preprocessed data pre_data , F -list, G -list;

Output: Mapping path $\langle key, value = path \rangle$;

a): **Procedure:** Create a hash table $HTable$

```

1:     Read  $G$ -list from HDFS
2:     For each  $g$  in  $G$ -list do
3:         For each item in group do
4:              $HTable[group[item]] = gid$ 
5:         End for
6:     End for
b): Procedure: Generate map path
1:      $res \leftarrow \emptyset$ 
2:     For each trans in  $pre\_data$  do
3:          $items[] \leftarrow Split(trans)$  //decomposed the input data and store
         them in  $items[]$ 
4:         For  $j = len(items)-1$  to 0 do
5:             if  $HTable[items[j]]$  not null do //determine which group
         path  $items[j]$  belongs to
6:                  $path = \{items[0], items[1], \dots, items[j]\}$ 
7:                  $res = res \cup \{\langle key = HTable[j], value = path \rangle\}$ 
8:             End if
9:             del( $HTable[items[j]$ )
10:        End for
11:    End for
12:    Return  $res$ 

```

Reduce Phase. In the Reduce phase, the *insert_tree()* function is first called to generate PPC-Tree structures at each node of the cluster, and the PPC-Tree is traversed in pre-order and post-order to construct the N-list structure for 2-itemsets. The process mainly consists of two parts: 1) Frequent Itemset Merging: Subsequently, the *PAS* strategy is proposed to accelerate the merging process of the N-list structures of two frequent itemsets; 2) Reducing Redundant Searches: Finally, the set enumeration tree is adopted as the search space, and the *PSES* strategy is introduced to avoid redundant searches during the mining process, optimizing the search space and generating the final mining results.

1) Frequent Itemset Merging: Subsequently In the process of parallel mining frequent itemsets, the most important step is to generate frequent $(k+1)$ -itemsets by merging the *N-list* structure of two frequent k -itemsets. How to reduce the running time of the merging process is the most important problem of this algorithm. For this reason, we propose a pre-abandon strategy named *PAS*, which can reduce the invalid calculation in the merging process. Therefore, this strategy greatly improved the merging efficiency of *N-list* structure.

Given any two frequent k -itemsets, the *N-list* structure is expressed as $N-list_1$ and $N-list_2$ respectively, and their length are m and n . Their forms are as follows.

$$N-list_1 = \{(a_{11}, b_{11}, c_{11}), (a_{12}, b_{12}, c_{12}), \dots, (a_{1m}, b_{1m}, c_{1m}), \} \quad (7)$$

$$N-list_2 = \{(a_{21}, b_{21}, c_{21}), (a_{22}, b_{22}, c_{22}), \dots, (a_{2n}, b_{2n}, c_{2n}), \} \quad (8)$$

The process of merging $N-list_1$ and $N-list_2$ by using *PAS* strategy is as follows:

First of all, according to Definition 4, we calculate sF_1 (the support degree of $N-list_1$) and sF_2 (the support degree of $N-list_2$) respectively, then we add sF_1 and sF_2 to get sF (the total support degree of $N-list$). For any PP-code C_i , when C_i is not satisfied with Definition 5, update sF to $sF - C_i * count$. When sF is smaller than $Min_Support$, we consider the current itemset is infrequent. Therefore, the comparison process should be terminated. The *PAS* strategy is implemented as follows:

Algorithm 4. The execution process of *PAS*

Inputs: minimum support threshold $Min_Support$, $N-list_1$, $N-list_2$;

Output: $N-list_3$

Procedure: NL-intersection ($N-list_1$, $N-list_2$)

- 1: $N-list_3 \leftarrow \emptyset$
- 2: $sF_1 \leftarrow$ the support of $N-list_1$
- 3: $sF_2 \leftarrow$ the support of $N-list_2$
- 4: $sF = sF_1 + sF_2$
- 5: $i = 0, j = 0$
- 6: while $i < |N-list_1|$ and $j < |N-list_2|$ do
- 7: if $N-list_1[i].pre < N-list_2[j].pre$ then
- 8: if $N-list_1[i].post < N-list_2[j].post$ then
- 9: if $N-list_3[i] > 0$ and pre-value of the last element in $N-list_3$ equal to $N-list_1[i].pre$ then
- 10: Increase the *sum* value of the last element in $N-list_3$ by $N-list_2[j].sum$
- 11: **Else do**
- 12: $N-list_3[i].add(< N-list_1[i].pre, N-list_1[i].post, N-list_2[j].sum >)$
- 13: $j += 1$
- 14: **Else do**
- 15: $sF = sF - N-list_1[i].sum$
- 16: $i += 1$
- 17: **Else do**
- 18: $sF = sF - N-list_1[i].sum$
- 19: $j += 1$
- 20: **if** $sF < Min_Support$ do

21: **return** null
 22: **return** $N-list_3$

2) Reducing Redundant Searches: Since the algorithm adopts Apriori-like approach to mining frequent itemsets, which will lead to many redundant searches during the mining process. In order to optimize the mining efficiency of this algorithm, we use the set-enumeration tree as the search space when algorithm mine the frequent itemsets, then we propose *PSES* strategy to avoid redundant searches in the mining process and generate the final mining results.

Given a set of itemsets $I = \{i_1, i_2, \dots, i_m\}$, and $i_1 \prec i_2 \prec \dots \prec i_m$, the construction process of the set-enumeration tree is as follows: 1) Create the root node; 2) Take out each i_x ($1 \leq x \leq m$) in itemset as the child node of the root node in turn; 3) Intersect each node with other nodes on the left side of the *F-list*, then we can get their child node; 4) Repeat process 3 until all leaf nodes are generated.

