Benjamin Schlegel, Tomas Karnagel, Tim Kiefer, Wolfgang Lehner

Scalable frequent itemset mining on many-core processors

Erstveröffentlichung in / First published in:

SIGMOD/PODS'13: International Conference on Management of Data, New York 24.06.2013. ACM Digital Library, Art. Nr. 3. ISBN 978-1-4503-2196-9

DOI: https://doi.org/10.1145/2485278.2485281

Scalable Frequent Itemset Mining on Many-core Processors

Benjamin Schlegel, Tomas Karnagel, Tim Kiefer, Wolfgang Lehner
Technische Universität Dresden
Dresden, Germany
{firstname,lastname}@tu-dresden.de

ABSTRACT

Frequent-itemset mining is an essential part of the association rule mining process, which has many application areas. It is a computation and memory intensive task with many opportunities for optimization. Many efficient sequential and parallel algorithms were proposed in the recent years. Most of the parallel algorithms, however, cannot cope with the huge number of threads that are provided by large multiprocessor or many-core systems. In this paper, we provide a highly parallel version of the well-known Eclat algorithm. It runs on both, multiprocessor systems and many-core coprocessors, and scales well up to a very large number of threads—244 in our experiments. To evaluate mcEclat's performance, we conducted many experiments on realistic datasets. mcEclat achieves high speedups of up to 11.5x and 100x on a 12-core multiprocessor system and a 61-core Xeon Phi many-core coprocessor, respectively. Furthermore, mcEclat is competitive with highly optimized existing frequent-itemset mining implementations taken from the FIMI repository.

1. INTRODUCTION

Frequent-itemset mining is an essential part of the association rule mining process, which has many application areas like market-basket analysis, gene expression analysis, recommendation, and web-mining. In many of these applications, large datasets need to be mined so there is a need for efficient mining algorithms.

Frequent-itemset mining can be described as follows: Let $\mathcal{I} = \{ a_1, \ldots, a_m \}$ be a set of *items* and $\mathcal{D} = (T_1, \ldots, T_n)$ be a database of *transactions*, where each transaction $T_i \subseteq \mathcal{I}$ consists of a set of items. The *relative support* of an *itemset* $I \subseteq \mathcal{I}$ denotes the percentage of transactions that contain the itemset I. The goal of itemset mining is to find all itemsets that satisfy a certain *minimum relative support* ξ. The chosen ξ value thereby influences the effort for mining; it becomes more expensive as ξ decreases because more frequent itemsets are found.

There exists a large variety of algorithms tackling the challenge of finding frequent itemsets. Basically—as discussed by various authors [4, 7]—none of them is superior over all other algorithms; the dataset being mined and the chosen ξ value determine which algorithm performs best. The most popular frequent-itemset mining algorithms are FP-growth [6] and Eclat [13] for which many optimizations and variants were proposed. Since the sequential performance of processors has stopped increasing in recent years, parallel versions [14, 9] of these algorithms were proposed as well. These algorithms, however, are intended for a rather small number of threads. They thus cannot efficiently run on large multiprocessor systems and many-core coprocessors.

In this paper, we propose mcEclat, which is a highly scalable version of Eclat, for systems that provide a large number of threads. mcEclat is founded on efficient internal data structures and fast set intersections, incorporates an efficient memory management, and provides scalable scheduling techniques for exploring the search space for potential frequent itemsets in parallel. The basic idea of these scheduling techniques is to explore the space in groups of threads instead of using only single threads. This allows to distribute the load more evenly among the available threads and results in a much better scalability.

The contributions of this paper are:

- We review the Eclat algorithm and its state-of-the-art parallel version and discuss why it does not scale to a large number of threads. We further give an overview about the Intel Xeon Phi, which is a relatively new many-core coprocessor.
- We propose mcEclat, which is a highly scalable version of Eclat. It runs well on many-core coprocessors and multiprocessor systems that provide a large number of threads.
- We conduct a large number of experiments on realistic datasets. We show that mcEclat is highly scalable and performs better than existing highly-efficient mining algorithms.

2. PREREQUISITES

In this section, we explain the sequential Eclat algorithm and its state-of-the-art parallel version and provide details about the used many-core coprocessor.

2.1 Eclat

Eclat employs candidate generation and testing to obtain the frequent itemsets of a dataset, i.e., it repeatedly generates candidates based on previously obtained frequent items or itemsets and checks the support of the candidates. To perform the support counting of candidates efficiently, ECLAT transforms the dataset in the *vertical data layout*. In this layout, each frequent item has a *tid-set* assigned that contains the ids of the transactions (tids) in which the item occurs. ECLAT represents tid-sets using lists (tid-lists), which are sorted in ascending order. Intersecting the tid-lists of two items reveals the transactions that contain both items. In general, the support of a k-itemset can be easily determined by intersecting the tid-lists of two arbitrary subsets of length $(k-1)$ and counting the elements in the result tid-list. For example, if the itemsets $\alpha_1 = ab$ and $\alpha_2 = ac$ occur in the transactions $\mathcal{T}(\alpha_1) = \{1,3,4,6\}$ and $\mathcal{T}(\alpha_2) = \{1,2,4,6,7\}$, respectively, then the itemset $\beta = abc$ has a support of 3 since it occurs in the transactions $\mathcal{T}(\beta) = \{1,4,6\}$, which is the intersection result of $\mathcal{T}(\alpha_1) \cap \mathcal{T}(\alpha_2)$.

