Knowledge-Based Systems 276 (2023) 110733

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Parallel approaches to extract multi-level high utility itemsets from hierarchical transaction databases

Trinh D.D. Nguyen [a], N.T. Tung [b], Thiet Pham [a], Loan T.T. Nguyen [c,d,*]

[a] *Faculty of Information Technology, Industrial University of Ho Chi Minh City, Ho Chi Minh City, Viet Nam*
[b] *Faculty of Information Technology, HUTECH University, Ho Chi Minh City, Viet Nam*
[c] *School of Computer Science and Engineering, International University, Ho Chi Minh City, Viet Nam*
[d] *Vietnam National University, Ho Chi Minh City, Viet Nam*

A R T I C L E I N F O

Article history:
Received 1 February 2023
Received in revised form 13 June 2023
Accepted 14 June 2023
Available online 19 June 2023

Keywords:
High utility itemset
Multi-level
Hierarchical database
Data mining
Load balancing
Parallelism
Multi-core
Shortest-jobs first

A B S T R A C T

In the field of data mining, high utility itemset mining (HUIM) is a relevant mining task, with the aim of analyzing customer transaction databases. HUIM consists of exploiting the set of items that are often purchased together and yield high profit value. In real-world applications, transaction databases often come with item categorization, stored in a taxonomy. Items in these databases can be clustered into specific categories at higher levels of abstraction. Extracting and analyzing itemsets discovered from different levels of abstraction can provide more useful insights into customer behaviors. However, considering item taxonomy increases the problem's complexity, hence prolonging the execution time needed to explore the search space. Parallelism is thus employed to address this drawback, but previous approaches are not efficient as they only adopt simple scheduling strategies or do not utilize the full capabilities of a multi-core processor. This work introduces three new efficient strategies to significantly boost the performance of the multi-level high utility itemset mining task using multi-core processing. Two new algorithms, called MCML+ and MCML++, are also proposed by adopting the suggested strategies. Extensive experiments on several large databases show that our proposed algorithms have better performance compared to previous approaches in terms of running time and scalability, up to 4.0 times better than the previous parallelized algorithm, the MCML-Miner algorithm; and over 9.0 times faster than the original sequential algorithm, the MLHUI-Miner algorithm.

1. Introduction

Since Agrawal first introduced the concept of Frequent Itemset Mining (FIM – abbreviations are shown in Table 1) in 1993 [1], it has remained an important data mining topic [2–4]. FIM is an essential task with many several real-world applications. The mining task requires a threshold known as minimum support, denoted up *minsup*, and a dataset comprised of transactions, or transaction databases. It returns all itemsets whose support, or occurrence count, satisfies the specified minimum support (*minsup*) threshold. However, FIM ignores the importance, or the number of occurrences, of an item appearing in a transaction, and this is not practical in many contexts. For instance, when a customer makes a transaction at a retail store, this transaction often contains details such as the quantities of items purchased and their retail prices, and when applied to these types of databases FIM approaches will discard this information, and their results often contain high occurrence itemsets but yield low profits, while ruling out many low occurrences with high profits.

Based on this, the task of High Utility Itemset Mining (HUIM) was introduced [5] and has since been studied extensively [6–11]. HUIM addresses the limitations of FIM by considering the important information per item, its unit price and purchased quantity when appearing in transactions. Itemsets whose profit (utility) is no less than a user-specified threshold called minimum utility, denoted as *minutil*, are called High Utility Itemsets. Several real-world applications, such as biomedical, click-stream analysis, cross-marketing, etc. adopt HUIM as their essential core. Compared to FIM, HUIM is a more challenging task [5]. FIM relies on a well-known and important property known as the Downward Closure Property (DCP) to eliminate a tremendous number of unpromising itemsets from its problem space, reducing mining time and memory usage. However, the utility measure used in HUIM does not satisfy the DCP, as it is neither monotonic nor anti-monotonic. As such, the effective pruning strategies which were previously employed in FIM cannot be adopted in HUIM.

* Corresponding author at: School of Computer Science and Engineering, International University, Ho Chi Minh City, Viet Nam.
E-mail addresses: 20126291.trinh@student.iuh.edu.vn (T.D.D. Nguyen), nt.tung@hutech.edu.vn (N.T. Tung), phamthithiet@iuh.edu.vn (T. Pham), nttloan@hcmiu.edu.vn (L.T.T. Nguyen).

https://doi.org/10.1016/j.knosys.2023.110733
0950-7051/© 2023 Elsevier B.V. All rights reserved.

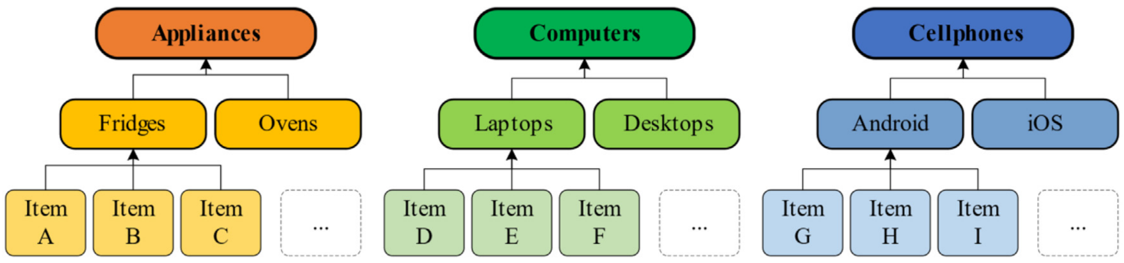

Fig. 1. An example of items categorization of business products from a retail store.

Table 1
Abbreviations used.

Symbol	Description
CLHUI	Cross-Level High Utility Itemset
DCP	Downward Closure Property
DFS	Depth-First Search
EUCS	Estimated Utility Co-occurrence Structure
FIM	Frequent Itemset Mining
GWU	Generalized transaction Weighted Utilization
HUI	High Utility Itemset
HUIM	High Utility Itemset Mining
minutil	Minimum utility threshold
MLHUI	Multi-Level High Utility Itemset
TWU	Transaction Weighted Utilization

Furthermore, traditional HUIM approaches ignore the semantic relationships of items, known as item categorization. Real-world retail stores or e-commerce sites often organize their business products under a hierarchical structure, some can be viewable under a navigational menu on their marketing websites. Business products are organized into related groups. In addition, these groups can be further classified into categories at higher level of abstraction if needed. For example, a retail store that has numerous of products on-shelf such as *"Desktop computers"*, *"Laptops"* of several brands; *"Cellphones"* from various service providers with different type networks; *"Home appliances"* and *"Electronics"* for different household tasks and usages, etc. These products are organized in categories as presented in Fig. 1. This structure is often referred as a taxonomy of items. Traditional HUIM algorithms can only reveal the itemsets found at the lowest abstraction level of the structure, as they do not have the information about item categorizations. The obtained knowledge in this case is only focus on the combination of items that yield high profits. When the database is extended with a taxonomy, each abstraction level in this taxonomy has its own set of items and meaning. The extracted itemsets at each level are called multi-level HUIs, as they contain items from that level only, not spreading to other levels. This knowledge provides more insights for decision makers to devise productive and important business plans, such as boosting promotion for *"Laptops"*, clearance of *"Cellphones"* from a specific operating system, or reducing *"Home appliances"* imports. However, by extending the traditional transaction databases with this type of information significantly enlarge the problem search space, lengthen the mining time. Many FIM approaches were introduced to solve this problem effectively [12–14]. However, these approaches still rely on the DCP to optimize the mining time and memory utilization and are not applicable for HUIM. Although this new mining task can bring forth useful information, only a few approaches have been proposed in HUIM to work on hierarchical transaction databases, until recently [15–18]. This can be addressed to its higher complexity, larger search space and more computationally expensive.

1.1. Motivation

Multi-core processing is a technique that helps utilize the full power of modern multi-core processors, which are available widely at a reasonable cost [19,20]. Many multi-core processors now come with at least two, four, eight, or more physical cores. The multi-core processing technique is often used by several applications to improve their performance, such as raytracing, image analysis, web browsers, web servers, etc. However, the application of this technique to the task of HUIM is still limited [21,22]. Our work thus aims to propose a framework that can be later adopted into other HUIM algorithms to boost their efficiency. Previous works on this topic [21,22] have the following drawbacks:

(a) They do not fully utilize the power of a multi-core processors [21]. Even when allocated with more processing cores, the approaches only make use of a few of these. Furthermore, parallelism is only implemented during the mining phase.
(b) They adopt a simple scheduling strategy known as First Come, First Served when parallelly exploring the search space [22]. Since this strategy is simple, it is not very efficient as the average waiting time is high between tasks. Short tasks at the back of the processing queue must wait for long tasks from the front of the queue to finish.

Therefore, these drawbacks must be addressed to enhance the mining performance, and parallelism should be applied to other phases in the algorithm, if possible, not only the mining phase. In addition, the level-based parallel mining process should also be modified to make it level independent to put all the available processing cores into use. Furthermore, the simple scheduling strategy used has been shown to be inefficient as it may cause a bottleneck when handling longer tasks before shorter ones. A better scheduling strategy is thus needed to lower the waiting time.

1.2. Main contributions

In this work we introduce several effective parallelism strategies to significantly improve the MLHUI performance and so address the given major drawbacks from the previous approaches. The major contributions of our work can be summarized as follows.

(a) Parallelism will be applied to several parts of the algorithm, not just at the search space exploration step, such as single item pruning from transactions using TWU and EUCS construction, thus improving overall mining performance.
(b) Transforming the level-based mining model into a level independent mining process for better CPU utilization, while retaining the taxonomy-driven mining information.

(c) Adopting a more efficient scheduling strategy called Shortest Jobs First to explore the search spaces in parallel, with tasks that have a shorter processing time given higher priority, significantly lowers the average waiting time.
(d) Extending the original work, the MCML-Miner algorithm, in [21] by employing the proposed strategies to propose two new algorithms, namely MCML+ and MCML++.
(e) Thoroughly evaluating the proposed algorithms on several databases, using different number of processor cores to assess their performance in terms of execution time, memory usage, and scalability.

The rest of the text is organized as follows. Recent works related to HUIM and parallel techniques used in HUIM will be discussed in the next section. The problem statement and preliminaries are given in Section 3. Section 4 introduces several strategies to increase the efficiency of parallelism and suggests two new algorithms to solve the limitations of previous works. Extensive experimental evaluations are carried out and presented in Section 5. Finally, Section 6 draws conclusions and discusses further studies on this topic.

2. Related work

2.1. High utility pattern mining

The task of mining high utility itemsets was first introduced by Yao et al. [5] as a generalized problem of itemset mining. It has attracted several research groups, focusing on improving the effectiveness of the mining task [6–9,23–25] as well as several variants [26–31]. Some notable methods are Two-Phase [6], TWU-Mining [23], dTWU-Mining [7], HUI-Miner [24], FHM [8], EFIM [25], and iMEFIM [9]. All these approaches take inputs as a quantitative transaction database and a *minutil* threshold. The output of these algorithms is the complete set of high utility patterns. The differences among them are the data structures, the strategies to explore the problem space and the techniques to efficiently eliminate unpromising candidates when mining HUIs. Because the utility measure is not anti-monotonic, HUIM is considered a more challenging task than FIM, although it is an extension of FIM. As such, HUIM approaches cannot use this measure directly to prune unpromising candidates from the problem space. There are two categories of the state-of-the-art HUIM algorithms, namely two- and one-phase methods. Two-Phase [6], TWU-Mining [23] and dTWU-Mining [7] are some notable algorithms that discover HUIs using two separate stages. During the first stage, database scans are carried out to generate the candidates. This is done by over-estimating the utility value of the candidates. The database is then scanned again to determine the candidates' exact utility value to identify HUIs correctly. Two-Phase enhanced the Apriori algorithm to mine HUIs using a breadth-first search strategy. It was the first complete HUIM algorithm and the first to propose the anti-monotonic *TWU* upper-bound [6]. It can be used to safely eliminate unpromising candidates, those having $TWU < minutil$, from the problem space. This upper-bound was later adopted by many HUIM algorithms as the basis of their pruning strategies. The algorithm then executes several further database scans to compute the exact utility value of the generated candidates. Those with utility greater than *minutil* are filtered out and returned to the users. TWU-Mining enhances the IT-Tree [32] structure to become the WIT-Tree and uses this to reduce the search space as well as improve mining time. dTWU-Mining further enhances the previous approach using diffset [33] to reduce memory usage and carry out faster TWU evaluations. Both TWU-Mining and dTWU-Mining utilize TWU to prune unpromising candidates.

However, these algorithms still require a lengthy execution time to discover HUIs. Several approaches were thus introduced to solve the task using in just one phase. The one-phase algorithms directly compute the utility value of the candidates without generating them, and so there is no need for extra memory space to store the candidates. One of the first one-phase algorithms was HUI-Miner [24]. The authors of this introduced the concept of a tighter upper-bound than TWU, called the remaining utility. By utilizing this new upper-bound, HUI-Miner can remove a larger number of unpromising candidates from the search space than with TWU. Besides the new upper-bound, the authors also proposed the concept of a utility list structure to help speed up the utility calculations. The FHM algorithm [8] extended the HUI-Miner algorithm and introduced a new data structure, named EUCS. EUCS can be constructed in just one database scan and is very efficient in candidate pruning. The pruning strategy based on EUCS is called EUCP. Zida et al. proposed the EFIM algorithm [25]. It comes with two novel utility-based upper-bounds, called local- and subtree-utility, to further prune the candidates. The authors designed the algorithm to handle each pattern found in the search space in linear time. Besides, to lower memory consumption and speed up database scans, EFIM also introduces two strategies called high-utility database projection (HDP) and similar transactions merging (HTM). The authors also proposed a technique called Fast Utility Counting (FUC) to quickly calculate itemsets' utility value. The iMEFIM [9] further enhances the EFIM algorithm with the P-set structure. This new structure significantly lowers the cost of the database scans, and the same work also introduces the concept of an item's dynamic utility value in real-world scenarios and proposes a framework to adapt this important characteristic. More recently, an efficient bit-based approach was introduced by Wu et al. to solve the task of HUIM [34]. The algorithm, known as UBP-Miner, utilizes a novel bitwise operation called Bit mErge cOnstruction (BEO) to speed up the construction of utility-lists. In addition, to support this new bitwise operation, a new data structure named Utility Bit Partition (UBP) was also designed. Hence, the mining performance of the algorithm is also improved when compared to those of HUI-Miner and ULB-Miner.