Theorem 2 (PSES) Given itemset S and item i , if the support of S is equal to $S \cup \{i\}$'s support $sup(S \cup \{i\})$, then for any itemset A , ($A \cap S = \emptyset \wedge i \notin A$) satisfies the following relationship:

$$sup(A \cup C.head) = sup(A \cup C.head \cup \{i\}) \quad (9)$$

Proof: For itemsets P , its support is equal to the support of $P \cup \{i\}$, this means that any included P transaction also includes item i , given a transaction T , if T contains itemset $A \cup P$, T contain itemset A and itemset P , Then it can be deduced that transaction T must also contain item i , it means the support of $A \cup P$ is equal to that of $A \cup P \cup \{i\}$.

Algorithm 5. The reduce phase of parallel mining frequent

Inputs: mapping path, minimum support threshold $Min_Support$,

frequent 1-itemsets F_1 ;

Output: frequent itemsets F

Procedure: frequent itemsets F

```

1:   Create the PPC-tree by calling insert_tree()
2:   Traverse the PPC-Tree to build the N-list of frequent
     1-itemsets
3:   Traverse the PPC-Tree to get the set of frequent 2-itemsets
4:   For each  $i_s, i_t (i_s, i_t \in F_2)$ , build its N-list by using NLP_intersection(N-list1, N-list1)
5:    $F \leftarrow F_1$ 
6:   For each itemsets in  $F_2$ , marked as  $i_x i_y$  do
7:     Generate the root of the tree,  $FPT_{xy}$ , and marked as  $i_x i_y$ 
8:     Build_Enumerate_Tree ( $FPT_{xy}, \{i \mid i \in F_1, i \succ i_x\}, \emptyset$ )
9:   Return  $F$ 
Function
Build_Enumerate_Tree ( $NLd, Cad\_items, Par\_fit$ )
1:    $NLd\_equivalent\_items \leftarrow \emptyset$ 
2:    $NLd\_childrensnodes \leftarrow \emptyset$ 
3:    $Next\_Cad\_items \leftarrow \emptyset$ 
4:   For each  $i \in Cad\_items$  do
5:      $NLd\_itemset \rightarrow P_1$ 
6:      $\{i\} \cup (P_1 - P_1[1]) \rightarrow P_2$ 
7:      $\{i\} \cup P_1 \rightarrow P$ 
8:     if  $P.support = P_1.support$  then
9:        $NLd\_equivalent\_items \leftarrow NLd\_equivalent\_items \cup \{i\}$ 
10:    Else if  $P.support \geq min\_support$ , then
11:      Build node  $NLd_i$ 
12:       $i \rightarrow NLd_i.label$ 

```

```

13:    $P \rightarrow Nld\_itemset$ 
14:    $Next\_Cad\_items \leftarrow Nest\_Cad\_items \cup \{i\}$ 
15:    $Nld\_childrensnodes \leftarrow Nd\_childrensnodes \cup \{Nld_i\}$ 
16:   End if
17: End for
18: If  $Nld\_equivalent\_items \neq \emptyset$  then
19:    $Subset \leftarrow$  the set of all subsets of  $Nld\_equivalent\_items$ 
20:    $Cand\_itemsets \leftarrow \{A \mid A = Nld.label \cup A', A' \in Subset\}$ 
21:   If  $Par\_fit \leftarrow \emptyset$  then
22:      $Nld\_fit \leftarrow Cand\_itemsets$ 
23:     Else
24:        $Nld\_fit \leftarrow \{P \mid P' = P_1 \cup P_2, (P_1 \neq \emptyset \wedge P_1 \in$ 
 $Cand\_itemsets \wedge P_2 \neq \emptyset \wedge P_2 \in Parent\_fit)\}$ 
25:     End if
26:      $F \leftarrow F \cup Nld\_fit$ 
27:   End if
28: If  $Nld\_childrensnodes \neq \emptyset$  then
29:   For each child node,  $Nld_i$ , do
30:     Build_Enumerate_Tree ( $Nld_i, \{j \mid j \in Next\_Cad\_items, j \succ i\}, Nld\_fit$ )
31:   End for
32: Else Return
Function insert_tree( $curNode, item$ )
1: If  $curNode.child.contains(item)$  then
2:    $tempNode \leftarrow currentNode.getChild(item)$ 
3:    $tempNode \leftarrow tempNode.count + 1$ 
4:    $curNode \leftarrow tempNode$ 
5: Else
6:   Create a new node,  $newNode$ 
7:    $newNode.name = item$ 
8:    $newNode.count = 1$ 
9:    $curNode.addChild(tempNode)$ 
10:   $curNode \leftarrow newNode$ 
11: End if

```

For explain the algorithm 5 more clearly, we divide the process of mining frequent itemsets in parallel into the following steps:

- (1) Firstly, we call the *insert_tree()* function to generate PPC-tree on each node. In this process,
- (2) Then we traverse the PPC-subtree in preorder and postorder to generate all *N-list* of frequent 1-itemset.
- (3) Finally, the algorithm finds all frequent itemsets suffixed with item in each group. These frequent itemsets constitute the final mining result. In addition, in order to avoid repeated searches during the mining process, the algorithm constructs a local set-enumeration tree in each computing node as the search space, and uses the *PAS* strategy to mine frequent itemsets.

4.4 Time Complexity Analysis

To effectively analyze the time complexity of the MR-FIMNA, the algorithms DIAFM, MR-HUPSPM, and SFUP-MR are selected for comparison.

In the MR-FIMNA algorithm There are three stages, we denote the time complexity of the following three stage as T_1 , T_2 and T_3 respectively:

- (1) In the obtaining frequent 1-itemsets stage. Suppose n is the number of records on each node, S is the average length of records, but S is a constant. The time complexity in Map phase is $T_M = O(n)$. Therefore, the time complexity of this stage is $T_1 = T_M = O(n)$.