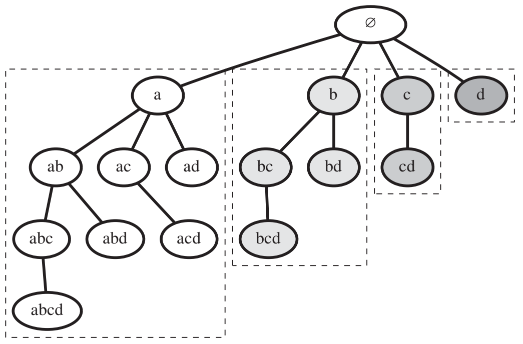


Figure 1: Itemsets in the equivalence classes $[a]$, $[b]$, $[c]$, and $[d]$

To keep the size of the intermediate results during mining as small as possible, ECLAT employs *equivalence class clustering* to break the search space of possible frequent itemsets into smaller sub-problems. All candidate k-itemsets that share a prefix of length $(k-1)$ and thus differ only in their last item form an equivalence class. For example, the itemsets $\{ab, ac, ad\}$ and $\{bc, bd\}$ would be in the equivalence classes $[a]$ and $[b]$, respectively. All itemsets produced by an equivalence class are independent from itemsets produced by other classes, i.e., for building larger itemsets based on the itemsets of a class, no itemset of any other class is required. This effectively reduces the number of intermediate tid-lists, which have to be maintained, and also allows an effective parallelization. Figure 1 illustrates the mapping of itemsets to the four equivalence classes $[a]$, $[b]$, $[c]$, and $[d]$.

The candidates are generated and tested bottom-up within each class. There is, however, no distinct candidate generation; it is performed simultaneously while mining the sets. The complete algorithm works as follows:

Preparation The database is scanned twice. The first scan reveals the frequent items F_1 while the second scan is used to transform the database into the vertical layout. For each frequent item $\gamma \in F_1$, a tid-list $\mathcal{T}(\gamma)$ is created, which contains all transactions that contain the item γ.

Bottom up mining **bottom-up**$(\cdot)$ is called once with F_1 as parameter and then recursively as the search space is traversed.

Bottom-up mining A set F_k of frequent k-itemsets is used as input. The equivalence class $[\alpha_i]$ for each itemset α_i of F_k is recursively mined.

Bottom-up(F_k): For each itemset $\alpha_i \in F_k$ with $i = 1, 2, \ldots, |F_k|$ do:

1. Set $F_{k+1} = \emptyset$.
2. For each $\alpha_j \in F_k$ with $j = i+1, \ldots, |F_k|$ do:
   (a) Intersect the tid-list $\mathcal{T}(\alpha_i)$ with the tid-lists $\mathcal{T}(\alpha_j)$ to obtain the tid-list $\mathcal{T}(\beta)$ for the itemset β where $\beta = \alpha_i \cup \alpha_j$.
   (b) If β fulfills ξ with $|\mathcal{T}(\beta)| \geq \xi$, then add β to F_{k+1}.
3. If F_{k+1} is not empty, then call **bottom-up**() recursively with F_{k+1} as parameter.

Mining is finished, when the initial call of **bottom-up()** returns. Zaki et al. [13] provide more details about the algorithm.

2.2 Parallel Eclat

Although there are many sequential variants of ECLAT, there exist only few parallel versions of it. PARECLAT [14] can be run on multiprocessor as well as on distributed systems. Its works in three phases: The *initialization phase* is used for obtaining the frequent 2-itemsets (including the frequent items), creating the equivalence classes, and converting the dataset. Each class has an assigned weight—based on its cardinality—which is used for a greedy-based load distribution. The parallel conversion of the transaction database into the vertical layout is performed by dividing the dataset into equal-sized partitions; each thread converts a different range of tids, i.e., the tids for the transactions that are in its assigned partition. Knowledge about these tids, the threads' local count values, and global count values are exploited to obtain the global tid-lists in which the tids are lexicographically ordered. Each thread basically writes its tids starting from certain offsets in the final tid-lists. In the *asynchronous phase*—denoted as asynchronous because there is no synchronization between the threads (or processors) required—each thread generates frequent itemsets using its assigned equivalence classes; this is done via tid-list intersection as in the sequential algorithm. In the last phase, the *reduction phase*, all threads are synchronized and the results are combined.