Several variations of the traditional HUI mining task were also proposed to address more challenges and intricacies met in the real-world. Such as mining closed-HUIs [10,31,35,36], maximal HUIs [37–39], incremental databases [40–43], uncertainty HUIs [44–47], top-*k* HUIs [48–50], high-utility association rules [30,51–53], high-utility itemsets with negative utility [54–58], dynamic databases [43,59–62], and many more variations [63–65].

The concept of discovering HUIs from hierarchical databases was brought up by Cagliero et al. in 2017 [15]. As an extension of the generalized frequent itemset mining, this task was previously studied in depth [12,13,66]. Mining generalized high utility itemsets is an important task in real world applications, as the outputs of the task are more meaningful to the users. However, it is also challenging compared to traditional HUI mining, as the problem's search space is now larger. For instance, a retail store has several items in stock, such as *Diet Coke*, *orange juice*, *mineral water*, *bread*, *donuts*, *pizzas*, etc. In real world applications, these items are often categorized into higher abstraction levels, such as "*drinks*" for *Diet Coke*, *orange juice* and *mineral water*, "*snacks*" for *bread*, *donuts,* and *pizzas*. Traditional HUI and FIM mining algorithms only consider the items at the lowest abstraction level, such as *donuts* or *Diet Coke*. Cagliero et al. proposed an approach to address this drawback, called MLHUI-Miner [15], to discover generalized high utility itemsets from hierarchical databases. The authors also proposed the concept of a taxonomy of items in a hierarchical transaction database. The patterns extracted by

MLHUI-Miner are called multi-level high utility itemsets (MLHUI), as they only contain items appearing from the same abstraction level. Recently, this mining task started to attract more studies due to its importance in real world applications. Sivamathi and Vijayarani [67] introduced an algorithm called MUMA to discover high utility itemsets from multi-level databases. The algorithm implements a tree structure called MUMT to store itemsets' utility information. In this algorithm, each item is encoded based on its abstraction level instead of using a taxonomy, and a different *minutil* is used at each level. The algorithm, however, was only evaluated on small and synthetic databases due to the high complexity of the mining task. Tung et al. suggested an approach called MLHMiner, which is a variation of the algorithm MLHUI-Miner. The algorithm uses HMiner as the core of the mining phase instead of FHM [17]. The CLH-Miner algorithm was proposed by Fournier-Viger et al. in 2020 [16]. It discovers cross-level HUIs (CLHUIs) from hierarchical databases. The authors introduced the concept of GWU, a TWU-based upper-bound to prune generalized itemsets. However, the algorithm still requires a very long execution time when applied to large databases. Recently, Nouioua et al. adopted the CLH-Miner into the top-k cross-level HUI itemset mining task, and the approach is named TKC [18]. Nguyen et al. introduced two new algorithms called MINE_FWUIS and FAST_MINE_FWUIS [14] to mine frequent weighted utility itemsets from hierarchical quantitative databases by using the extended dynamic bit vector structure with large integer elements. Tung et al. suggested an approach named FEACP [68] to efficiently mine CLHUIs. FEACP enhances several tighter upper bounds such as local utility and subtree utility to work with taxonomy and prune a significant number of unpromising candidates. It is thus the best algorithm to date to mine CLHUIs from hierarchical databases.

2.2. Parallel processing for high utility pattern mining

Parallel processing is also adopted by several HUIM approaches besides those on FIM, with some notable works being the following. pEFIM [69] and MCH-Miner [70] extended the EFIM and iMEFIM algorithms, respectively, using multi-core processing to boost the mining performance. Three multi-core algorithms, namely USHPA, USHP and USHR, were introduced by Huynh et al. [71] in 2022. The proposed algorithms aim to efficiently protect the privacy of the input datasets by hiding their sensitive high utility patterns. Extending from the MLHUI-Miner algorithm, Nguyen et al. proposed a multi-core version of the original algorithm, called MCML-Miner [21]. In this approach, each abstraction level in the taxonomy is mined independently using the available processing core, thus lowering the mining time. However, MCML-Miner still does not fully utilize the power of a multi-core processor if the number of available cores is higher than the taxonomy's maximum level. Tung et al. employed the same multi-core processing strategy in the CLH-Miner algorithm, resulting in pCLH-Miner [22]. In this, all available processor cores are exploited to reduce the time needed to discover CLHUIs.

The works mentioned above utilize the multi-core architecture to improve the efficiency of the mining phase. However, as they run on a single computer, the computing resources are limited. As such, these algorithms cannot handle large-scale databases. To address this limitation, several distributed algorithms based on popular frameworks such as MapReduce or Spark have been suggested. Adopting the MapReduce architecture, the PHUI-Miner algorithm [72], proposed by Chen and An, enhances the HUI-Miner algorithm by using a distributed computing framework. Similarly, Sethi et al. introduced a distributed version of the FHM+ [73], namely P-FHM+ [74] to mine HUIs with length constraints. Lin et al. suggested an MapReduce-based algorithm name PHUI-Growth [75] in 2015. PHUI-Growth transforms the input database into a set of reorganized transactions and feeds them to the Mappers. During the Reduce phase, this algorithm relies on several pruning strategies, revised to work on local Reducers. In 2019, a Spark-based parallel algorithm named PKU was proposed by Lin et al. [76]. PKU is designed to extract the top-k HUIs from large-scale databases, utilizing the in-memory architecture provided by the Apache Spark framework. In 2021, Wu et al. proposed an efficient algorithm named HFUPM [77]. This Map-Reduce based algorithm consists of several phases and pruning strategies to efficiently discover fuzzy high utility itemsets from large scale datasets. Recently, a distributed version of EFIM was introduced by Cheng et al. The approach is called P-EFIM, and is based on MapReduce architecture [78]. P-EFIM partitions the search space, under the perspective of a set enumeration tree, into smaller spaces. Each sub search space is a branch of the set enumeration tree. The sub spaces are then distributed into computer nodes and the mining phase is carried out using the local EFIM algorithm.

Table 2
An example of a transaction database.

TID	Transaction	Internal utility	TU
T_1	a, c, d	1, 1, 1	8
T_2	a, b, c, e	2, 2, 6, 2	26
T_3	a, b, c, d, e, f	1, 2, 1, 6, 1, 5	30
T_4	a, b, e	1, 4, 1	16
T_5	b, c, e	1, 1, 1	6
T_6	a, c, d	3, 3, 3	24
T_7	a, b, c, d, f	1, 1, 1, 2, 3	15
T_8	a, c, d, e, f	1, 2, 2, 1, 1	15

As noted above, the multi-core algorithms are only parallelized at the mining step, which is not very efficient. On the distributed side, these algorithms rely mostly on MapReduce and Spark as the underlying architecture. And on each computing node, a local version of the employed algorithm is invoked to do the mining task on the partitioned data.

With this in mind, in this work we aim to increase the efficiency of the mining task by using multi-core architecture on a single computer before diving deeper into the distributed system. As observed earlier, with the multi-core processing model applied in some of the previous works, such as MCML-Miner and pCLH-Miner, the potential of the multi-core processors is still not fully utilized as the previous approaches only employ the multi-core processing into the mining phase, while doing the other parts of the mining process sequentially. In addition, the approaches rely only on the simple scheduling method, which is not efficient.

3. Preliminaries

Key definitions related to the multi-level high utility itemset mining problem are presented in this section. It also provides the problem statement.

Definition 1 (*Transaction Database [5]*)**.** Let $\mathbb{I}$ be the set of n distinct items, $\mathbb{I} = \{i_1, i_2, \dots, i_n\}$, a transaction T_c ($T_c \subseteq \mathbb{I}$) has a unique transaction identifier c (TID). A transaction database is a multiset of transactions, denoted as $\mathbb{D}$, $\mathbb{D} = \{T_1, T_2, \dots, T_m\}$, whereas m denotes the total transactions in $\mathbb{D}$ and $1 \leq c \leq m$. In each transaction T_q, an item $i_j \in T_q$ is connected to a positive integer (internal utility), denoted as $iu\left(i_j, T_q\right)$. Each item $i_j \in \mathbb{I}$ is also linked with another positive integer called its external utility (or unit profit), denoted as $eu\left(i_j\right)$, and effective through $\mathbb{D}$.

For instance, an example of a transaction database is given in Table 2, and the unit profit of each item is given in Table 3.

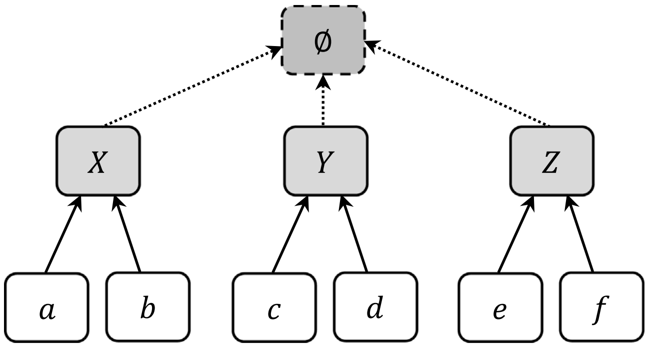

Fig. 2. A sample taxonomy of the database given in Table 2.

Table 3
Unit profit (external utility) of all items in Table 2.

Items	*a*	*b*	*c*	*d*	*e*	*f*
Unit profit	5	2	1	2	3	1

Definition 2 (*Single Item's Utility [5]*)**.** Let $i \in \mathbb{I}$ be an item that appears in transaction $T_q \in \mathbb{D}$, the utility of i in T_q, denoted as $u\left(i, T_q\right)$, is determined as follows.

$$u\left(i, T_q\right) = iu\left(i, T_q\right) \times eu\left(i\right)$$

Definition 3 (*Utility of an Itemset [5]*)**.** Let $\mathcal{X}$ be an itemset of length k that appears in transaction T_q $\left(\mathcal{X} \subseteq \mathbb{I}, T_q \in \mathbb{D}\right)$, the utility of $\mathcal{X}$ in T_q, denoted as $u\left(\mathcal{X}, T_q\right)$, is determined as follows.

$$u\left(\mathcal{X}, T_q\right) = \sum_{i \in \mathcal{X} \wedge \mathcal{X} \subseteq T_q} u\left(i, T_q\right)$$

The utility of $\mathcal{X}$ in the whole database $\mathbb{D}$, $u\left(\mathcal{X}\right)$, is defined as follows [5].

$$u\left(\mathcal{X}\right) = \sum_{T_i \in \mathbb{T}(\mathcal{X})} u\left(\mathcal{X}, T_i\right)$$

In which, the notion $\mathbb{T}\left(\mathcal{X}\right)$ represents the set of all transactions containing the itemset $\mathcal{X}$ in $\mathbb{D}$.

Definition 4 (*Transaction Utility [6]*)**.** The transaction utility of a transaction T_q in database $\mathbb{D}$, denoted as $TU\left(T_q\right)$ is defined as follows.

$$TU\left(T_q\right) = \sum_{i \in T_q} u\left(i, T_q\right)$$

Definition 5 (*Taxonomy [15]*)**.** A taxonomy of items in database $\mathbb{D}$, denoted as τ, is an undirected acyclic graph defined on $\mathbb{D}$. In τ, each external vertex represents one and only one specific item $i \in \mathbb{I}$. An abstract category at a higher abstraction level in τ is represented as an internal vertex. It aggregates all of its child vertices or subcategories [15]. Assuming all items $i \in \mathbb{I}$ can be categorized by τ into only one specific generalized item g. Each generalized item g can be further generalized into another generalized item at a higher abstraction level in the taxonomy τ. A set $\mathbb{G}$ contains all generalized items g in τ. All the child vertices of g using taxonomy τ are denoted as $\Delta\left(g, \tau\right)$, which is a subset of $\mathbb{I}$, $\Delta\left(g, \tau\right) \subseteq \mathbb{I}$.

For the sake of simplicity, hereafter the hierarchical transaction database $\mathbb{D}$ enriched with the taxonomy τ is simply referred to as transaction database $\mathbb{D}$.

For instance, Tables 2 and 3 represent the database $\mathbb{D}$, its taxonomy τ is shown in Fig. 2. In this example, items a, b are categorized into the generalized item X, the same with c, d for the generalized item Y, and e, f for the generalized item Z. All these generalizations are represented as non-dashed edges in Fig. 2. The generalized items X, Y, Z can be later generalized into a higher abstraction level if needed, represented as dashed edges in Fig. 2. Please note that Fig. 2 represent a two-level taxonomy for the database $\mathbb{D}$ given in Tables 2 and 3.

The maximum level of taxonomy τ is denoted as ρ. The taxonomy τ shown in Fig. 2 has $\rho = 2$.