(2) In the grouping frequent 1-itemsets stage. The time complexity in this stage is mainly depends on the time consuming of *GM-GSK* strategy. Assume G is the clusters number and n is the number of records on each cluster. Due to the *GM-GSK* strategy requires two iterations. The time complexity of this stage is $T_2 = O(n^2/G^2)$.

(3) In the mining frequent itemsets in parallel stage. Suppose $\{I_1, I_2, \dots, I_{l_1}\}$ is the *F-list* of frequent 1-itemsets and n is the number of records on each cluster. The time complexity is $T_3 = O(n \cdot \log_2(n))$.

Since n is large enough, the time complexity of the MR-FIMNA algorithm is the sum of the above three stages:

$$T_{MR-FIMNA} = O(T_1 + T_2 + T_3) = O\left(n + \frac{n^2}{G^2} + n \cdot \log_2(n)\right) \quad (10)$$

In the DIAFM algorithm, assuming k is the average transaction length, the time complexity is as follows: the dataset is partitioned into G shards, each processed in parallel with a local frequent itemset mining complexity of $O\left(\frac{n^2}{G^2} \cdot k\right)$, followed by a global aggregation and profile integration step with a complexity of $O(G^2/k)$, resulting

in a total time complexity of $O\left(\frac{n^2}{G^2} \cdot k + G^2 \cdot k\right)$, which simplifies to:

$$T_{DIAFM} = O\left(\frac{n^2}{G^2} \cdot k + G^2 \cdot k\right) \quad (11)$$

In the MR-HUPSPM algorithm, the dataset is partitioned into G subsets, each processed in parallel to build N-Lists with $O\left(\frac{n^2}{G^2} \cdot k^2\right)$, followed by global aggregation and pruning with $O(G/k^2)$, the total time complexity is:

$$T_{MR-HUPSPM} = O\left(\frac{n^2}{G^2} \cdot k^2 + G \cdot k^2\right) \quad (12)$$

In the SFUP-MR algorithm, assuming k is the maximum length of an itemset, the time complexity is as follows: the dataset is partitioned into G shards, each processed in parallel with a local skyline frequent-utility pattern mining complexity of $O\left(\frac{n^2}{G^2} \cdot 2^k\right)$, followed by a global aggregation and Skyline filtering step with a complexity of $O(G \cdot 2^k)$, resulting in a total time complexity of $O\left(\frac{n^2}{G^2} \cdot 2^k + G \cdot 2^k\right)$, which simplifies to:

$$T_{SFUP-MR} = O\left(\left(\frac{n^2}{G^2} + G\right) \cdot 2^k\right) \quad (13)$$

Since the value of n is much larger than other metrics in the big data environment, the time complexity of the DIAFM algorithm is smaller than other algorithms, namely, $T_{MR-FIMNA} < \min(T_{DIAFM}, T_{HBPrePos^*}, T_{SFUP-MR})$.

5 Experimental Evaluation

Here, the experimental setup will be discussed, the experimental results as well as the corresponding analysis.

5.1 Experimental Setup

The following four real datasets were used to verify the performance of MR-FIMNA algorithm:

The HIGGS [17] dataset records the data generated by Monte Carlo simulations which is used to measure the motion characteristics of particle detector. The official dataset consists 11000000 records and every record consists of 28 items.

The Adult [18] dataset records the prediction whether income exceeds \$50K/yr based on census data which contains 48842 records, each including 14 items.

The Susy [19] dataset records the usage of particle accelerators particles which contains 5000000 records, each including 190 items.

The HTRU2 [20] is a data set formed by the observation of pulsars in the universe by the School of Physics and Astronomy of the University of Manchester. It consists of 17,898 records, and each record includes 9 items. Table 1 shows the specific information of the data set.

Table 1. Experimental datasets

	HIGGS	Adult	Susy	HTRU2
Number of records	11000000	48842	5000000	17898
Number of items	28	14	190	9
Data size	2662.4	3.8	321	1.5

The experimental environment is as follows: AMD Ryzen 7 5800 CPU, 2TB hard disk, 32GB RAM, Windows 10, Hadoop 2.7.3 with four nodes, JDK 1.8.0. The node configuration is shown in Table 2.

Table 2. The foundation configuration of each node

Node type	Host name	IP	Role
Primary Node	Primary Node	192.168.1.116	Primary /JobTracker /NameNode
Secondary Node	Slaver Node_1~3	192.168.1.117~ 192.168.1.120	Secondary/TaskTracker /DateNode

5.2 Performance Metrics

Speed-up ratio [21] is used as a substantial indicator to compute the parallelization performance of the algorithm. The speed-up ratio is the ratio of time, defined as:

$$S_p = T_1 / T_p \quad (14)$$

where T_1 and T_p denotes the running time of the algorithm on a single node and in parallel respectively. The larger S_p is, the relative time spent in parallel computing is less, and the cluster efficiency is greater.

5.3 Feasibility Analysis of MR-FIMNA

Speedup Ratio Analysis in Different Support. In this experiment, the above 4 datasets are applied. The speed-up ratio of the association rules is used for evaluating the parallelization performance of the MR-FIMNA algorithm. The higher the speedup ratio value the stronger parallelization is. Our experiment chooses 1000, 5000, 20000 and 100000 as minimum support thresholds. All experiments are performed ten times. The average value is taken as the final result. Fig. 1 shows the growth of the speedup ratio on the different number of computing nodes. We can notice that with the increase of support, the MR-FIMNA algorithm has a higher speedup ratio when it processes the two larger datasets of HIGGS and Susy. However, when the algorithm deal with small datasets such as Adult and HTRU2, as the number of nodes increases, the speedup ratio does not increase sig-

nificantly. Even when the support is 100,000, the speedup ratio shows a downward trend and is less than 1. This is because when the algorithm process small-scale datasets, the G-list is divided into different computing nodes, which increases the processing time overhead of each node and reduces the speedup ratio. In large-scale datasets, the algorithm can mine local frequent itemsets and merge frequent itemsets in parallel, which improves the mining performance of the algorithm. All these shows that the MR-FIMNA algorithm is suitable for large datasets and has strong scalability.