PARECLAT works well as long as the number of threads is rather low. For a large number of threads, PARECLAT's load distribution is too coarse grained to feed all threads with an equal amount of work.

2.3 Intel Xeon Phi

As mentioned before, our algorithm MCECLAT is intended for multiprocessor systems as well as for many-core coprocessors. The Intel Xeon Phi coprocessor [8] is such a many-core coprocessor, which provides a large number of threads.

The Xeon Phi is intended for high parallel applications and contains a large number of homogeneous x86 cores. The coprocessor we used, for example, has 61 cores running at a core frequency of 1.1GHz. Each core can further run multiple threads to hide latencies while accessing the device memory, i.e., four threads run per core, so up to 244 threads can run in parallel. Each core has a local L1- and L2-cache and can access the device memory (for our system 8GB GDDR5 memory) via a fast bidirectional ring. Thereby, the caches are coherent across the entire coprocessor.

Besides the large number of cores, the Xeon Phi provides a powerful vector instruction set with instructions—including scatter and gather—that operate on 512-bit registers. Hence, up to sixteen 32-bit integer or float values can be processed with one instruction. It is thus crucial to vectorize applications to obtain high performance on the coprocessor. Unfortunately, mcEclat cannot fully exploit this SIMD instruction set so that its performance relies on the many parallel threads.

Finally, the coprocessor runs an own operating system (linux), which allows two usage models for the coprocessor: (1) An application can be directly compiled for and run on it and (2) an application can be run on the host processor and some load can be offloaded on the coprocessor during processing. Thereby, offloading involves copying data from the host's memory into the coprocessor's memory. In both usage models, the offloaded task/application runs independently from the host processor. Communication is only required for exchanging results or data. Note that there are no restrictions on the code being run. Since regular x86 cores are used, different tasks could be run in parallel and recursive programs are possible.

3. MCECLAT

mcEclat has basically the same core algorithm as Eclat or parEclat. It scans a dataset being mined once to obtain the frequent items, converts the dataset into an internal representation, and thereafter starts recursive mining. Mainly the internal representation of the sets, the memory management, and parallel equivalence class mining differ from parEclat. In the following, we discuss these differences in more detail.

3.1 Internal dataset representation

Similar to various other Eclat implementations [2, 12], mcEclat uses *tid-bitmaps* to represent the tid-sets of the converted dataset. Instead of using lists to store all tids in which a frequent item occurs, the tids of an itemset (or item) are mapped to bits in a bitmap at certain positions. If the itemset α, for example, occurs only in the transactions 3, 5, 6, and 7, then the 3-th, 5-th, 6-th, and 7-th bit in α's tid-bitmap are set to one. All other bits in the bitmap are set to zero. The length of each tid-bitmap for a converted dataset is fixed and is determined by the largest tid; i.e., if a dataset has n transactions that contain at least one frequent item, then each tid-bitmap requires $\lceil n/8 \rceil$ bytes.

Tid-bitmaps have two major advantages compared to tid-lists: They often require less space than tid-lists for dense datasets with long transactions and they can be intersected with only bitwise `AND` instructions. Obtaining the support of an itemset then becomes counting the one bits in its respective tid-bitmap. This *bit population count* can be either performed using (1) lookup tables [2], (2) calculation using various bit operations [12], or (3) population count instructions. Both, our employed CPUs and the Xeon Phi coprocessor, provide a 64-bit `popcnt` instruction. We employ it within mcEclat to obtain the support of an itemset.

Converting a dataset into tid-bitmaps typically amounts to only a small fraction of mcEclats overall runtime. Each of the dataset's transactions needs to be parsed and for each of its frequent items a bit in the item's respective tid-bitmap must be set. Thereby, we store all bitmaps 512-bit aligned to avoid false sharing and enable aligned load instructions during the intersections when mcEclat is run on the coprocessor. Parallel conversion can be performed as for parEclat. When mining is offloaded to the coprocessor, it is sufficient to perform this task solely on the host processor. In our current implementation, we even perform this conversion only using a single thread. After all tid-bitmaps are created, they are transfered to the coprocessor. As suggested by our experiments, the transfer time is—for reasonable ξ values—negligible compared to the time required for mining.