Definition 6 (*Generalized Item's Level [15]*)**.** In taxonomy τ, let $g \in \mathbb{G}$ be a generalized item, $l\left(g, \tau\right)$ denotes its level in τ, is defined as the length of the shortest path between g and any of its external vertex in τ.

Property 1 (*Level oF Specialized Items*)**.** *All specialized items $i \in \mathbb{I}$, which are located at the external vertices, have $l(i, \tau) = 0$.*

Property 2 (*The Maximum Level of a Generalized Item*)**.** *The maximum level of a generalized item $g \in \mathbb{G}$ is corresponding to the height of the taxonomy.*

For instance, the level of specialized item e and f in the taxonomy τ shown in Fig. 2 is $l\left(\{e\}, \tau\right) = level\left(\{f\}, \tau\right) = 0$. The level of generalized item Z in τ is $l\left(\{Z\}, \tau\right) = 1$.

Definition 7 (*Generalized Itemset [15]*)**.** A generalized itemset $\mathcal{X}$ is a set of items $g \in \mathbb{G}$ from the same abstraction level in taxonomy τ. Level of $\mathcal{X}$ is determined by the level of its generalized items.

For instance, using the taxonomy τ in Fig. 2, itemset $\{X, Z\}$ is a generalized itemset since $l\left(\{X\}, \tau\right) = l\left(\{Y\}, \tau\right) = 1$. However, itemset $\{Z, a\}$ does not satisfy the definition of a generalized itemset since $l\left(\{Z\}, \tau\right) \neq l\left(\{a\}, \tau\right)$ in the same taxonomy τ.

Definition 8 (*Generalized Item's Utility [15]*)**.** In database $\mathbb{D}$, using taxonomy τ, let $g \in \mathbb{G}$ be a generalized item. $u\left(g, T_q\right)$ denotes the utility of g in transaction $T_q \in \mathbb{D}$, and is defined as follows.

$$u\left(g, T_q\right) = \sum_{i \in \Delta(g,\tau)} iu\left(i, T_q\right) \times eu\left(i\right)$$

For instance, using the taxonomy τ in Fig. 2, considering the generalized item X which is the parent of two leaf items a and b, the utility of X in T_1 is $u\left(X, T_1\right) = 1 \times 5 + 0 \times 2 = 5$. Similarly, $u\left(X, T_2\right)$ is determined as $u\left(X, T_2\right) = u\left(a, T_2\right) + u\left(b, T_2\right) = 2 \times 5 + 2 \times 2 = 14$.

Definition 9 (*Utility of a Generalized Itemset [15]*)**.** In database $\mathbb{D}$, using taxonomy τ, let $\mathcal{X}$ be a generalized itemset. $u\left(\mathcal{X}, T_q\right)$ denotes the utility of $\mathcal{X}$ in transaction $T_q \in \mathbb{D}$, and is determined as follows.

$$u\left(\mathcal{X}, T_q\right) = \sum_{g \in \mathcal{X}} u\left(g, T_q\right)$$

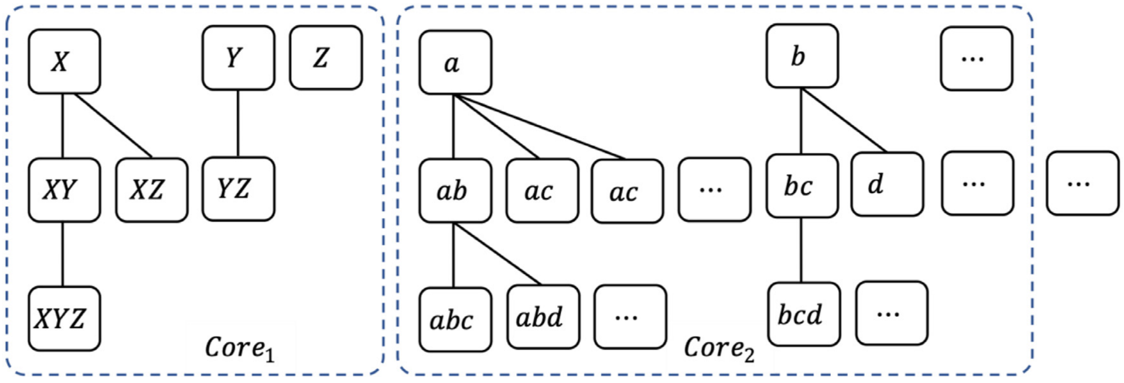

Fig. 3. MCML-Miner search space level-wise parallelism.

For instance, using the taxonomy τ in Fig. 2, considering the generalized itemset $\mathcal{X} = \{X, Y\}$, the utility of $\{X, Y\}$ in T_1 is determined as $u(\{X, Y\}, T_1) = u(X, T_1) + u(Y, T_1) = 5 + 3 = 8$

Similarly, the utility of the generalized itemset $\mathcal{X}$ in the whole database $\mathbb{D}$, denoted as $u(\mathcal{X})$, is determined as follows [15].

$$u(\mathcal{X}) = \sum_{T_q \in \mathbb{D}} u\left(\mathcal{X}, T_q\right)$$

If $\mathcal{X} \notin T_q$, then $u\left(\mathcal{X}, T_q\right) = 0$.

For instance, consider the taxonomy τ in Fig. 2 and the generalized itemset $\mathcal{X} = \{X, Y\}$. The utility of $\mathcal{X}$ in the whole database $\mathbb{D}$ is $u(\mathcal{X}) = u(\mathcal{X}, T_1) + u(\mathcal{X}, T_2) + u(\mathcal{X}, T_3) + u(\mathcal{X}, T_5) + u(\mathcal{X}, T_6) + u(\mathcal{X}, T_7) + u(\mathcal{X}, T_8) = 8 + 20 + 22 + 3 + 24 + 12 + 11 = 100$. Please note that, $\mathcal{X} \notin T_4$, thus $u(\mathcal{X}, T_4) = 0$.

This definition is also applied to compute the utility of a single generalized item in the whole database. For instance, using the taxonomy τ in Fig. 2, the utility of X, Y and Z are 70, 43 and 27, respectively.

Definition 10 (*Multi-level High Utility Itemset [15]*)**.** Given a hierarchical transaction database $\mathbb{D}$ enriched with taxonomy information τ, and a minimum utility threshold *minutil* (which is a user-specified value). A (generalized) itemset $\mathcal{X}$ is called a multi-level high utility itemset if and only if $u(\mathcal{X}) \geq minutil$.

For instance, considering the database given in Table 2, Table 3 and the taxonomy τ shown in Fig. 2, assuming $minutil = 60$, the following itemsets are consider multi-level high utility itemsets: $\{a, d\} : 63$; $\{X, Y\} : 100$ as their utility satisfy the *minutil* threshold.

Definition 11 (*Multi-level High Utility Itemset Mining [15]*)**.** Assuming $\mathbb{D}$ is a hierarchical database enriched with taxonomy information τ, and a minimum utility threshold *minutil*. The goal of the multi-level high utility itemset mining task is to return the complete set of multi-level high utility itemsets found in $\mathbb{D}$.

For instance, considering the database given in Table 2, Table 3 and the taxonomy τ shown in Fig. 2, assuming $minutil = 60$, the following six itemsets are the discovered multi-level high utility itemsets: $\{a, d\} : 63$; $\{a, c, d\} : 71$; $\{X\} : 70$; $\{X, Y\} : 100$; $\{X, Z\} : 77$ and $\{X, Y, Z\} : 92$

4. Proposed algorithms

The MCML-Miner algorithm [21] was proposed by Nguyen et al. in 2020, which is already a parallelized version of the MLHUI-Miner [15]. This algorithm allows concurrent extraction of multi-level HUIs from hierarchical databases. Fig. 3 demonstrates the execution of MCML-Miner on a two-level taxonomy given in Fig. 2. As pointed out in Section 3, the multi-level high utility itemset mining task explores the taxonomy levels independently to extract the MLHUIs per level. The search space of each level is completely separated from the others. And thus to improve the mining performance each level in the taxonomy, which is shown in dashed boxes, is assigned to a separate processor core to be explored simultaneously using a depth-first search (DFS) strategy by MCML-Miner [21]. However, this is also the major drawback of the MCML-Miner algorithm. If the CPU has more available cores than the maximum number of levels of the given taxonomy, then this strategy would be inefficient since it does not fully utilize all available cores. In the worst case, if the taxonomy is empty or contains only one level, then the parallel algorithm MCML-Miner reverts to the sequential version, the MLHUI-Miner. This section therefore proposes our approaches to further boost the performance of the MCML-Miner algorithm.

4.1. Parallel pruning of unpromising candidates

Performing database scans to remove unpromising items and sort the transactions is one of the core operations of the mining task. In this step, all items whose $TWU < minutil$ are considered unpromising will be ruled out from the search space. All of the traditional one-phase HUI mining algorithms, such as HUI-Miner [24], FHM [8], EFIM [25], iMEFIM [9], MLHUI-Miner [15] or even the parallelized version of MLHUI-Miner, the MCML-Miner algorithm [21], still conduct this step sequentially. When performing a database scan at this step most of the HUI mining algorithms will perform the following operations on each transaction:

- Scan through the transaction to remove unpromising items [24,25].
- Sort items in the transaction in the ascending order of their TWU values [24,25].
- Re-evaluate the *TU* value of the transaction.

At each transaction, three operations as mentioned above must be carried out to eliminate the unpromising items, initially reducing the search space prior to exploration. Traditionally, only one transaction can be processed at a time, in a sequential manner. As such, this cannot fully utilize the full potential power of the multi-core processors and leads to a long execution time. This drawback will have a negative impact on overall mining performance, especially on large databases containing millions of transactions. Parallel processing can be applied to this step to reduce the pruning time, as shown in Fig. 4. Assuming $|\mathbb{D}| = m$ transactions, the CPU has ρ processing cores, denoted as $Core_i (1 \leq i \leq \rho)$. If m is evenly divided to ρ, each processor

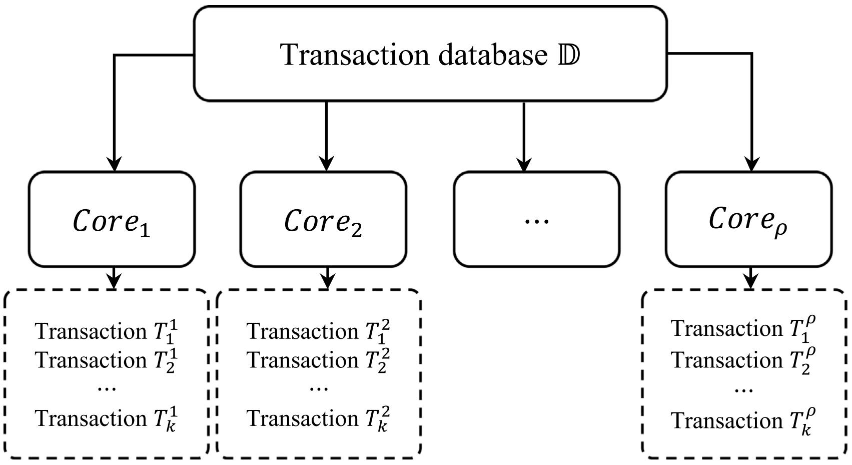

Fig. 4. Parallel transaction processing using multi-core strategy.

core is then distributed exactly $\frac{m}{p}$ transactions to perform the pruning task. Considering processor core ith, it will be assigned by up to $\frac{m}{p}$ transactions, denoted as T_k^i, whereas $1 \leq i \leq \rho$ and k is determined *globally* as $\frac{(i-1)\times m+1}{p} \leq k \leq \frac{i\times m}{p}$, into its own memory space to be processed sequentially. Otherwise, the first $\rho - 1$ processing cores will be assigned up to $k = \left\lceil \frac{m}{\rho} \right\rceil$ transactions. The remaining transactions, if they exist, would be $m \bmod \rho$, and will be assigned to the last processing core ρth. Each processor will now handle the TWU-based single item pruning on the assigned the transactions and the process is carried out simultaneously throughout the cores until all the transactions are completely processed.

Take $Core_1$ in Fig. 4, for instance. It will sequentially process up to k transactions, denoted *locally* as T_j^1 with $(1 \leq j \leq k)$ for the sake of simplicity, $Core_2$ will process k transactions, denoted as T_j^2 with $(1 \leq j \leq k)$, and so on, until all the transactions are assigned and fully processed. This whole processing step of pruning unpromising items occurs in parallel per k transactions in $\mathbb{D}$.

For example, considering a transaction database with $m = 4000$ transactions and a processor with $\rho = 4$ cores. Then, $Core_1$ will be assigned transactions from T_1 to T_{1000}, $Core_2$ will be assigned transactions from T_{1001} to T_{2000} and so on, until all 4000 transactions are fully distributed into all four cores.

After being assigned into a separate processor core to be processed, the *TU* value of each transaction will be re-evaluated as the unpromising items are removed. The transaction's *TU* values are different for each taxonomy level, and they can be determined using the following definitions.

Definition 12 (*List of Transaction's Items at a Taxonomy Level*). Let T_c $(1 \leq c \leq m)$ be a transaction in $\mathbb{D}$, the list of items contained in transaction T_c at level l is a list that stores all descendant items (or generalized items of descendant items in T_c) appearing at the same level l in the taxonomy τ, denoted as $L_l(T_c)$.

Definition 13 (*Transaction Utility at a Level*). The utility of transaction T_c at level l in taxonomy τ, denoted as $TU(T_c/l)$ is defined as follows.

$$TU(T_c/l) = \sum_{x\in L_l(T_c)} u(x, T_c)$$

The *TU* value with respect to level l of transaction T_c will be stored in a one-dimensional array. The size of this array is equal to the maximum level ρ of the taxonomy τ, and is shown in Table 4. These values will be later used in the EUCS construction step.