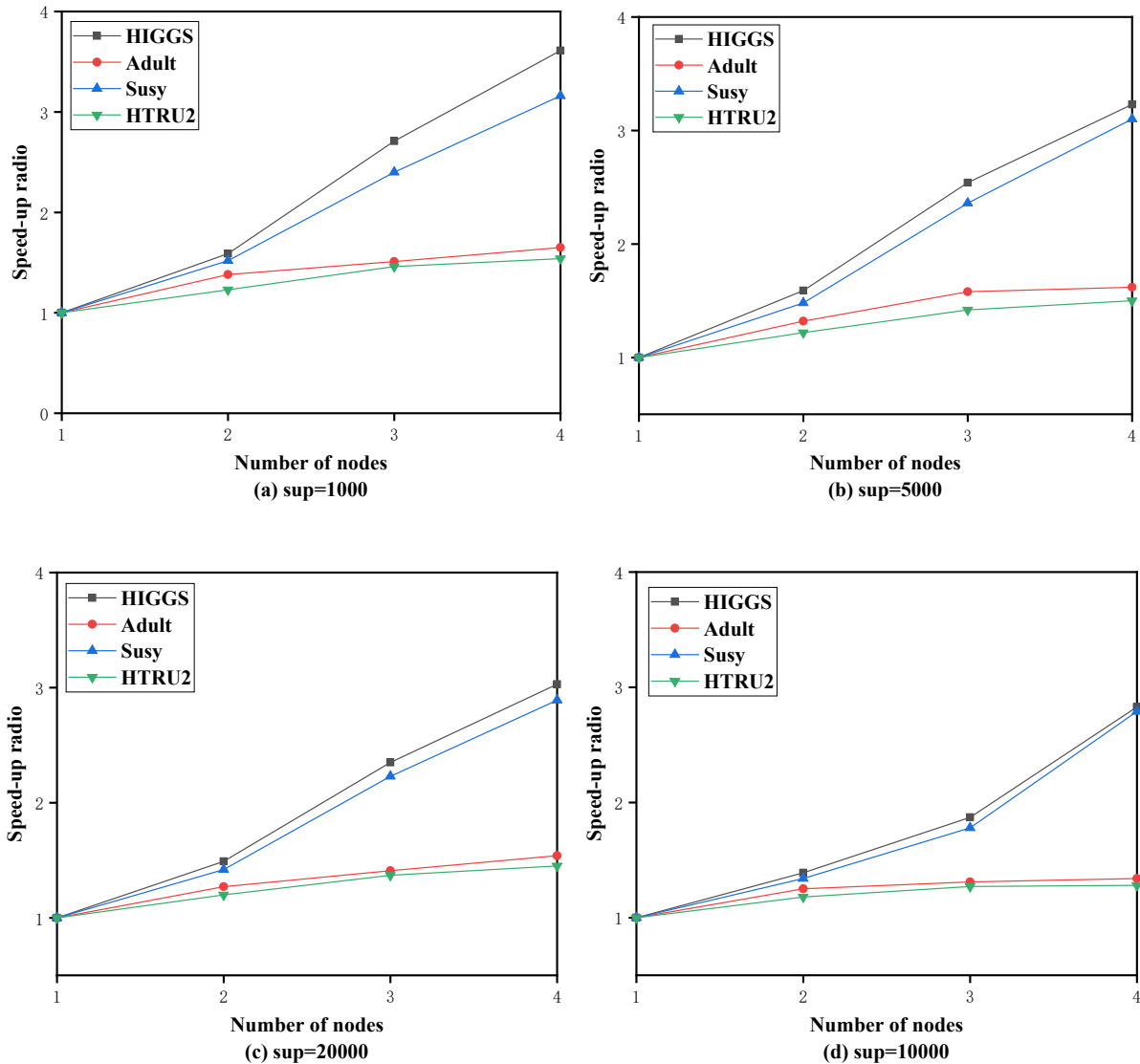


Fig. 1. Speed-up ratio of the MR-FIMNA algorithm under different nodes and different sup

Speedup Ratio Analysis in Different Algorithm. In order to verify the parallelization performance of the MR-FIMNA algorithm in the big data environment, this paper is based on the HIGGS, Adult, Susy and HTRU2 data sets, and uses the speedup ratio as a measure to compare with the DIAFM, MR-HUPSPM, and SFUP-MR algorithms respectively. At the same time, in order to ensure the accuracy of the experimental results, the average running time of each algorithm is 10 times to calculate the speedup ratio as the final experimental result. The experimental results are shown in Fig. 2.