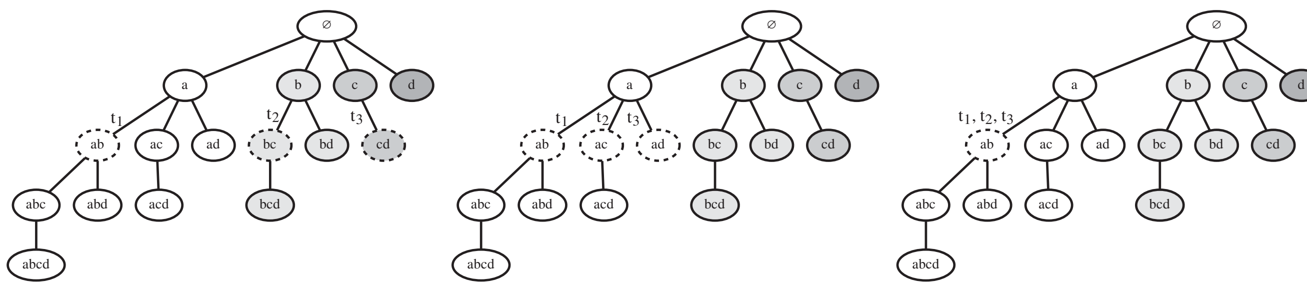
3.2 Memory management

An important part of mcEclat is the threads' memory management. Each thread allocates and deallocates memory for storing newly created tid-bitmaps during mining. Whenever a thread traverses down in the search space, it creates tid-bitmaps for itemsets that share all but the last item. For the class $[ab]$, for example, the tid-bitmaps for the itemsets abc and abd must be materialized (cf. Figure 1). They are removed as soon as all larger itemsets that are based on them are mined. Hence, a stack per thread is sufficient for such an allocation pattern. To provide stacks for each thread and avoid expensive `malloc` calls during processing, we allocate a large chunk of virtual memory using `mmap`, i.e., we allocate for each thread 1GB so that up to 244GB are allocated when all 244 threads are used. We pass the flag `MAP_NORESERVE` to `mmap` to enable such large chunks of virtual memory despite the small swap space. Since virtual memory is not mapped to physical memory until it is touched, mcEclat's memory footprint is typically much smaller, e.g., below the coprocessor's available 8GB. Hence, memory management is completely performed by the operating system. We further use huge pages (i.e., 2MB pages) to reduce the costs for the page mapping. On the coprocessor, the page size is controlled using the environment variable `PHI_USE_2MB_BUFFERS`. We observe an average performance improvement of about 10% with this optimization enabled.

3.3 Parallel equivalence-class mining

As mentioned before, the mining step dominates mcEclat's overall runtime as ξ gets sufficiently small. Hence, it is very beneficial to run it in parallel. In the following, we discuss three parallel equivalence class mining approaches, which are employed in combination to make mcEclat's mining step scalable.

ParEclat distributes each class to a single thread, which mines an assigned class *independently* from all other threads. In Figure 2(a), for example, the three threads t_1, t_2, and t_3 mine the classes $[a]$, $[b]$, and $[c]$ independently from one another. This approach exhibits low synchronization costs because each thread only synchronizes with other threads when it fetches the next class for mining. However, mining the classes independently often cause high load imbalances when a large number of threads is used. Threads that mine heavy classes—t_1 with class $[a]$ in Figure 2(a)—often finish later than other threads mining lighter classes. Such imbalances can only be avoided when there are much more classes than threads. This, however, is rarely the case for a large number of threads. Besides the load imbalance, the memory consumption of independent class mining is much higher than for sequential processing because all threads hold all of



(a) Independent class (b) Shared class (c) Shared itemset

Figure 2: Parallel equivalence class mining using three threads t_1, t_2, and t_3

their tid-bitmaps at the same time in memory.

The *shared class mining* (Figure 2(b)) partially avoids the problems of the independent class mining approach by mining a single class using multiple threads. This reduces the memory required for parallel mining but comes at the price of higher synchronization costs. The threads must be synchronized when they materialize a new tid-bitmap for a found frequent itemset, i.e., atomic increments are used to increase local variables that count the number of sets in a class. Load imbalance still occurs when the number of threads is larger than the number of tid-lists that need to be intersected within a class.

The threads build even the tid-sets of single itemsets together when *shared itemset mining* is employed (cf. Figure 2(c)). Since the tid-bitmaps are stored in cacheline granularity (512-bit), the threads can each process parts of the tid-bitmap being built without false sharing. Clearly, this approach allows the most fine-grained work distribution at the cost of a high synchronization between the threads. They must be synchronized always after two tid-sets are intersected to obtain the support of the new itemset. Therefore, we do not employ shared itemset mining in mcEclat.

Instead, we employ a hybrid between independent and shared class mining in mcEclat: *Grouped class mining* partitions the threads in groups. Each group has a different class assigned and all threads in a group process together a class. Group class mining is only performed in the first few recursion levels to allow an even load distribution with reasonable synchronization costs. In the deeper recursion levels, the thread groups are split up and the classes are processed independently by the threads. We thereby process *b* classes per thread simultaneously. This *blocked class* processing increases instruction-level parallelism by avoiding data dependent latencies and improves cacheline utilization.

4. EXPERIMENTAL EVALUATION

In this section, we provide the results for our experimental evaluation. We first give details about the setup. Thereafter, we provide the results for the scalability of our algorithm and how it performs compared to existing frequent itemset mining algorithms.