Table 4
Level-based TU array.

Level	0	1	$\cdots$	ρ
Level-based TU	$TU(T_c/0)$	$TU(T_c/1)$	$\cdots$	$TU(T_c/\rho)$

4.2. Parallel EUCS construction

Parallel processing can also be applied to the construction of the EUCS structure. To achieve this goal, EUCS should be placed at the global scope of the algorithm so that all the processor cores can share this structure in a synchronized way. Each time a transaction T_c is scanned, for each $X, Y \subseteq T_c$, the *TU* value for T_c will be updated into the EUCS as follows.

$$EUCS[X, Y] = EUCS[X, Y] + TU(T_c/level(X))$$

4.3. MCML+ algorithm

Our first approach is to adopt parallel pruning of unpromising candidates and parallel EUCS construction into the MCML-Miner algorithm, resulting in an algorithm named MCML+. MCML+ can utilize all available CPU cores to reduce the time needed to prune unpromising items from all transactions in $\mathbb{D}$.

The MCML+ algorithm now has two separate parallelized stages. The first stage is to perform the proposed approach to prune unpromising items. The second stage is to discover MLHUI concurrently for each level. **Algorithm 1** shows the pseudo-code of MCML+. Please note that in this work the per-level threshold function $\gamma(l)$ [21] is set to 1 for all levels in τ.

The pseudo-code of MCML+ is quite similar to that in the MCML-Miner algorithm [21], which can be explained in detail as follows.

- Line #1 initializes the set of found MLHUIs to empty.
- Lines #2, #3 scans the database $\mathbb{D}$ using taxonomy τ to construct the set of all specialized items $\mathbb{I}$ and the set of all generalized item $\mathbb{G}$, and determine the TWU values for all items $i \in \mathbb{I}$. Please note that to perform these two operations efficiently, they are carried out in a single database scan. However, they are presented as two separate lines in the pseudo-code to help readers to understand what operations are carried out in this step.
- Line #4 uses the information obtained from the previous step and the taxonomy τ to determine the TWU values for all generalized items $g \in \mathbb{G}$.

Algorithm 1. MCML+

```
Input: transaction database $\mathbb{D}$; taxonomy $\tau$; $minutil$ threshold
Output: the set of all discovered MLHUIs $\mathcal{R}$.
 1: Initialize $\mathcal{R}$ as an empty set.
 2: Construct the set $\mathbb{I}$ and $\mathbb{G}$ from $\mathbb{D}$ and taxonomy $\tau$.
 3: Scan $\mathbb{D}$ using $\tau$ to calculate TWU values for $i \in \mathbb{I}$.
 4: Calculate TWU values for generalized item $g \in \mathbb{G}$.
 5: $\mathbb{I}^+ \leftarrow \{i \in \mathbb{I} \mid TWU(i) \geq minutil\}$
 6: $\mathbb{G}^+ \leftarrow \{g \in \mathbb{G} \mid TWU(g) \geq minutil\}$
 7: FOREACH $T_c \in \mathbb{D}$ DO PARALLEL                          // approach 4.1
 8:   Scan $T_c$ to remove $i \notin \mathbb{I}^+ \cup \mathbb{G}^+$.
 9:   Sort $T_c$ in ascending order of TWU.
10:   Re-evaluate $TU$ for $T_c$.
11: END FOR
12: Construct the utility-list for each item in $\mathbb{I}^+ \cup \mathbb{G}^+$.
13: Construct $EUCS$ for each item in $\mathbb{I}^+ \cup \mathbb{G}^+$ in parallel.     // approach 4.2
14: FOREACH level $l$ in $\tau$ DO PARALLEL
15:   $\mathbb{GI}_l^+ \leftarrow$ items in $\mathbb{I}^+ \cup \mathbb{G}^+$ at level $l$.
16:   $\mathcal{R}_l \leftarrow$ FHM_Search ($\emptyset, \mathbb{GI}_l^+, minutil, EUCS$)
17: END FOR
18: RETURN $\mathcal{R} \leftarrow \bigcup_{l \in \tau} \mathcal{R}_l$
```

- Lines #5 and #6 remove all unpromising items from both sets $\mathbb{I}$ and $\mathbb{G}$ to form two new sets $\mathbb{I}^+$ and $\mathbb{G}^+$, which contain only the items that have their $TWU \geq minutil$.
- Lines #7 to #11 perform the parallel TWU-based pruning on all transactions, which is presented in Section 4.1.
- Line #12 performs the utility-list constructions for all items in the set $\mathbb{I}^+ \cup \mathbb{G}^+$.
- Line #13 performs the parallel EUCS construction as described in Section 4.2.
- Starting from Line #14 to Line #17, the parallel level-based search space exploration is carried out, exactly the same as MCML-Miner. In which
    - Line #15 extracts the promising items from the set $\mathbb{I}^+ \cup \mathbb{G}^+$ at level l and store them in the set $\mathbb{IG}_l^+$.
    - Line #16 perform the search space exploration using DFS strategy by calling the *FHM-Search* function using the set $\mathbb{IG}_l^+$, *minutil* and the EUCS structure as input parameters. The MLHUIs extracted at level l are returned and stored in the subset $\mathcal{R}_l$.
- Finally, after all taxonomy level were fully explored, all the subsets $\mathcal{R}_l$ are then aggregated to form the final set $\mathcal{R}$ and this is returned to the user.

The major differences between MCML+ and MCML-Miner are from lines #7 to #11 and line #13 of **Algorithm 1**, as follows.

- Lines #7 to #11 perform the work described in Section 4.1.
- Line #13 performs the work described in Section 4.2.

The *FHM-Search* function is completely identical to the one used in MCML-Miner algorithm [21].

4.4. MCML++ algorithm

The approach proposed in Section 4.3 using the MCML+ algorithm can improve the overall mining performance, especially on large databases. However, after line #13, the algorithm is back to the original MCML-Miner algorithm, which is not efficient, as noted above. To further improve the mining performance, this drawback must be addressed thoroughly. All the available CPU cores must be put into use instead of using only a few cores to concurrently explore the taxonomy levels.

One approach to address the MCML-Miner's drawback is to distribute the sub-search space of every single item to a separate processor core. This approach is the same as applied in previous works, such as pEFIM [69], MCH-Miner [70], etc. The scheduling strategy used in these works is simply First Come, First Served. The advantage of this approach is to relieve the CPU from being bottlenecked at a few cores, while leaving the rest unoccupied like the strategy used in MCML-Miner. However, assigning search spaces randomly to CPU cores regardless of their properties could be inefficient, as the task to explore larger spaces might be executed before smaller ones. Thus, the partitioned search spaces must be processed in a specific order: The search spaces with a smaller number of extensions should be explored first, then the larger spaces. This scheduling strategy is known as Shortest Jobs First [79]. Smaller search spaces require a shorter time to explore, and vice versa.

Strategy 1. Order of processing

Let $\mathcal{A}$ and $\mathcal{B}$ be the two itemsets that need to be explored, the search space of itemset $\mathcal{A}$ is considered smaller than the search space of itemset $\mathcal{B}$ if $|E(\mathcal{A})| < |E(\mathcal{B})|$.

Whereas $E(\mathcal{A})$ and $E(\mathcal{B})$ denotes the set of items that can be used to extend itemset $\mathcal{A}$ and $\mathcal{B}$, respectively.

By using *Strategy 1*, the items whose $TWU \geq minutil$ will be sorted before assigning to CPU cores to be explored. *Strategy 1* ensures smaller search spaces (with less item extensions) have higher priority than larger search spaces. The order of processing based on this strategy can be achieved by using a priority queue.

This strategy is applied to the MCML++ algorithm starting from line #14 to #25 in **Algorithm 2**. It ensures that smaller search spaces will be explored before larger ones, instead of picking a search space to explore randomly.

The proposed MCML++ algorithm adopts all the approaches introduced in this section. The level dependent mining phase, which was used by both MCML-Miner and MCML+, is completely replaced by the proposed scheduling strategy. Below are the features of the MCML++ algorithm.

Algorithm 2. MCML++

```
Input: transaction database $\mathbb{D}$; taxonomy $\tau$; minutil threshold
Output: the set of all discovered MLHUIs $\mathcal{R}$.
 1: Initialize $\mathcal{R}$ as an empty set.
 2: Construct the set $\mathbb{I}$ and $\mathbb{G}$ from $\mathbb{D}$ and taxonomy $\tau$.
 3: Scan $\mathbb{D}$ using $\tau$ to calculate TWU values for $i \in \mathbb{I}$.
 4: Calculate TWU values for generalized item $g \in \mathbb{G}$.
 5: $\mathbb{I}^{+} \leftarrow \{i \in \mathbb{I} \mid TWU(i) \geq minutil\}$
 6: $\mathbb{G}^{+} \leftarrow \{g \in \mathbb{G} \mid TWU(g) \geq minutil\}$
 7: FOREACH $T_c \in \mathbb{D}$ DO PARALLEL                    // approach 4.1
 8:   Scan $T_c$ to remove $i \notin \mathbb{I}^{+} \cup \mathbb{G}^{+}$.
 9:   Sort $T_c$ in ascending order of TWU.
10:   Re-evaluate $TU$ for $T_c$.
11: END FOR
12: Construct the utility-list for each item in $\mathbb{I}^{+} \cup \mathbb{G}^{+}$.
13: Construct $EUCS$ for each item in $\mathbb{I}^{+} \cup \mathbb{G}^{+}$ in parallel.     // approach 4.2
    // --- approach 4.4 ---
14: Insert $i \in \mathbb{I}^{+} \cup \mathbb{G}^{+}$ to a priority queue $Q$ using the ascending order of $TWU$
15: FOREACH $i \in Q$ DO PARALLEL   // level-independent parallelized search
                                   // space exploration
16:   $\mathcal{R}_l \leftarrow \emptyset$
17:   IF $i.utilitylist.iutil > minutil$ THEN $\mathcal{R}_l \leftarrow \mathcal{R}_l \cup \{i\}$
18:   IF $i.utilitylist.iutil + i.utilitylist.rutil > minutil$ THEN
19:       $itemsToExplore \leftarrow \emptyset$
20:       FOREACH $x$ at level $l$ that can extend $i$ and $x \succ i$ DO
21:          IF $\exists(i,x,c) \in EUCS$ such that $c \geq minutil$ THEN
22:                $iy.utilitylist \leftarrow Construct(\emptyset, i, y)$;
23:             $itemsToExplore \leftarrow itemsToExplore \cup \{iy\}$
24:       END
25:   $\mathcal{R}_l \leftarrow \mathcal{R}_l \cup \{$FHM_Search $(i, itemsToExplore, minutil, EUCS)\}$
26: END FOR
27: RETURN $\mathcal{R} \leftarrow \bigcup_{l \in \tau} \mathcal{R}_l$
```

- A one-phase algorithm that operates on taxonomy-based transaction database to discover MLHUIs.
- Adopting multi-core processing strategy to improve mining performance.
  - Parallel pruning items from transaction databases utilizing all available CPU cores (*approach 4.1*)
  - Parallel construction of EUCS structure (*approach 4.2*).
  - Removing the level-wise search space parallel exploration limit of MCML-Miner by “flattening” all the promising single items. Each of these items are then processed by a separate core, simultaneously. Fully harnessing all available CPU cores instead of relying on the number of levels in the taxonomy, and thus dramatically improving the mining performance.
  - Adopting the Shortest Jobs First strategy into the parallelized search space exploration (*approach 4.4*).

The first two approaches were implemented in the MCML+ algorithm, while the pseudo-code of MCML++ is presented in **Algorithm 2**.

The full explanation of the MCML++ algorithm is given as follows. MCML++ takes the input database as parameter $\mathbb{D}$, using taxonomy τ and an input minimum utility threshold, *minutil*. The output of the algorithm is the set $\mathcal{R}$ containing all the MHUIs discovered.

- Line #1, the algorithm clears the output set $\mathcal{R}$.
- Similar to **Algorithm 1**, lines #2, #3 scan the database $\mathbb{D}$ using taxonomy τ to construct the set of all specialized items $\mathbb{I}$ and the set of all generalized item $\mathbb{G}$, and determine the TWU values for all items $i \in \mathbb{I}$. And to perform these two operations efficiently, they are carried out in a single database scan.
- From the obtained TWUs from $\mathbb{I}$, line #4 determines the TWUs for all $g \in \mathbb{G}$.
- The sets $\mathbb{I}^{+}$ and $\mathbb{G}^{+}$ are then constructed in lines #5 and #6 by removing unpromising items, whose $TWU < minutil$, from $\mathbb{I}$ and $\mathbb{G}$.
- Using *approach 4.1*, starting from lines #7 to #11, the algorithm performs single item pruning from all transactions $T_c \in \mathbb{D}$ in parallel, as follows.
  - For each T_c, any item that does not exist in the set $\mathbb{I}^{+} \cup \mathbb{G}^{+}$ will be removed.
  - Sorting T_c using the ascending order of TWU values.
  - Transaction T_c then has its transaction utility value reevaluated.
- Line #12 constructs the utility-list for each item $i \in \mathbb{I}^{+} \cup \mathbb{G}^{+}$.
- Line #13, using the *approach 4.2*, constructs the EUCS structure in parallel, for each item $i \in \mathbb{I}^{+} \cup \mathbb{G}^{+}$.
- Lines #14 to #26 are where the major contribution of our work located. Line #14 pushes all $i \in \mathbb{I}^{+} \cup \mathbb{G}^{+}$ into a priority queue $\mathbb{Q}$ based on the ascending order of their *TWU* to guarantee the order of processing as described in *approach 4.4*.
- Starting from line #15 to line #26, the algorithm utilizes *approach 4.4* by popping item *i* off the priority queue $\mathbb{Q}$. This yields the Shortest Jobs First scheduling strategy. Then, each level in taxonomy τ is explored in parallel, as follows.