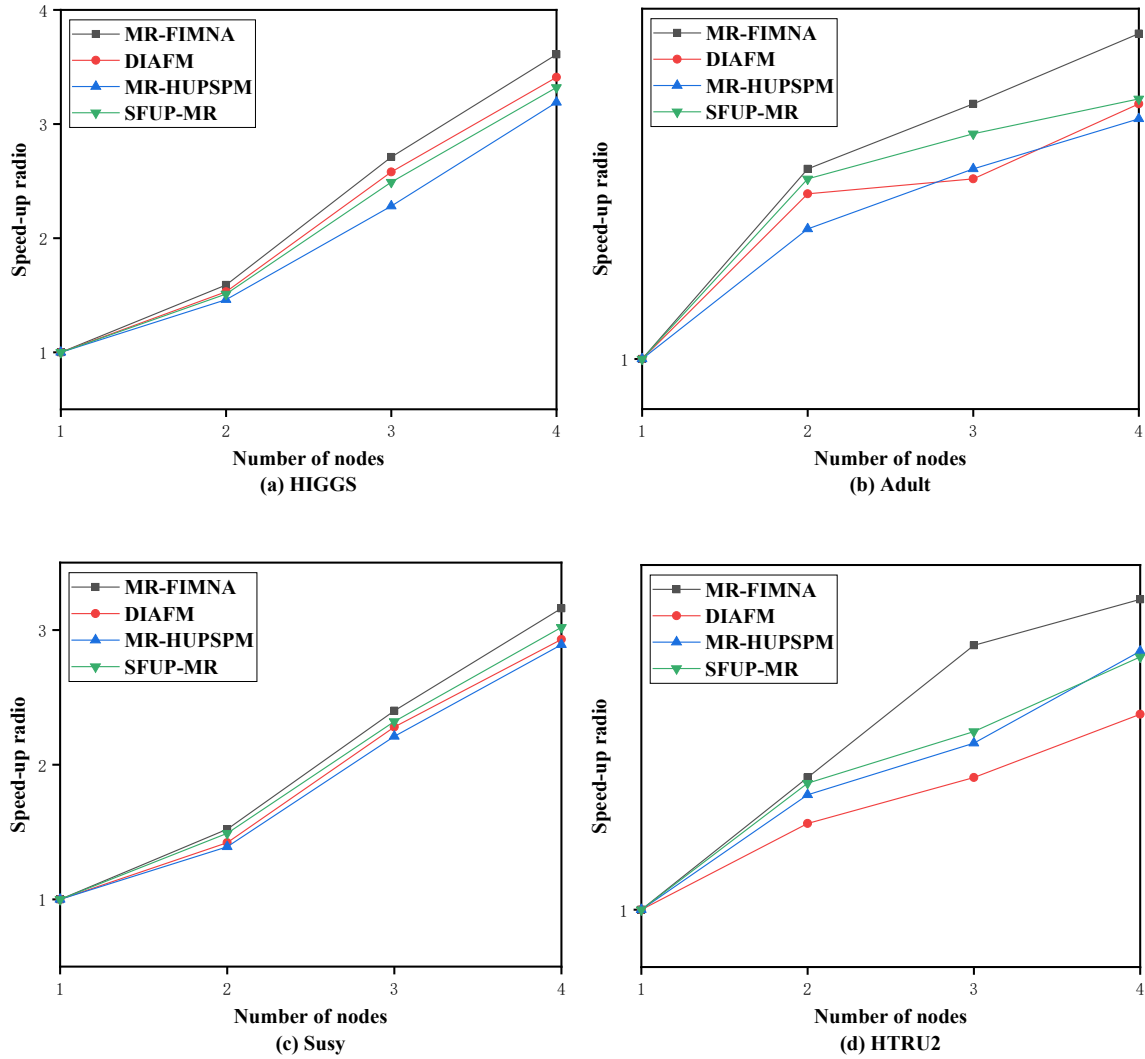


Fig. 2 Speedup ratio of four algorithms under different dataset

We can notice that as the number of nodes increases, the speedup ratio of each algorithm also increases gradually. Among them, the speedup ratio of MR-FIMNA algorithm is better than other algorithms. When the number of nodes is 4, the speedup ratio of the MR-FIMNA algorithm is 0.2, 0.32 and 0.19 higher than that of the DIAFM, MR-HUPSPM, and SFUP-MR algorithms in HIGGS, respectively. This is because the algorithm can mine local frequent itemsets and merge frequent itemsets in parallel, which improves the mining performance of the algorithm. And the PAS strategy is proposed to determine whether the current item is a frequent itemset. The PAS strategy simplifies complex operations and enhances the speedup ratio of the algorithm. All these shows that the MR-FIMNA algorithm is suitable for large datasets and has strong scalability.

Feasibility Analysis of GM-GSK. To show the importance of the proposed GM-GSK strategy, we conduct comparative experiments on the above 4 datasets with the minimum support thresholds of 1000 and 10000. Both the GM-GSK strategy and default strategy (PrePost algorithm) are employed for evaluating its impact on the comprehensive performance of the algorithm. Fig. 3 shows the running time of the algorithm under the two different strategies. We can notice that the time consuming of the GM-GSK strategy is less than default strategy. In particular, when the GM-GSK strategy deals with large-scale datasets like HIGGS and Susy, the running time is greatly reduced. This is due to the dataset is evenly grouped by using GM-GSK strategy. That is to say, The GM-GSK strategy shortens the time required for nodes to traverse the PPC-tree.