4.1 Setup

We conducted all our experiments on a multiprocessor system. It contains two Intel X5680 processors, which run at a core frequency of 3.3GHz, each has 6 cores and a 12MB L3-cache. The processors support Hyperthreading, so up to 24 threads can run in parallel. Each processor has 16GB of main memory attached so that 32GB are available for the complete system. The multiprocessor system further contains a Intel Xeon Phi coprocessor[1], which has 61 cores running at 1.1GHz and 8GB GDDR5 RAM. As mentioned earlier, each of these cores runs 4 threads in parallel. Hence, up to 244 threads can be run on the coprocessor.

Both the multiprocessor system and the coprocessor run linux. We implemented mcEclat in C++ and compiled it using `icc` from Intel Composer XE 2013 (13.1.0). We used OpenMP to enable thread-level parallelism. We passed the flags `-mmic` and `-no-offload` to obtain an executable with and an executable without offloading, respectively. We further employed intrinsics to integrate the used vector instructions. In all experiments, we measure the wall-clock time within the algorithms using `gettimeofday()`. The time for data transfers between the host system and the coprocessor and the amount of transfered data is measured by setting the environment variable `OFFLOAD_REPORT=2`.

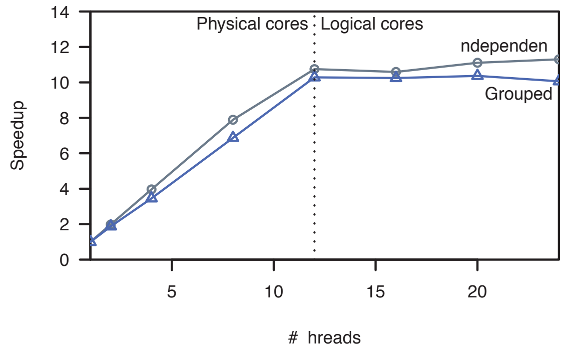
	# items	avg. card.	# transactions
`netflix`	480,189	5654.1	17,771
`BMS-POS`^{T}	515,597	2030.7	1,657
`webdocs`	5,267,656	177.2	1,692,082

Table 1: Characteristics of the used datasets

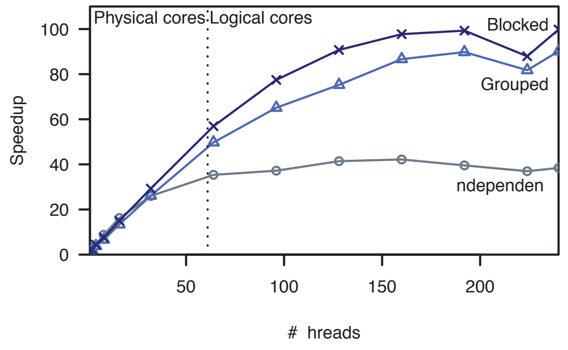
For the experiments, we use three realistic datasets that are eligible for Eclat. They all consist of long transactions and contain many distinct items. The `webdocs` dataset [10] is created from a collection of web documents and is taken from the FIMI repository [3], which contains efficient implementations and datasets for frequent-itemset mining. The `BMS-POS`^{T} dataset is the transposed version of a market-basket analysis dataset [3]. The `netflix` dataset is built from a freely available dataset that was originally not intended for frequent itemset mining. The original dataset contains movie ratings and was used in the Netflix Prize competition.[2] We do not use other freely available datasets because (1) they are often too small so that offloading would be useless and (2) many of them are not eligible for Eclat so that other algorithms like FP-growth or Apriori perform better on them. Table 1 summarizes the characteristics of the three used datasets. The generated `netflix` and `BMS-POS`^{T} dataset as well as the complete source code of our implementations are available online.[3]

[1]We use an engineering sample (B0 hardware) with Gold software.
[2]See `http://www.netflixprize.com/` for more information.
[3]`http://wwwdb.inf.tu-dresden.de/schlegel`.

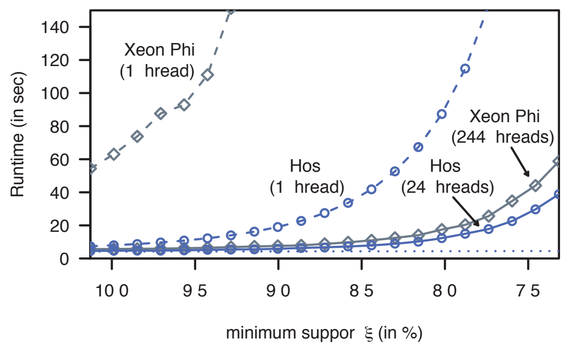
Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 3, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485281