  - The output of each level $l \in \tau$ is stored in a set denoted as $\mathcal{R}_l$, which is initialized as an empty set, in line #16.
  - If an itemset has its utility satisfy the *minutil* threshold, it is considered a MLHUI and then stored in the set $\mathcal{R}_l$, which is done in line #17.
  - From lines #18 to #24, if an item has as its sum of remaining utility and utility greater than *minutil*, its extensions are then constructed using EUCS and it will be explored further to find larger MLHUIs. The extension items are then placed into the set *itemsToExplore*.
  - With the all the required information thus obtained, the *FHM-Search* function is then invoked for each itemset i, the set containing all its extensions *itemsToExplore*, *minutil* threshold and the EUCS structure. The returned results are appended into the set $\mathcal{R}_l$.

  The *for* loop from line #15 to line #26 is executed simultaneously for each item $i \in \mathcal{Q}$. This removes the limitation of MCML-Miner as well as MCML+. All available processor cores will be put into use to explore the search space of each item i, based on the processing order presented in *Strategy 1*, which is the Shortest Jobs First scheduling, with the help of the priority queue $\mathcal{Q}$.
- Finally, all the returned sets $\mathcal{R}_l$ of each level $l \in \tau$ are then aggregated into the final set $\mathcal{R}$.

With the MCML++ algorithm we completely remove the per-level parallelism, which is existed in MCML-Miner and MCML+. The MCML++ algorithm now puts all available processor cores into use to increase the mining performance. Table 5 puts into perspective all the contributions of our work by comparing the two proposed approaches with the MCML-Miner algorithm and the original sequential algorithm, the MLHUI-Miner [15]. The first column of Table 5 lists all the algorithms that are compared, namely MLHUI-Miner, MCML-Miner, MCML+ and MCML++. The second, third, fourth and fifth columns show the parallel strategies applied. In each column, the symbol ✕ denotes that the respective strategy is not employed, and the symbol ✓ denotes the strategy is presented in the respective algorithm. *TWU pruning*, *Level independent mining* and *SJF Scheduling* are our proposed strategies in this work, while *Level dependent mining* is the technique used in previous work [21].

For the *TWU pruning* column, it represents the first approach mentioned in Section 4.1, the parallel TWU-based pruning of unpromising items from transactions, which MLHUI-Miner and MCML-Miner are not capable of. This approach is employed in both the MCML+ and MCML++ algorithms.

The *Level dependent mining* column represents the level-based parallel mining strategy. This strategy is, of course, absent in MLHUI-Miner. But it is employed in MCML-Miner and MCML+. However, this strategy is not efficient and was replaced by the one proposed in Strategy 1. Hence, MCML++ does not employ this approach. It can be seen that MCML+ provides another parallelism phase in the pruning of unpromising items from transactions but is still inefficient at the second parallelism phase.

The *Level independent mining* column denotes the approach presented in Section 4.4. As mentioned earlier, it addresses the drawback from MCML-Miner and MCML+ in the second phase by removing the limitation of taxonomy level-based mining. This strategy is adopted into MCML++ as shown in Table 5. MCML++ is thus capable of utilizing all CPU cores to discover MLHUIs in parallel and is level independent.

The last column of Table 5 denotes the Shortest Job First scheduling strategy. It is also shown that only the MCML++ algorithm adopts this strategy.

4.5. Complexity analysis

Adopting parallelism into our proposed algorithms does not reduce the search space of the mining task. It only helps in reducing the overall mining time, and the complexity remains the same as with the sequential versions of the algorithms. Thus, we conducted the complexity analysis in relation to the traditional algorithm.

To stay consistent, we assume that $m = |\mathbb{D}|$, $n = |\mathbb{I}|$, $s = |\mathbb{G}|$, $\rho = |\tau|$ and $\rho \ll m$. We then have the following.

The MCML+ algorithm (Algorithm 1):

- First the database scan is combined with the taxonomy scan to construct sets $\mathbb{I}$ and $\mathbb{G}$ and thus takes $O(m \times \rho)$ time in the worst case.
- The second database scan is used to compute TWU for all items in $\mathbb{I}$, the cost of this operation is approximately $O(m \times n \times \rho)$. Since $\rho \ll m$, this would be approximately $O(m \times n)$.
- To compute the TWU of all items in $\mathbb{G}$, a scan on the taxonomy and the set $\mathbb{I}$ is required. However, this can be mitigated by using a hash map per level. Thus, the complexity of this step is roughly $O(s + \rho)$.
- Line #7 to Line #11 perform the TWU-based pruning on all transactions. At each transaction T_c, the process consists of removing unpromising items from T_c, sorting T_c and reevaluating its transaction utility *TU*. Assuming in the worst case, $|T_c| = m$, then removing unpromising items from T_c and reevaluate its *TU* require $O(m)$ cost. Sorting the items in T_c has the average cost of $O(m \log m)$. Thus, the overall cost for this step is $O\left(m^2 \log m\right)$.
- Line #12 performs the utility-list construction for all single items in the set $\mathbb{I}^+ \cup \mathbb{G}^+$. The complexity of this step is determined as $O(n \times s)$.
- The EUCS construction can be done very quickly using a hash map. Thus, its complexity is also bound by $O(n \times s)$.
- For the DFS phase to explore the search space, there are total of n items for the lowest taxonomy level. Thus, in the worst case there are up to $2^n - 1$ itemsets in the search space of this level. For the remaining higher levels, there would be s generalized items. And thus, in the worst case, the number of itemsets is bound by $2^s - 1$.

Since $\rho \ll m$, thus, the time complexity in the worst-case of the MCML+ algorithm is $O\left(m \times 2^{n+s}\right) \approx O(2^{n+s})$, theoretically.

The MCML++ algorithm (Algorithm 2):

MCML++ share the first 13 lines of pseudo-code with the MCML+ algorithm. Hence, these parts all have the same worst-case time complexity. The differences starting from lines #14 to #26.

- The construction of the priority queue $\mathcal{Q}$ using the items from both set $\mathbb{I}$ and set $\mathbb{G}$ costs $O(n + s)$ time.
- The search space exploration using DFS takes place from line #15 to line #26. In this phase, as we removed the limitation of level dependent mining, then all items in $\mathbb{I}$ and $\mathbb{G}$ are put into $\mathcal{Q}$ for exploration. The number of itemsets to be considered in theory is up to $2^{n+s} - 1$. Thus, in the worst-case, the time complexity of MCML++ is also bound by $O\left(2^{n+s}\right)$.

The next section will evaluate the proposed algorithms MCML+ and MCML++ against the original MCML-Miner algorithm.

Table 5
Comparison of the proposed approaches with the previous works.

Algorithm	Level of parallelism			
	TWU pruning	*Level dependent mining*	*Level independent mining*	*SJF scheduling*
MLHUI-Miner	×	×	×	×
MCML-Miner	×	✓	×	×
MCML+	✓	✓	×	×
MCML++	✓	×	✓	✓

5. Experimental evaluation

Experimental evaluations of the proposed algorithms MCML+ and MCML++ are carried out in this section against the parallel MCML-Miner algorithm and the state-of-the-art, sequential MLHUI-Miner algorithm. The evaluated characteristics are mining time, memory usage, scalability, and the order of processing (or scheduling order).

5.1. Experimental design

All our experiments were carried out using a computer equipped with an Intel® Xeon® E5-2678 V3 64-bit processor, containing a maximum of 12 physical cores. The processor base clock is at 2.5 GHz. The machine has 32 GB of DDR4 ECC memory, running the 64-bit Windows 10 Pro Workstation operating system.

We extended the original MCML-Miner algorithm [21], which was previously implemented in Java, into two versions using the parallelism optimizations as presented in the previous section, namely MCML+ and MCML++, respectively. The Java Development Kit used in the coding work is JDK8.0. To make sure all available memory is put into use without any interference from the Java Garbage Collector during the tests, the heap is reserved by up to 96% of system memory via the JVM option: `-Xmx` and `-Xms`. Thus, the actual amount of memory reserved to the JVM heap is 30.7 GB. To limit the number of processing cores for each test, the following JVM option is also used: `-XX:ActiveProcessorCount=`ρ, in which ρ is the desired number of processing cores.

The original MCML-Miner algorithm was tested using $\rho = 12$ processor cores on all scenarios. However, the actual processor cores utilized by MCML-Miner are dependent on the maximum levels of the taxonomy of the experimental databases [21]. For example, MCML-Miner would make use of nine processor cores when applying on *Chainstore* or *Chainstore4X*, even though it was allocated up to 12 cores. The original MCML-Miner is denoted as MCML(12) for the sake of simplicity in the upcoming sub-sections. The MCML+ and MCML++ are tested using $\rho = 2, 8$ and 12 processor cores. They are denoted respectively as MCML+(2), MCML+(8), MCML+(12) for the MCML+ algorithm, and MCML++(2), MCML++(8), MCML++(12) for the MCML++ algorithm.

To ensure the correctness of the results, all the algorithms were executed ten times. The obtained statistics are then averaged and reported. Standard Java API was used to record the running time and memory usage of each algorithm. In the runtime and memory usage test, we varied the *minutil* threshold on each database and record the returned statistics. For the scalability evaluations, we fixed the *minutil* to the lowest values used in the runtime tests, then varied the number of transactions in each database by a ratio of 25, 50, 75 and 100 percentage of the original database. We also evaluate the performance of the order of processing strategies as proposed in the previous section. They are Shortest Jobs First, Reversed Shortest Jobs First, and without any strategy applied. The statistics returned after each test are also recorded, averaged, and reported.

Table 6
Experimental database characteristics.

Database	$\vert\mathbb{D}\vert$	$\vert\mathbb{I}\vert$	$\vert\mathbb{G}\vert$	$\vert\tau\vert$	T_{MAX}	T_{AVG}
Connect	67,577	129	39	4	43	43.0
Chainstore	1,112,949	46,086	9674	9	170	7.2
Chainstore4X	4,451,796	46,086	9674	9	170	7.2
SUSY	5,000,000	190	63	4	19	19.0

5.2. Experimental databases

Three databases were used in the tests, which are *Connect*, *Chainstore* and *SUSY*. The databases *Connect* and *Chainstore* are provided at the SPMF library [80]. The *SUSY* database was obtained from the UCI Machine Learning Repository[1] and transformed into a transactions-based database. The internal utility information of items in the *Connect* and *SUSY* databases were randomly generated in the range [1, 10] to make them on a par with those used in previous HUIM studies. *Chainstore* and *SUSY* are the two large databases, with over 1.1M and 5.0M transactions, respectively. However, to further assess the performance and scalability of the proposed algorithms, we multiplied the number of transactions in *Chainstore* by a factor of four. The result is another large database called *Chainstore4X*, containing over 4.4M transactions. To the best of our knowledge, *Chainstore* and *SUSY* are the two largest real-world transaction databases used in several works in HUIM. Even previous distributed approaches, such as [75,76,78], rely on only *Chainstore* to carry out their experiments.

Since these databases come without a taxonomy, we synthesized a taxonomy for each of them using a different number of levels. Please note that the databases *Chainstore* and *Chainstore4X* share the same taxonomy.

Table 6 shows the features of the databases used in the evaluations. The features are as follows. The total number of transactions in each database is denoted as $|\mathbb{D}|$; The total number of distinct items is represented as $|\mathbb{I}|$; The number of generalized items in the taxonomy τ is given in $|\mathbb{G}|$; The maximum number of taxonomy levels in each database is denoted as $|\tau|$; The maximum and average transaction length of each database are respectively given in T_{MAX} and T_{AVG}.

5.3. Runtime evaluation

In this test, although the MCML-Miner was tested on all *minutil* thresholds using 12 processor cores, denoted as MCML(12), it only utilized four cores on the *Connect* and *SUSY* databases and nine cores on *Chainstore* and *Chainstore4X* [21]. The results obtained from runtime experimental evaluations are shown in Fig. 5. In this figure, all the dashed lines represent the MCML+ algorithms and solid lines represent MCML++ algorithms. The solid gray line with a vertical marker represents the original MCML algorithm. The MLHUI-Miner is represented as a solid black line with a cross marker.

[1] https://archive.ics.uci.edu/ml/datasets/SUSY.