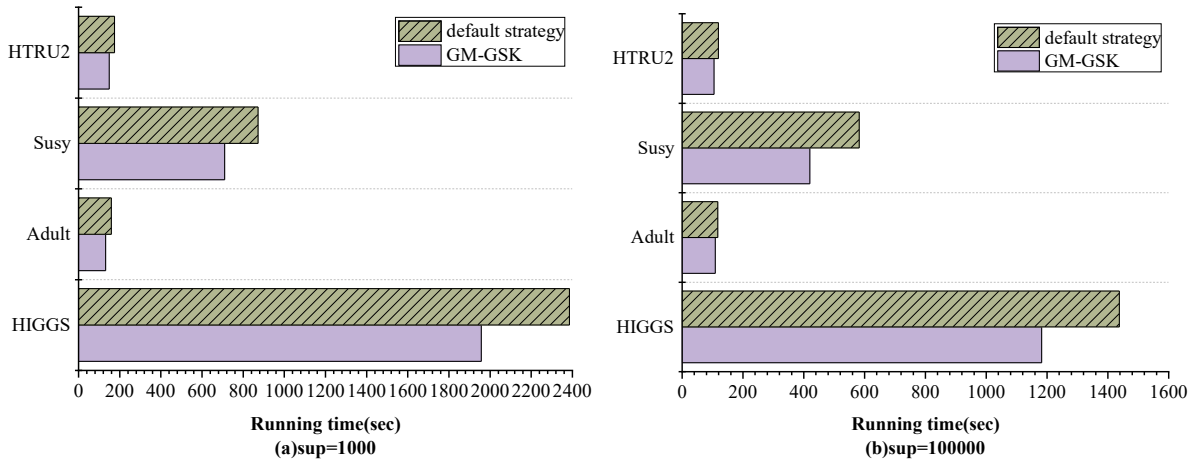
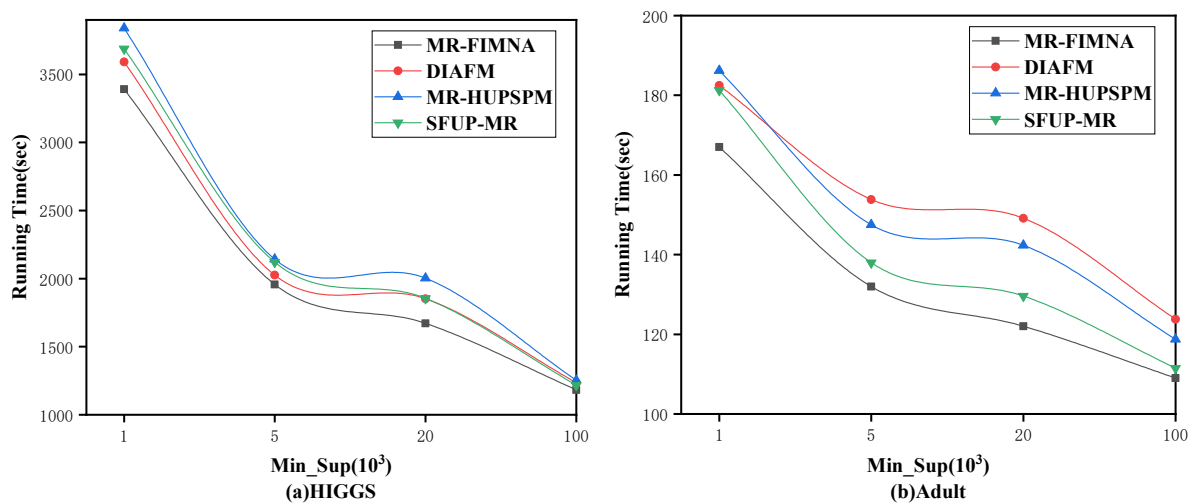


Fig. 3. The comparison of MR-FIMNA algorithm running time with GM-GSK strategy and default strategy

5.4 Performance of MR-FIMNA

To verify the performance of the MR-FIMNA algorithm. We conduct a comparative experiment on the above experimental datasets, and compare the performance of the algorithm among the DIAFM algorithm, the MR-HUPSPM algorithm and the SFUP-MR algorithm according to the running time and memory usage of the algorithm.

Comparison of Running Time. It is important to establish the number of groups according to the F-list scale of every dataset. The size of F-lists for the four datasets with different support degrees is shown in Table 3. We set 50 groups for the Susy, 100 groups for Adult and HTRU2, 1000 groups for the HIGGS. The experimental results are shown in Fig. 4. We can notice that the running time of MR-FIMNA algorithm in each dataset is less than other algorithm. This is due to the following two reasons. Firstly, the MR-FIMNA algorithm uses the GM-GSK strategy to evenly distribute the frequent 1-itemsets to each node. Second, the PAS strategy is proposed to determine whether the current item is a frequent itemset. The PAS strategy simplifies complex operations and reduces the running time of the algorithm.



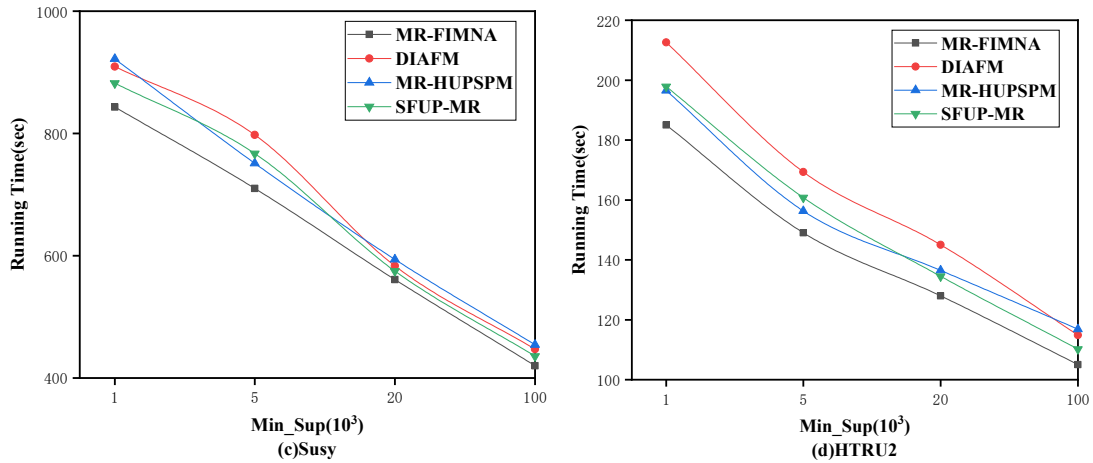
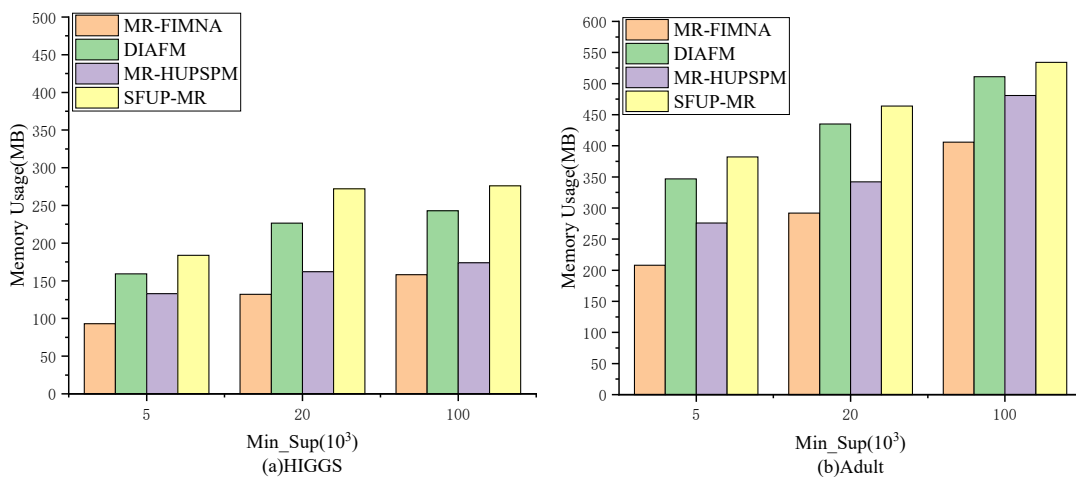