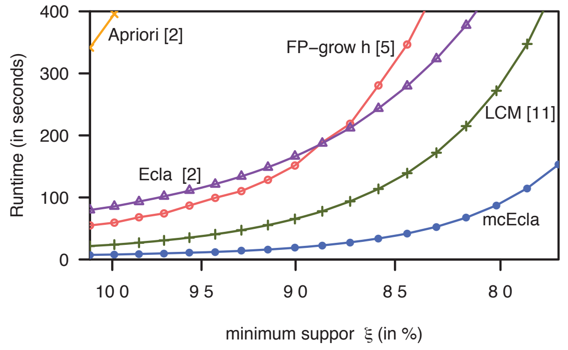
(a) Scalability on the host system with $\xi = 7.9\%$ (mining only)



(b) Scalability on the Xeon Phi coprocessor with $\xi = 7.9\%$ (mining only)



(c) Execution times on both platforms



(d) Comparison with existing algorithms (single-threaded and host only)

Figure 3: Experimental results on `netflix`

4.2 Scalability experiments

In the first set of experiments, we evaluate MCECLAT's scalability by running it on both platforms for a varying number of threads. We compare the performance of INDEPENDENT and GROUPED class mining. For the latter, we further employ our blocked class processing (BLOCKED) with $b = 2$. Figure 3(a) illustrates MCECLAT's performance when it is solely run on the host processor. In what follows, we use INDEPENDENT's single-threaded runtime as base for obtaining the speedups. As can be seen, both equivalence class mining approaches perform similarly. GROUPED is slightly slower than INDEPENDENT because of the higher synchronization between the threads in a group; the better load distribution does not pay off for such a small number of threads. Interestingly, BLOCKED (not shown) is not beneficial for the host-only version. We obtain similar results for the `BMS-POS`^{T} dataset. On the `webdocs` dataset, however, MCECLAT does not scale beyond four threads. This dataset has many transactions—leading to long tid-bitmaps. While intersecting these large bitmaps, the threads saturate the available memory bandwidth so that more threads do not reduce the runtime. More elaborate blocking techniques—at cacheline level—need to be developed for such datasets.

GROUPED and BLOCKED scale much better than INDEPENDENT when mining is offloaded to the Xeon Phi coprocessor. As illustrated in Figure 3(b) for the `netflix` dataset, BLOCKED achieves speedups up to 100x whereas INDEPENDENT achieves at best 40x. We obtain similar results for the other datasets with, however, smaller speedups (cf. the appendix).

Figure 3(c) illustrates the runtime of the host-only and coprocessor version when mining `netflix`. The latter version is single-threaded multiple times slower than the single-threaded host-only version. Multi-threaded, however, the coprocessor version exploits the large number of threads of the Xeon Phi coprocessor. Thus, the runtimes of both versions approach each other when the maximum number of threads is used on both systems.

		CPU		Xeon Phi	
# threads		1	24	1	244
`netflix`	$\xi = 9.0\%$	14.7	1.4	199.8	3.1
	$\xi = 7.6\%$	202.7	18.1	2643.5	28.8
`BMS-POS`^{T}	$\xi = 1.0\%$	76.8	7.6	1127.9	21.2
`webdocs`	$\xi = 10.6\%$	7.7	2.7	112.9	5.7
	$\xi = 7.7\%$	111.2	38.1	1502.7	39.9

Table 2: Runtime of mining (in seconds)

Table 2 shows the runtimes (mining only) for both versions with SHARED enabled on our three datasets. The coprocessor version (with blocking enabled) is single-threaded up to 14.6x slower than the host version because the host processor cores have a high core frequency and support out-of-order processing. Furthermore, mcEclat does not fully exploit the Xeon Phi's powerful vector processing capabilities. When the maximum number of threads is employed on both systems, however, the performance difference is smaller. On `webdocs`, for example, both versions have almost the same runtime. When considering that (1) a single Xeon Phi card is much cheaper than a multiprocessor system and (2) multiple cards can be employed in a single server, then offloading load to the coprocessor is beneficial.

4.3 Comparison with existing algorithms

In the next set of experiments, we compare the host version of mcEclat with available, highly efficient frequent itemset mining implementations taken from the FIMI repository [3]. The purpose of these experiments is to show that mcEclat is competitive with existing algorithms. Otherwise, it would be meaningless to parallelize it. We measure the execution times of Apriori [2], Eclat [2], FP-growth [5], and LCM [11]. These algorithms cover a broad spectrum of available sequential mining algorithms and typically perform best among all algorithms used in the FIMI competition [4]. Details about these four existing algorithms can be found in Section 5.

Figure 3(d) illustrates the execution time on `netflix` for various ξ values. As can be seen, Apriori performs worst. It is much slower than the other algorithms because it is optimized for processing datasets containing mostly small transactions (cf. the discussion in Section 5). FP-growth and Eclat show almost the same performance and LCM performs best of the four competitors. mcEclat is always the fastest of all five algorithms within the tested ξ range. The results are similar on `webdocs` and `BMS-POS`^{T} dataset. mcEclat is always among the fastest of the five tested algorithms on these Eclat-suitable datasets. Summing up, mcEclat is competitive with existing mining algorithms.