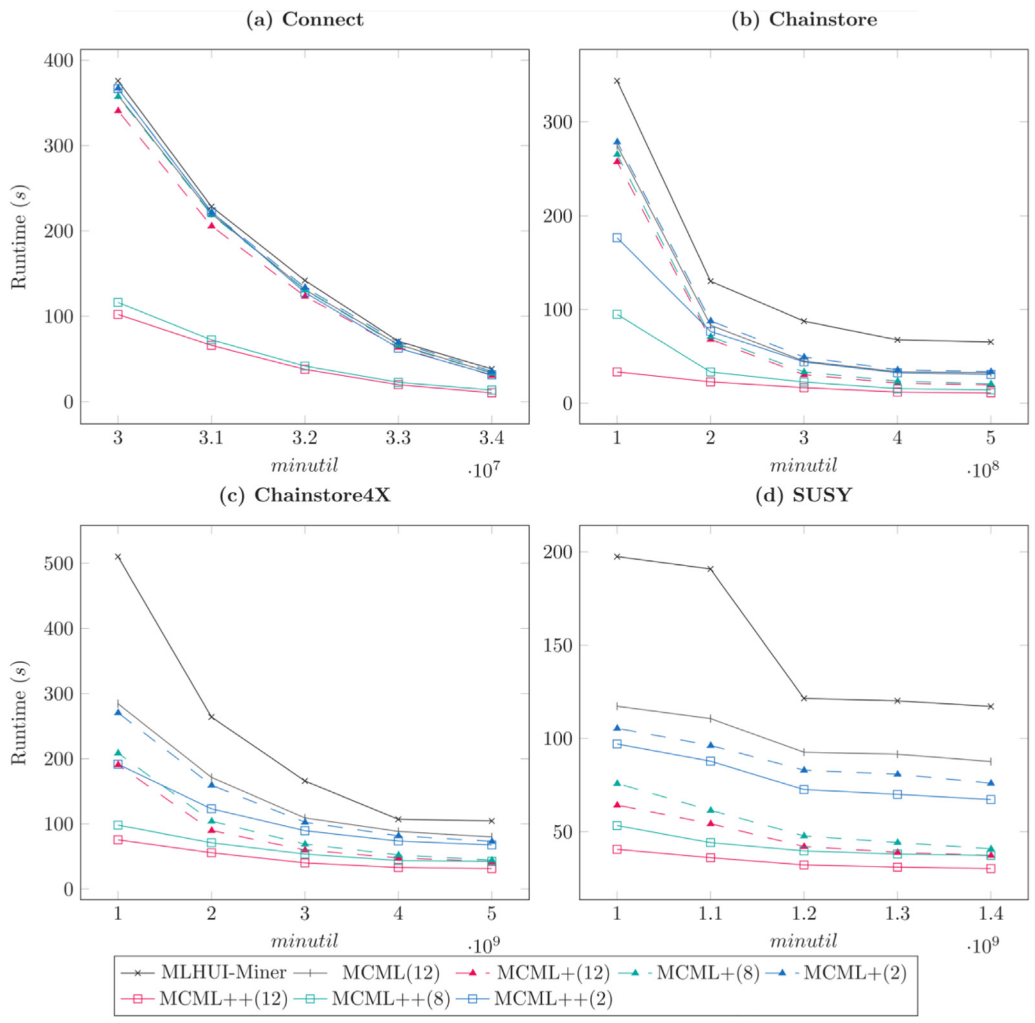

Fig. 5. Runtime comparisons on experimental databases.

It can be seen on all the tested databases that MCML++, which adopts all the proposed approaches, has the best performance compared to MCML+, MCML-Miner and MLHUI-Miner. Besides, throughout all *minutil* thresholds, MCML++ is shown to be far superior in performance than MCML+ when assigned a high number of processor cores to operate. On the *Connect* database, as shown in Fig. 5a, the speed gained is up to 4.0 times faster than that of MCML+, MCML-Miner and MLHUI-Miner. *Connect* is the smallest database of all four, with over 67K transactions. The parallel pruning in the first parallel stage has minimal effect in this case. The remaining runtime is in the second parallel stage, mining using all the assigned cores for the case of MCML++ or level-based mining for the rest of the algorithms. In the case of the *Chainstore* database (Fig. 5b), with 1.1M transactions, the speed gained by MCML++ is over eight times faster than that with MCML using the same number of cores and over ten times compare to MLHUI-Miner. With twelve and eight cores used, MCML++ is up to 8.0× faster than MCML+. By limiting to only two cores, the speed up gained by MCML++ over MCML+ and MCML is the lowest, up to 1.1 times. With *Chainstore4X* (Fig. 5c) and *SUSY* (Fig. 5d), the speed-up gained by MCML++(12) is approximately 4.0 times faster than MCML(12) and 7.0 times faster than MLHUI-Miner. It can be observed that MCML++(12) is up to 2.5 times faster than MCML+(12), the same amount for MCML+ with eight and two cores. Since these are large databases, the approaches adopted in MCML++ are shown to have high effectiveness even when utilizing only two cores when compared to MCML(12).

Throughout the runtime experiments, it can be seen that there are diminishing returns by adding more cores to the algorithm. This depends on the structure of the search space of the algorithm and is represented as a set enumeration tree. The more balanced the tree is, the greater the benefit. Fig. 5 shows that MCML++(12) has less performance gain than MCML++(8). However, the performance gap between MCML++(8) and MCML++(2) is large, as seen with the Connect (Fig. 5a) and Chainstore databases (Fig. 5b). Thus, adding more cores does not always mean the algorithm gains more benefit from the processing power. Besides, the efficiency achieved by parallelizing is high. The performance of the MCML++(12) is up to ten times better than that seen with the sequential MLHUI-Miner algorithm. If we lower the *minutil* threshold more and as long as there is sufficient memory to carry out the algorithm, this performance gap could be even higher.

5.4. Memory usage evaluation

The memory consumption of the tested algorithms is also recorded and averaged during the runtime tests, as shown in Fig. 6. It can be seen from Fig. 6a to d that as we lower the *minutil* threshold the memory usage increases. MCML++ has the highest memory consumption ratio when compared to the MCML+, MCML-Miner and MLHUI-Miner algorithms. The more cores that are allocated to the algorithms, the more memory that

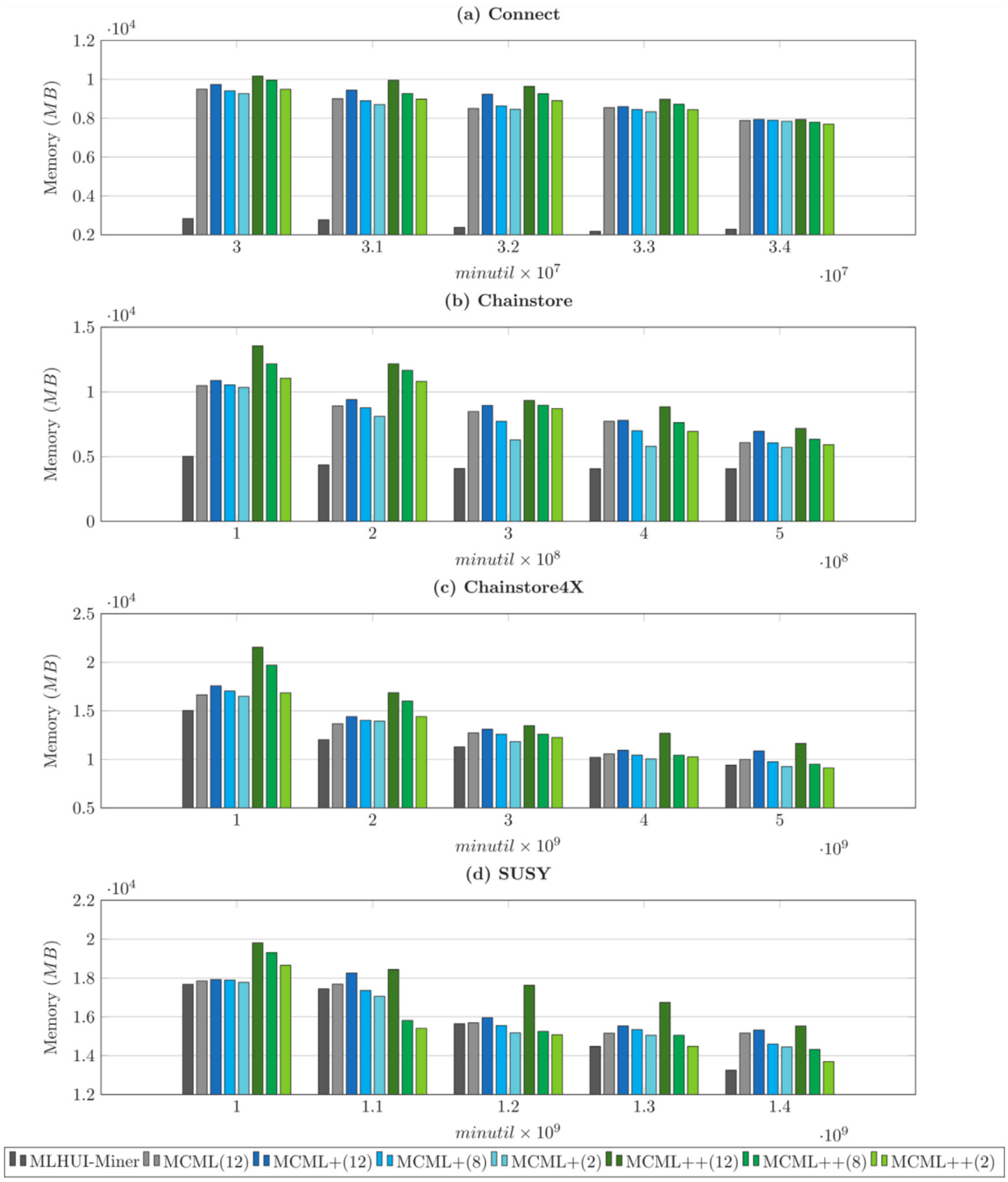

Fig. 6. Comparisons of memory usage on experimental databases.

is required. The MLHUI-Miner has the lowest memory footprint in all the tests.

This can be seen clearly with the *Chainstore* (Fig. 6b) and *Chainstore4X* (Fig. 6c) databases, as these are sparse databases with a large number of items. However, for the dense databases such as *Connect* (Fig. 6a) and *SUSY* (Fig. 6d), the difference is marginal. The maximum memory allocated for MCML++(12), MCML+(12) and MCML(12) at the lowest *minutil* threshold is observed in *SUSY*, with 19.8 GB, 17.9 GB and 17.8 GB, respectively (Fig. 6d).

5.5. Scalability

We also assessed the scalability of the proposed algorithms, MCML++ and MCML+ against MCML-Miner and MLHUI-Miner, on the two largest databases used in this evaluation study, *Chainstore4X* and *SUSY*. The number of transactions in the two databases, which is denoted as $|\mathbb{D}|$ in Fig. 7, is varied from 25% to 100% of the original. MLHUI-Miner, MCML, MCML+ and MCML++ were fixed at the lowest *minutil* threshold used in the runtime experiments: $1.0E + 9$ for both *Chainstore4X* and *SUSY*. We run the algorithms several times on each database and the obtained results are then averaged. The runtime-based scalability results are given in Fig. 7 and the memory-based scalability results are shown in Fig. 8.

For the execution time scalability test (Fig. 7), as we increase the size of the test databases the runtime of the algorithms starts to diverge, and the same results as in the runtime tests are observed here. MCML++(2), MCML++(8) and MCML++(12) dominate the charts. MCML++ out-performed MCML-Miner even with only two processing cores assigned and the runtime scales

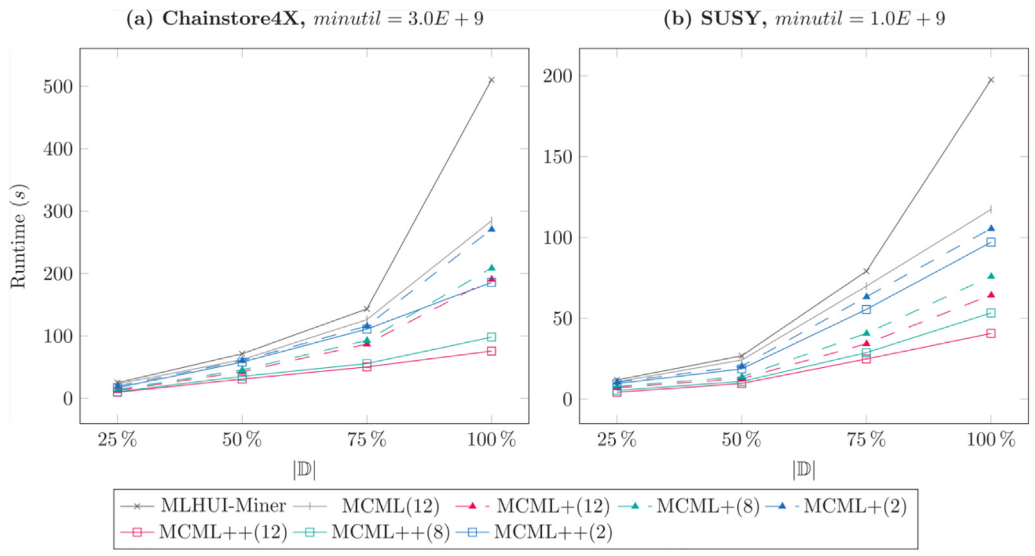

Fig. 7. Experimental on scalability - Execution time.