Fig. 4. Comparison of the running time of the four algorithms on different datasets

Table 3. The F-list size of four datasets with different support degrees

Minimum support	1000	10000	20000	100000
HIGGS	17104	6385	2420	556
Adult	1253	156	27	4
Susy	117	112	99	91
HTRU2	207	151	102	32

Comparison of Memory Usage. In this experiment, we choose the MR-FIMNA algorithm, the DIAFM algorithm, the MR-HUPSPM algorithm and the SFUP-MR algorithm to perform the experiments. Generally, in computation of memory usage in each node, average memory is implemented. The related range for minimum support threshold is set between 10000, 20000 and 100000. Fig. 5 shows the memory used with great details. We can emphasize this fact that MR-FIMNA algorithm needs less memory usage in comparison with others, which is an interesting fact in its kind. This is because compared with the DIAFM algorithm, MR-FIMNA algorithm only need to build N-list of frequent 1-itemsets according to the PPC-tree. The PPC-tree can be removed after generating the N-list. Moreover, for MR-HUPSPM algorithm, the subtree of conditional mode will be stored in memory, which will be consuming a lot of memory space. The MR-FIMNA algorithm uses the GM-GSK strategy to evenly group the frequent 1-itemset of F-list. In the stage of mining frequent itemsets in parallel, the algorithm only needs to save the item as frequent itemsets with suffixes. These two reasons reduce the memory usage of the MR-FIMNA algorithm.



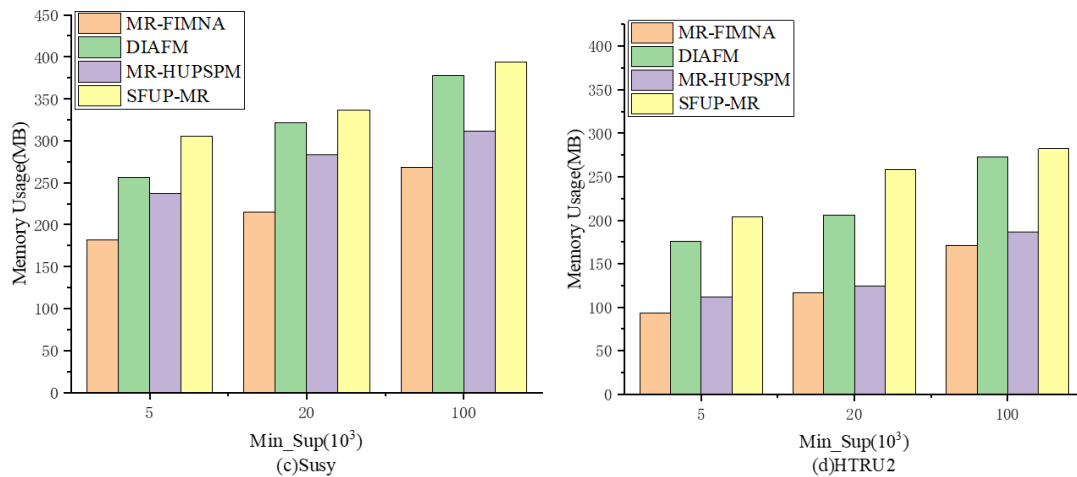


Fig. 5. Comparison of the memory usage of the four algorithms on different dataset

6 Conclusions

In this paper, we proposed the design and implementation of MR-FIMNA, to solve the shortcomings of traditional frequent itemset mining algorithms in a big data environment. we design the *LCEF* to compute the load for individual item in frequent 1-itemset and propose the *GM-GSK* to diminish the limitations brought up by clusters load balance algorithm, and present two strategies named *PAS* to improve the merge efficiency of *N-list* structure and *PSES* to avoid redundant searches during data mining. Experiments are carried out on four large datasets, which involve up to 11.7 million counts of data. Compared to the DIAFM algorithm, the MR-HUPSPM algorithm and the SFUP-MR algorithm, the MR-FIMNA algorithm has shown significant improvements in performance, time consuming as well as memory usage.

Acknowledgement

This research was funded by the Key Improvement Projects of Guangdong Province (Grant Number: 2022ZDJS048), the Shaoguan Science and Technology Plan Projects (Grant Numbers: SZ2022KJ06 and 220607154531533), and the Science and Technology projects of Education Government in Jiangxi province (Grant Numbers: GJJ209406, GJJ218505, and GJJ218504), and the National Nature Sciences Foundation of China with (Grant Number: 42250410321).