5. FURTHER RELATED WORK

In this section, we review related work that is not already covered within the paper, i.e., we discuss sequential mining algorithms that are not based on Eclat.

Apriori [1] is considered the first algorithm eligible for mining large datasets. It represents the transaction database in the *horizontal layout* in which the filtered transactions are stored one after another in memory. The frequent itemsets are obtained by counting candidate itemsets in the transactions. Apriori typically performs best on datasets that have many small transactions.

FP-growth [6] represents the transaction database using *frequent-pattern trees*. Such trees resemble prefix trees and are often smaller than the transaction database. During mining, the frequent-pattern trees are repeatedly traversed to build smaller ones of them and thereby obtaining the frequent itemsets. FP-growth usually outperforms Eclat on datasets with small and medium-sized transactions or when mainly small frequent itemsets are found. In the remaining cases, Eclat performs better because building the prefix trees then does not payoff.

The LCM algorithm [11] resembles the Eclat algorithm but maintains besides the transactions stored in the vertical layout also their representation in the horizontal layout. The latter is used to obtain the former without intersections. LCM often performs well on the same datasets on which Eclat performs well.

6. CONCLUSION

Frequent-itemset mining is an essential part of the association rule mining process and has many application areas. Existing parallel mining algorithms often cannot exploit the large number of threads that is provided by increasingly popular many-core systems like Intel Xeon Phi or large multiprocessor systems. In this paper, we propose mcEclat, which is a highly scalable parallel version of the well-known mining algorithm Eclat. mcEclat converts a dataset being mined into a set of tid-bitmaps, which are repeatedly intersected to obtain the frequent itemsets. For a highly scalable mining, we discuss and evaluate three parallel equivalence class mining approaches. The hybrid between independent and shared class mining shows the best performance. In our experiments, we observe that mcEclat outperforms efficient existing algorithms and achieves high speedups of up to 100x when run on a many-core coprocessor that has 61 cores.

Acknowledgements

This work is partly funded by the European Regional Development Fund (EFRE) and the Free State of Saxony under the grant 100067363 ("cool iBit computing"), the ESF young researcher group "IMData" (grant 100098198), and the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" and the grant LE 1416/22-1.

We further would like to thank the ZIH (TU Dresden) for their valuable comments and Intel for providing the hardware.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[2] C. Borgelt. Efficient implementations of apriori and eclat. In *FIMI*, 2003.

[3] FIMI. Frequent itemset mining implementations repository. http://fimi.cs.helsinki.fi/, November 2004.

[4] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: report on fimi'03. *SIGKDD Explorations*, 6(1):109–117, 2004.

[5] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[7] M. HooshSadat, H. W. Samuel, S. Patel, and O. R. Zaïane. Fastest association rule mining algorithm predictor (farm-ap). In *C3S2E*, pages 43–50, 2011.

[8] *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. Intel, September 2012.

[9] E. Li and L. Liu. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, pages 1275–1285, 2007.

[10] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Webdocs: a real-life huge transactional dataset. In *FIMI*, 2004.

[11] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.

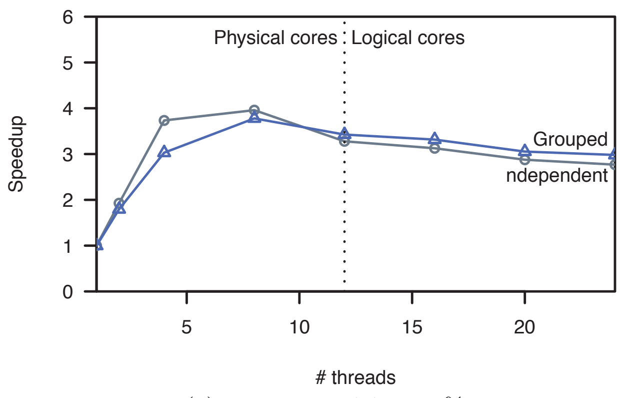
[12] M. Wei, C. Jiang, and M. Snir. Programming patterns for architecture-level software optimizations on frequent pattern mining. In *ICDE*, pages 336–345, 2007.

[13] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical report, 1997.