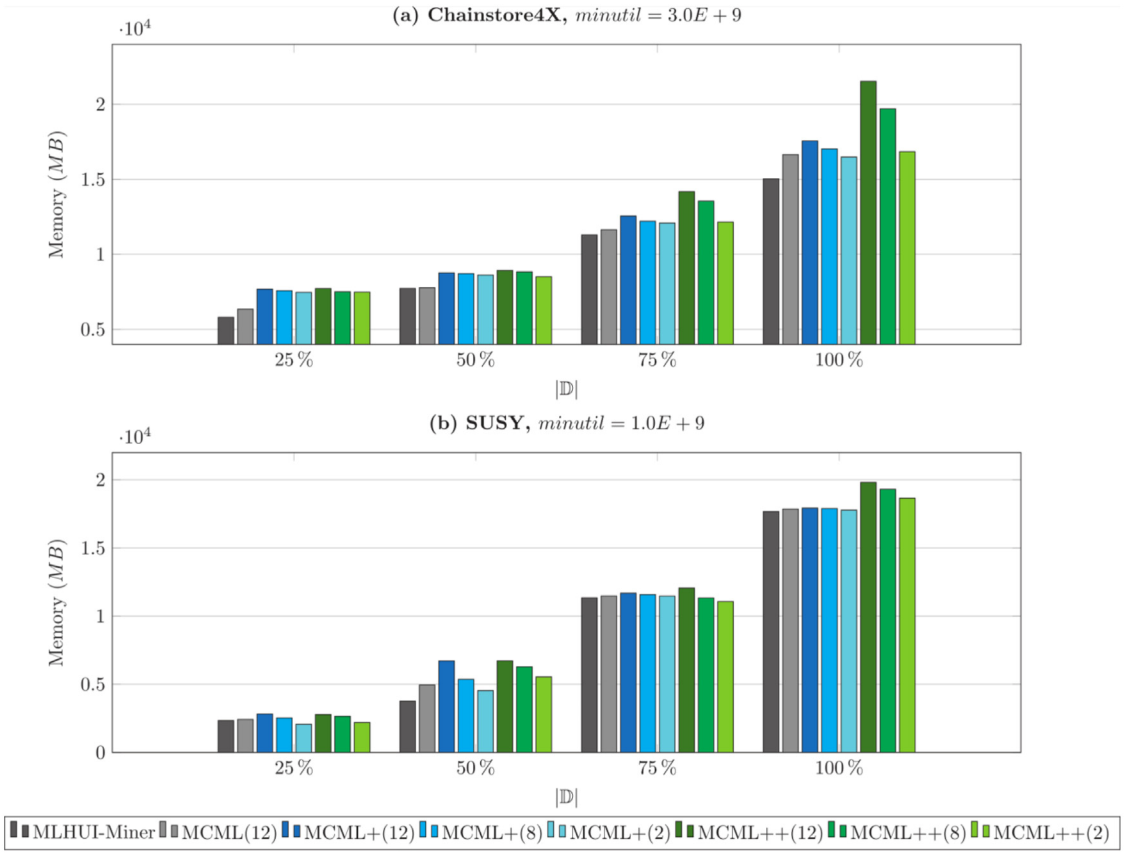

Fig. 8. Experimental on scalability - Memory usage.

almost linearly to the size of the databases, especially with a high number of cores utilized. The scalability of MLHUI-Miner is the lowest in all the algorithms evaluated.

Fig. 8 shows the memory usage of the tested algorithms on the *Chainstore4X* and *SUSY* databases at various database sizes. As the number of transactions in each database increased, the amount of memory allocated to the algorithms also rose. Overall, MCML++ has slightly higher memory usage than MCML+ and MCML-Miner. Moreover, more processor cores assigned means more memory is needed to maintain the stability of the parallelized algorithms. And once again, the MLHUI-Miner algorithm has the lowest memory usage here.

5.6. Order of processing

To evaluate the effectiveness of the proposed scheduling strategy, Shortest Jobs First, we put the MCML++ algorithm to test on all four databases. The number of processor cores is locked at $\rho = 12$. In this test, the order of the priority queue used

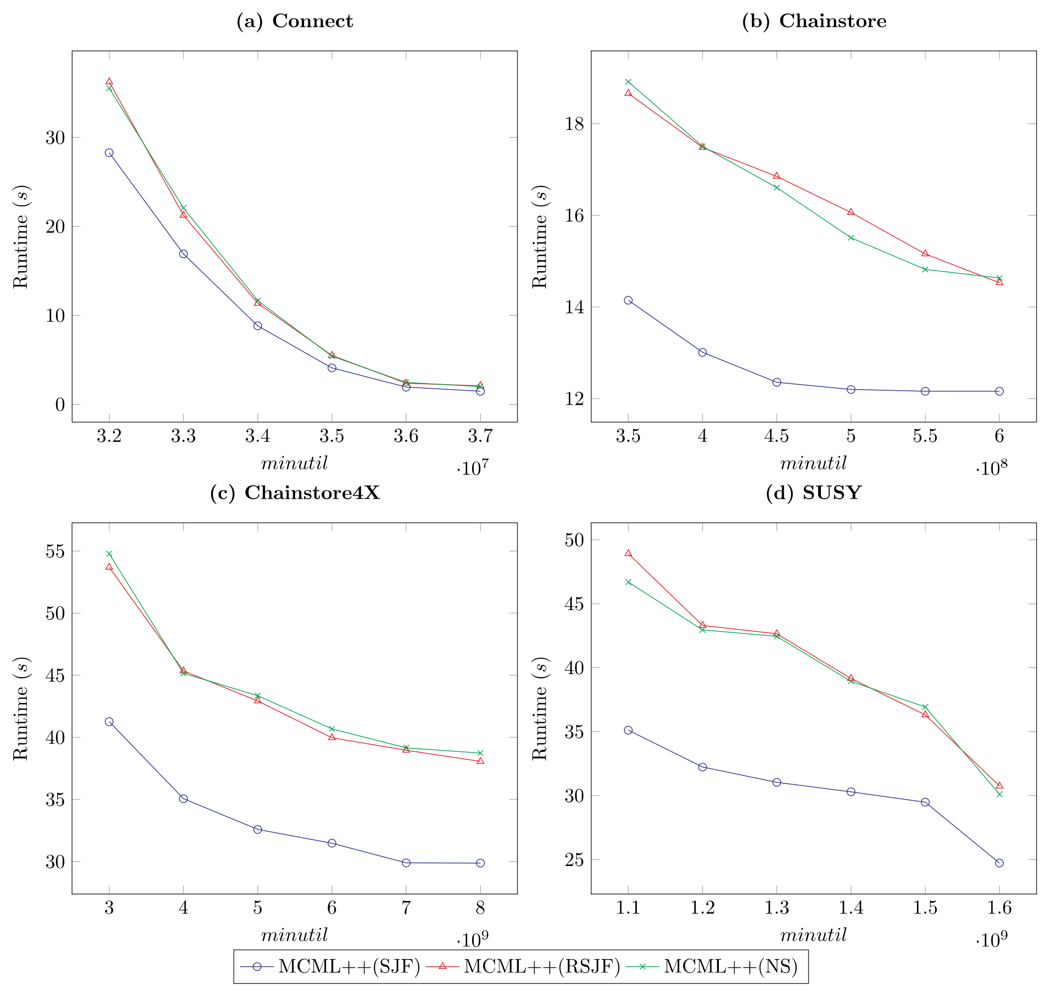

Fig. 9. Order of processing comparisons.

in the scheduling strategy is set to increase the size of search space of the items to be explored (Shortest Jobs First), denoted as MCML++(SJF), and the non-increasing size of search space of the items to be explored (Reversed Shortest Jobs First), denoted as MCML++(RSJF). In addition, we also evaluated the performance of our proposed strategy against the First Come, First Served strategy, whereas the priority queue is reverted to a normal queue to disable the scheduling strategy, denoted as MCML++(NS). The result of the evaluation is shown in Fig. 9.

It can be seen that MCML++(SJF) with the Shortest Jobs First scheduling strategy outperforms MCML++(RSJF) and MCML+(NS) in all the tests. The differences in performance are considerable on large databases such as *Chainstore* (Fig. 9b), *Chainstore4X* (Fig. 9c) and *SUSY* (Fig. 9d). MCML++(RSJF) and MCML++(NS) have almost the same performance. The MCML++(RSJF) uses the same processing order as the set enumeration tree [25], which is left-biased. In this scheduling strategy, items with larger search space will always be the first to be explored, which occupies the processor's time, and the smaller search spaces are left to wait in a queue. For the MCML++(NS), which is simply the First Come, First Served strategy, it depends on the system to assign which item the processor will explore. The order of processing is thus random, and hence the unstable performance when compared to MCML++(RSJF). Overall, MCML++(SJF) has shown its effectiveness when applying the Shortest Jobs First scheduling strategy.

6. Conclusions and future works

In this work, we have proposed three new approaches to boost the multi-level HUI mining performance using multi-core parallelism. Two new algorithms were also introduced, namely MCML+ and MCML++, which are extensions of the MCML-Miner algorithm by adopting the newly proposed approaches. The approaches attempt to put all the available processor cores into use, which leads to better multi-level HUI mining performance. Pruning unpromising items from transactions, construction of the EUCS structure and search space exploration are fully paralleled. Furthermore, the Shortest Jobs First strategy is also employed to efficiently schedule the sub-mining tasks. This thus removes the limitation of the MCML-Miner algorithm, which is level-based parallelism. Extensive experiments were conducted to evaluate the proposed algorithms against the original MLHUI-Miner and parallelized MCML-Miner algorithm on several databases. The results showed that both MCML+ and MCML++ have better performance compared to MCML-Miner and much better than the non-parallel version, the MLHUI-Miner algorithm. On large databases, even with only with two cores assigned, the proposed algorithm MCML++ still outperformed the MCML-Miner, which allocated more processor cores to perform the task. The same results are observed in the scalability tests. However, the memory consumption of the MCML+ and MCML++ algorithms is higher than that of MCML-Miner and MLHUI-Miner. This shows that parallelism combined with effective strategies leads to better mining performance as well as higher processor utilization and improves the performance-to-cost ratio on systems with multi-core processor.

There are some possible improvements that can be done to the algorithms in future work. First, the search space partitioned in all the algorithms is imbalanced. And in most cases, the mining task on each processor core finishes at a different time. Therefore, better ways to partition the search space to split the workload of each task and more balancing are needed. Secondly, the memory usages of both MCML+ and MCML++ are still high. Thus, adopting better search space pruning strategies should be considered. Thirdly, currently all the proposed algorithms only work on a single computer with limited memory and limited computing power. It is possible to extend them further to operate in a distributed environment, such as Hadoop or Spark, to more effectively mine large scale databases.

CRediT authorship contribution statement

Trinh D.D. Nguyen: Writing – original draft, Software, Methodology. **N.T. Tung:** Writing – review & editing, Validation. **Thiet Pham:** Writing – review & editing, Validation, Supervision. **Loan T.T. Nguyen:** Writing – review & editing, Validation, Supervision, Project administration, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request

Acknowledgments

This research is funded by the Vietnam National University of Ho Chi Minh City (VNU-HCM) under grant number B2023-28-02.