References

- [1] L. Mohan, S. Jain, P. Suyal, A. Kumar, Data mining Classification Techniques for Intrusion Detection System, in: Proc. 2020 12th International Conference on Computational Intelligence and Communication Networks (CICN), 2020. <https://doi.org/10.1109/CICN49253.2020.9242642>
- [2] I. Lahmar, A. Zaier, M. Yahia, R. Bouallegue, A New Self Adaptive Fuzzy Unsupervised Clustering Ensemble Based on Spectral Clustering, in: Proc. 2020 17th International Multi-Conference on Systems, Signals & Devices (SSD), 2020. <https://doi.org/10.1109/SSD49366.2020.9364223>
- [3] M. Su, Y. Guo, C. Men, W. Wang, A robust self-weighted SELO regression model, International Journal of Machine Learning and Cybernetics 10(11)(2019) 3189-3199. <https://doi.org/10.1007/s13042-019-01009-1>
- [4] R. Iranzad, X. Liu, A review of random forest-based feature selection methods for data science education and applications, International Journal of Data Science and Analytics 20(2)(2025) 197-211. <https://doi.org/10.1007/s41060-024-00509-w>

- [5] W. Wang, Q. Li, F. Zhu, Association rules combined fuzzy decision quality control technology in intelligent manufacturing, *Intelligent Systems with Applications* 21(2024) 200331.
<https://doi.org/10.1016/j.iswa.2024.200331>
- [6] H. Guo, H. Liu, J.Y. Chen, Y. Zeng, Data Mining and Risk Prediction Based on Apriori Improved Algorithm for Lung Cancer, *Journal of Signal Processing Systems for Signal Image and Video Technology* 93(7)(2021) 795–809.
<https://doi.org/10.1007/s11265-021-01663-1>
- [7] J.M. Smith, An Efficient Parallel FP-Growth Algorithm for Big Data Association Rule Mining, *Journal of Computer Science and Software Applications* 4(1)(2024) 1-8.
<https://mfacademia.org/index.php/jcssa/article/view/57>
- [8] C.K. Zhang, P.B. Tian, X.D. Zhang, Q. Liao, Z.L. Jiang, X. Wang, HashEclat: an efficient frequent itemset algorithm 10(11)(2019) 3003–3016.
<https://doi.org/10.1007/s13042-018-00918-x>
- [9] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, *Communications of the Acm* 51(1) (2008) 107–113.
<https://doi.org/10.1145/1327452.1327492>
- [10] A. Vasoya, N. Koli, Mining of Association Rules on Large Database Using Distributed and Parallel Computing, *Procedia Computer Science* 79(2016) 221-230.
<https://doi.org/10.1016/j.procs.2016.03.029>
- [11] M. Shaikh, S. Akram, J. Khan, S. Khalid, Y. Lee, DIAFM: An Improved and Novel Approach for Incremental Frequent Itemset Mining, *Mathematics* 12(2024) 3930.
<https://doi.org/10.3390/math12243930>
- [12] J.M.T. Wu, S. Liu, J.C.W. Lin, Efficient Uncertain Sequence Pattern Mining Based on Hadoop Platform, *Journal of circuits, systems and computers* 31(15)(2022) 2250261.
<https://doi.org/10.1142/S0218126622502619>
- [13] J.M.T. Wu, R. Li, M.E. Wu, J.C.W. Lin, Mining skyline frequent-utility patterns from big data environment based on MapReduce framework, *Intelligent data analysis* 27(5)(2023) 1359-1377.
<https://doi.org/10.3233/IDA-220756>
- [14] Z.H. Deng, Z.H. Wang, J.J. Jiang, A new algorithm for fast mining frequent itemsets using N-lists, *Science China-Information Sciences* 55(9)(2012) 2008–2030.
<https://doi.org/10.1007/s11432-012-4638-z>
- [15] H. Bui, B. Vo, T.A. Nguyen-Hoang, U. Yun, Mining frequent weighted closed itemsets using the WN-list structure and an early pruning strategy, *Applied Intelligence* 51(3)(2021) 1439–1459.
<https://doi.org/10.1007/s10489-020-01899-7>
- [16] Z.H. Deng, An efficient structure for fast mining high utility itemsets, *Applied Intelligence* 48(9)(2018) 3161–3177.
<https://doi.org/10.1007/s10489-017-1130-x>
- [17] W. Wu, M. Xian, U. Paramalli, B. Lu, Efficient privacy-preserving frequent itemset query over semantically secure encrypted cloud database, *World Wide Web-Internet and Web Information Systems* 24(2)(2021) 607–629.
<https://doi.org/10.1007/s11280-021-00863-w>
- [18] S. Raj, D. Ramesh, M. Sreenu, K. Kumar, EAFIM: efficient apriori-based frequent itemset mining algorithm on Spark for big transactional data, *Knowledge and Information Systems* 62(9)(2020) 3565–3583.
<https://doi.org/10.1007/s10115-020-01464-1>
- [19] V.S. Verykios, E.C. Stavropoulos, P. Krasidakis, E. Sakkopoulos, Frequent itemset hiding revisited: pushing hiding constraints into mining, *Applied Intelligence* 52(3)(2022) 2539-2555.
<https://doi.org/10.1007/s10489-021-02490-4>
- [20] S.H. Cai, R.Z. Sun, S.B. Hao, S.C. Li, G. Yuan, Minimal weighted infrequent itemset mining-based outlier detection approach on uncertain data stream, *Neural Computing & Applications* 32(11)(2020) 6619–6639.
<https://doi.org/10.1007/s00521-018-3876-4>
- [21] Y. Ning, J. Dong, J. Lin, F. Zheng, Y. Fu, Z. Dong, F. Xiao, GRASP: Accelerating Hash-Based PQC Performance on GPU Parallel Architecture, *Cryptology ePrint Archive* (2024) 2024/1030.
<https://ia.cr/2024/1030>