References

[1] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, ACM SIGMOD Record 22 (2) (1993) 207–216.
[2] M. Ledmi, S. Zidat, A. Hamdi-Cherif, Grafci+ a fast generator-based algorithm for mining frequent closed itemsets, Knowl. Inf. Syst. 63 (7) (2021) 1873–1908.
[3] S. Raj, D. Ramesh, M. Sreenu, K.K. Sethi, EAFIM: efficient apriori-based frequent itemset mining algorithm on spark for big transactional data, Knowl. Inf. Syst. 62 (9) (2020) 3565–3583.
[4] K.-W. Chon, E. Yi, M.-S. Kim, Sgminer: A fast and scalable GPU-based frequent pattern miner on SSDs, IEEE Access (2022) 1.
[5] H. Yao, H.J. Hamilton, G.J. Butz, A foundational approach to mining itemset utilities from databases, in: SIAM International Conference on Data Mining, Vol. 4, 2004, pp. 482–486.
[6] Y. Liu, W.K. Liao, A. Choudhary, A two-phase algorithm for fast discovery of high utility itemsets, in: 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, Vol. 3518, 2005, pp. 689–695.
[7] B. Le, H. Nguyen, B. Vo, An efficient strategy for mining high utility itemsets, Int. J. Intell. Inf. Database Syst. 5 (2) (2011) 164–176.
[8] P. Fournier-Viger, C.W. Wu, S. Zida, V.S. Tseng, FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning, in: International Symposium on Methodologies for Intelligent Systems, Vol. 8502, 2014, pp. 83–92.
[9] L.T.T. Nguyen, P. Nguyen, T.D.D. Nguyen, B. Vo, P. Fournier-Viger, V.S. Tseng, Mining high-utility itemsets in dynamic profit databases, Knowl.-Based Syst. 175 (2019) 130–144.
[10] P. Fournier-Viger, S. Zida, J.C.W. Lin, C.W. Wu, V.S. Tseng, EFIM-closed: Fast and memory efficient discovery of closed high-utility itemsets, Mach. Learn. Data Min. Pattern Recognit. 9729 (2016) 199–213.
[11] L.T.T. Nguyen, V.V. Vu, M.T.H. Lam, T.T.M. Duong, L.T. Manh, T.T.T. Nguyen, B. Vo, H. Fujita, An efficient method for mining high utility closed itemsets, Inform. Sci. 495 (2019) 78–99.
[12] R. Srikant, R. Agrawal, Mining generalized association rules, Future Gener. Comput. Syst. 13 (2–3) (1997) 161–180.
[13] B. Vo, B. Le, Fast algorithm for mining generalized association rules, Int. J. Database Theor. Appl. 2 (3) (2005) 19–21.
[14] H. Nguyen, T. Le, M. Nguyen, P. Fournier-Viger, V.S. Tseng, B. Vo, Mining frequent weighted utility itemsets in hierarchical quantitative databases, Knowl.-Based Syst. 237 (2022) 107709.
[15] L. Cagliero, S. Chiusano, P. Garza, G. Ricupero, Discovering high-utility itemsets at multiple abstraction levels, in: European Conference on Advances in Databases and Information Systems, Vol. 767, 2017, pp. 224–234.
[16] P. Fournier-Viger, Y. Yang, J.C.-W. Lin, J.M. Luna, S. Ventura, Y. Wang, J.C.-W. Lin, J.M. Luna, S. Ventura, Mining cross-level high utility itemsets, in: 33rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, 2020, p. 12.
[17] N.T. Tung, L.T.T. Nguyen, T.D.D. Nguyen, B. Vo, An efficient method for mining multi-level high utility itemsets, Appl. Intell. 52 (5) (2022) 5475–5496.
[18] M. Nouioua, Y. Wang, P. Fournier-Viger, J.C.-W. Lin, J.M.-T. Wu, TKC: Mining top-k cross-level high utility itemsets, in: 2020 International Conference on Data Mining Workshops, ICDMW, 2020, pp. 673–682.
[19] S. Alias, N.M. Norwawi, pSPADE: Mining sequential pattern using personalized support threshold value, in: International Symposium on Information Technology 2008, Vol. 2, ITSim, 2008, pp. 1–8.
[20] T. Zhu, S. Bai, A parallel mining algorithm for closed sequential patterns, in: 21st International Conference on Advanced Information Networking and Applications Workshops/Symposia, AINAW'07, Vol. 2, 2007, pp. 392–395.
[21] T.D.D. Nguyen, L.T.T. Nguyen, A. Kozierkiewicz, T. Pham, B. Vo, An efficient approach for mining high-utility itemsets from multiple abstraction levels, Intell. Inf. Database Syst. 12672 (2021) 92–103.
[22] N.T. Tung, L.T.T. Nguyen, T.D.D. Nguyen, A. Kozierkiewicz, Cross-level high-utility itemset mining using multi-core processing, in: 13th International Conference on Computational Collective Intelligence, 2021, pp. 467–479.
[23] B. Le, H. Nguyen, T.A. Cao, B. Vo, A novel algorithm for mining high utility itemsets, in: First Asian Conference on Intelligent Information and Database Systems, 2009, pp. 13–17.
[24] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: ACM International Conference on Information and Knowledge Management, CIKM, 2012, pp. 55–64.
[25] S. Zida, P. Fournier-Viger, J.C.W. Lin, C.W. Wu, V.S. Tseng, EFIM: a fast and memory efficient algorithm for high-utility itemset mining, Knowl. Inf. Syst. 51 (2) (2017) 595–625.
[26] W. Gan, S. Wan, J. Chen, C.-M. Chen, L. Qiu, Tophui: top-k high-utility itemset mining with negative utility, in: 2020 IEEE International Conference on Big Data (Big Data), 2020, pp. 5350–5359.
[27] W. Gan, J.C.W. Lin, P. Fournier-Viger, H.C. Chao, P.S. Yu, HUOPM: High-utility occupancy pattern mining, IEEE Trans. Cybern. 50 (3) (2020) 1195–1208.
[28] G. Srivastava, J.C.-W. Lin, M. Pirouz, Y. Li, U. Yun, A pre-large weighted-fusion system of sensed high-utility patterns, IEEE Sens. J. 21 (14) (2021) 15626–15634.
[29] T. Ryu, H. Kim, C. Lee, H. Kim, B. Vo, J.C.-W. Lin, W. Pedrycz, U. Yun, Scalable and efficient approach for high temporal fuzzy utility pattern mining, IEEE Trans. Cybern. (2022) 1–14.
[30] T. Mai, B. Vo, L.T.T. Nguyen, A lattice-based approach for mining high utility association rules, Inform. Sci. 399 (2017) 81–97.
[31] T.D.D. Nguyen, L.T.T. Nguyen, L. Vu, B. Vo, W. Pedrycz, Efficient algorithms for mining closed high utility itemsets in dynamic profit databases, Expert Syst. Appl. 186 (2021) 115741.
[32] M.J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New algorithms for fast discovery of association rules, in: 3rd International Conference on Knowledge Discovery and Data Mining (KDD-97), 1997, pp. 283–286.
[33] M.J. Zaki, K. Gouda, Fast vertical mining using diffsets, in: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003, pp. 326–335.
[34] P. Wu, X. Niu, P. Fournier-Viger, C. Huang, B. Wang, UBP-miner: An efficient bit based high utility itemset mining algorithm, Knowl.-Based Syst. 248 (2022) 108865.
[35] V.S. Tseng, C.W. Wu, P. Fournier-Viger, P.S. Yu, Efficient algorithms for mining the concise and lossless representation of high utility itemsets, IEEE Trans. Knowl. Data Eng. 27 (3) (2015) 726–739.
[36] S. Pramanik, A. Goswami, Discovery of closed high utility itemsets using a fast nature-inspired ant colony algorithm, Appl. Intell. 52 (8) (2022) 8839–8855.
[37] B.E. Shie, P.S. Yu, V.S. Tseng, Efficient algorithms for mining maximal high utility itemsets from data streams with different models, Expert Syst. Appl. 39 (17) (2012) 12947–12960.
[38] L.T.T. Nguyen, D.B. Vu, T.D.D. Nguyen, B. Vo, Mining maximal high utility itemsets on dynamic profit databases, Cybern. Syst. 51 (2) (2020) 140–160.
[39] H. Duong, T. Hoang, T. Tran, T. Truong, B. Le, P. Fournier-Viger, Efficient algorithms for mining closed and maximal high utility itemsets, Knowl.-Based Syst. 257 (2022) 109921.

[40] J. Lee, U. Yun, G. Lee, Analyzing of incremental high utility pattern mining based on tree structures, Hum.-Cent. Comput. Inf. Sci. 7 (1) (2017) 31.
[41] U. Yun, H. Ryang, G. Lee, H. Fujita, An efficient algorithm for mining high utility patterns from incremental databases with one database scan, Knowl.-Based Syst. 124 (2017) 188–206.
[42] J. Lee, U. Yun, G. Lee, E. Yoon, Efficient incremental high utility pattern mining based on pre-large concept, Eng. Appl. Artif. Intell. 72 (2018) 111–123.
[43] U. Yun others, Efficient transaction deleting approach of pre-large based high utility pattern mining in dynamic databases, Future Gener. Comput. Syst. 103 (2020) 58–78.
[44] J.C.-W. Lin, W. Gan, P. Fournier-Viger, T.-P. Hong, V.S. Tseng, Efficiently mining uncertain high-utility itemsets, Soft Comput. 21 (11) (2017) 2801–2820.
[45] B. Vo, L.T.T. Nguyen, N. Bui, T.D.D. Nguyen, V.N. Huynh, T.P. Hong, An efficient method for mining closed potential high-utility itemsets, IEEE Access 8 (2020) 31813–31822.
[46] U. Ahmed, J.C.-W. Lin, G. Srivastava, R. Yasin, Y. Djenouri, An evolutionary model to mine high expected utility patterns from uncertain databases, IEEE Trans. Emerg. Top. Comput. Intell. 5 (1) (2021) 19–28.
[47] W. Gan, J.C.W. Lin, H.C. Chao, A.V. Vasilakos, P.S. Yu, Utility-driven data analytics on uncertain data, IEEE Syst. J. 14 (3) (2020) 4442–4453.
[48] X. Han, X. Liu, J. Li, H. Gao, Efficient top-k high utility itemset mining on massive data, Inform. Sci. 557 (2021) 382–406.
[49] S. Krishnamoorthy, Mining top-k high utility itemsets with effective threshold raising strategies, Expert Syst. Appl. 117 (2019) 148–165.
[50] W. Song, C. Zheng, C. Huang, L. Liu, Heuristically mining the top-k high-utility itemsets with cross-entropy optimization, Appl. Intell. 52 (15) (2022) 17026–17041.
[51] J. Sahoo, A.K. Das, A. Goswami, An efficient approach for mining association rules from high utility itemsets, Expert Syst. Appl. 42 (13) (2015) 5754–5778.
[52] T. Mai, L.T.T. Nguyen, B. Vo, U. Yun, T.P. Hong, Efficient algorithm for mining non-redundant high-utility association rules, Sensors 20 (4) (2020) 1078.
[53] T.D.D. Nguyen, L.T.T. Nguyen, Q. Tran, B. Vo, An efficient algorithm for mining high utility association rules from lattice, J. Comput. Sci. Cybern. 36 (2) (2020) 105–118.
[54] H. Kim, T. Ryu, C. Lee, H. Kim, E. Yoon, B. Vo, J. Chun-Wei Lin, U. Yun, EHMIN: Efficient approach of list based high-utility pattern mining with negative unit profits, Expert Syst. Appl. (2022) 209.
[55] J.C.-W. Lin, P. Fournier-Viger, W. Gan, FHN: An efficient algorithm for mining high-utility itemsets with negative unit profits, Knowl.-Based Syst. 111 (2016) 283–298.
[56] R. Sun, M. Han, C. Zhang, M. Shen, S. Du, Mining of top-k high utility itemsets with negative utility, J. Intell. Fuzzy Systems 40 (2021) 5637–5652.
[57] S. Krishnamoorthy, Efficiently mining high utility itemsets with negative unit profits, Knowl.-Based Syst. (2017) 145.
[58] K. Singh, H.K. Shakya, A. Singh, B. Biswas, Mining of high-utility itemsets with negative utility, Expert Syst. 35 (6) (2018) e12296.
[59] H. Nam, U. Yun, B. Vo, T. Truong, Z.H. Deng, E. Yoon, Efficient approach for damped window-based high utility pattern mining with list structure, IEEE Access 8 (2020) 50958–50968.
[60] H. Kim, U. Yun, Y. Baek, H. Kim, H. Nam, J.C.-W. Lin, P. Fournier-Viger, Damped sliding based utility oriented pattern mining over stream data, Knowl.-Based Syst. 213 (2021) 106653.
[61] C. Lee, Y. Baek, T. Ryu, H. Kim, H. Kim, J. Chun-Wei Lin, B. Vo, U. Yun, An efficient approach for mining maximized erasable utility patterns, Inform. Sci. 609 (2022) 1288–1308.
[62] H. Kim, C. Lee, T. Ryu, H. Kim, S. Kim, B. Vo, J.C.W. Lin, U. Yun, Pre-large based high utility pattern mining for transaction insertions in incremental database, Knowl.-Based Syst. 268 (2023) 110478.
[63] T. Ryu, U. Yun, C. Lee, J.C.-W. Lin, W. Pedrycz, Occupancy-based utility pattern mining in dynamic environments of intelligent systems, Int. J. Intell. Syst. 37 (9) (2022) 5477–5507.
[64] Y. Baek, U. Yun, H. Kim, J. Kim, B. Vo, T. Truong, Z.-H. Deng, Approximate high utility itemset mining in noisy environments, Knowl.-Based Syst. 212 (2021) 106596.
[65] U. Yun, H. Kim, T. Ryu, Y. Baek, H. Nam, J. Lee, B. Vo, W. Pedrycz, Prelarge-based utility-oriented data analytics for transaction modifications in internet of things, IEEE Internet Things J. 8 (2021) 17333–17344.
[66] E. Baralis, L. Cagliero, T. Cerquitelli, P. Garza, Generalized association rule mining with constraints, Inform. Sci. 194 (2012) 68–84.
[67] C. Sivamathi, S. Vijayarani, Multi-level utility mining: Retrieval of high utility itemsets in a transaction database, Comput. Electr. Eng. 76 (2019) 268–282.
[68] N.T. Tung, L.T.T. Nguyen, T.D.D. Nguyen, P. Fourier-Viger, N.T. Nguyen, B. Vo, Efficient mining of cross-level high-utility itemsets in taxonomy quantitative databases, Inform. Sci. 587 (2022) 41–62.
[69] T.D.D. Nguyen, L.T.T. Nguyen, B. Vo, A parallel algorithm for mining high utility itemsets, in: International Conference on Information Systems Architecture and Technology, Vol. 853, 2019, pp. 286–295.
[70] B. Vo, L.T.T. Nguyen, T.D.D. Nguyen, P. Fournier-Viger, U. Yun, A multi-core approach to efficiently mining high-utility itemsets in dynamic profit databases, IEEE Access 8 (2020) 85890–85899.
[71] U. Huynh, B. Le, D.-T. Dinh, H. Fujita, Multi-core parallel algorithms for hiding high-utility sequential patterns, Knowl.-Based Syst. 237 (2022) 107793.
[72] Y. Chen, A. An, Approximate parallel high utility itemset mining, Big Data Res. 6 (2016) 26–42.
[73] P. Fournier-Viger, J.C.W. Lin, Q.H. Duong, T.L. Dam, Fhm+: faster high-utility itemset mining using length upper-bound reduction, Trends Appl. Knowl.-Based Syst. Data Sci. 9799 (2016) 115–127.
[74] K.K. Sethi, D. Ramesh, D.R. Edla, P-fhm+: parallel high utility itemset mining algorithm for big data processing, Procedia Comput. Sci. 132 (2018) 918–927.
[75] Y.C. Lin, C.-W. Wu, V.S. Tseng, Mining high utility itemsets in big data, Adv. Knowl. Discov. Data Min. (2015) 649–661.
[76] C.-H. Lin, C.-W. Wu, J. Huang, V.S. Tseng, Parallel mining of top-k high utility itemsets in spark in-memory computing architecture, Adv. Knowl. Discov. Data Min. (2019) 253–265.
[77] J.M.-T. Wu, G. Srivastava, M. Wei, U. Yun, J.C.-W. Lin, Fuzzy high-utility pattern mining in parallel and distributed hadoop framework, Inform. Sci. 553 (2021) 31–48.
[78] Z. Cheng, W. Shen, W. Fang, C.-W. Lin, A parallel high-utility itemset mining algorithm based on hadoop, Complex Syst. Model. Simul. 3 (2023) 47–58.
[79] W. Stallings, Operating Systems - Internals and Design Principles, 7th ed., Prentice Hall Press, 2011.
[80] P. Fournier-Viger, J.C.W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, H.T. Lam, The SPMF open-source data mining library version 2, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Vol. 9853, 2016, pp. 36–40.