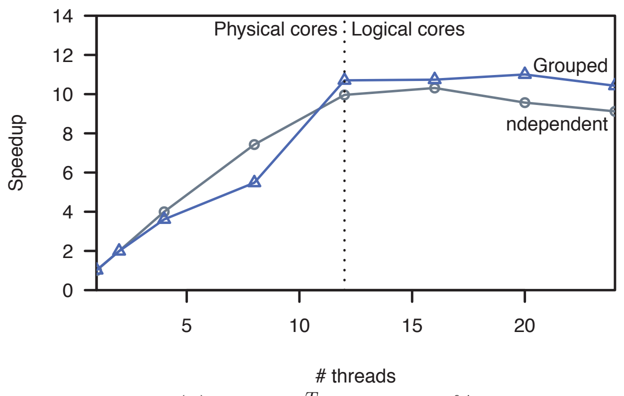
[14] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Min. Knowl. Discov.*, 1:343–373, December 1997.

APPENDIX

In this section, we provide further performance graphs. Figure 4(a) illustrates the scalability of the host-only version of MCECLAT on `webdocs`. As can be seen, both class mining approaches reach their peak speedup already for 4 threads. Figure 4(b) illustrates the scalability of the host-only version on `BMS-POS`^{T}. On this dataset, MCECLAT scales up to 12 threads almost linear.



(a) `webdocs` and $\xi = 8.3\%$



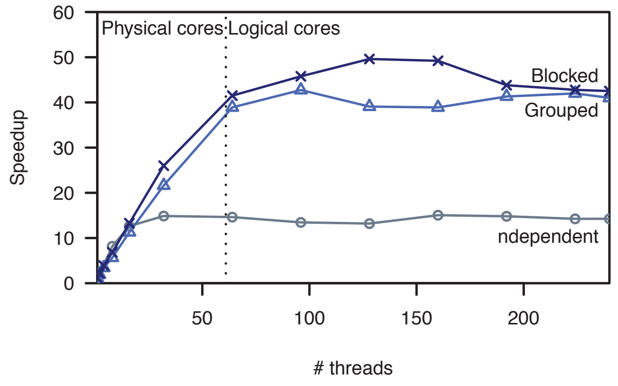
(b) `BMS-POS`^{T} and $\xi = 1.0\%$

Figure 4: Scalability of mcEclat's host-only version on the 12-core multiprocessor system

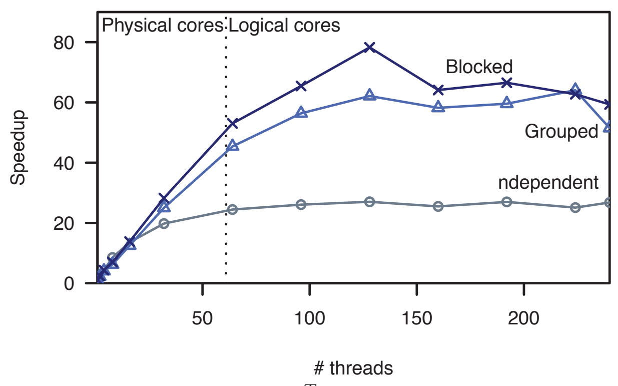
Figure 5(a) and Figure 5(b) illustrate the scalability of MCECLAT's coprocessor version on `webdocs` and `BMS-POS`^{T}, respectively. Group class mining—with blocking enabled—scales on both datasets better than independent class mining. The former is up to 4x faster than the latter for more than 60 threads.

Figure 6(a) illustrates the runtime of MCECLAT on `webdocs` for both platforms. The host-only version is single-threaded multiple times faster than the single-threaded coprocessor version. The latter, however, scales better so that both multi-threaded versions show a similar performance. On `BMS-POS`^{T}, the host-only version is single-threaded—as well as when the maximum number of threads are employed on both platforms—multiple times faster than the coprocessor version.

Finally, Figure 7(a) and Figure 7(b) illustrate the results for the comparison of single-threaded MCECLAT (host-only) with the existing frequent-itemset mining algorithms on `webdocs` and `BMS-POS`^{T}, respectively. MCECLAT performs best on the former dataset for $\xi > 7\%$. For lower ξ values, ECLAT is the fastest among the five tested algorithms. On `BMS-POS`^{T}, LCM is the fastest of all five tested algorithms for $\xi > 1.1\%$. For lower ξ values, MCECLAT is faster than all other algorithms under test.

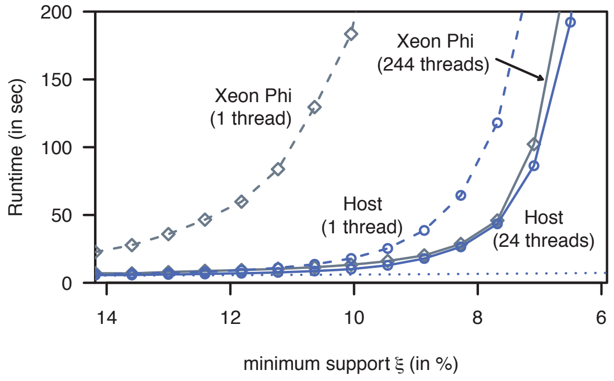


(a) webdocs and $\xi = 8.3\%$

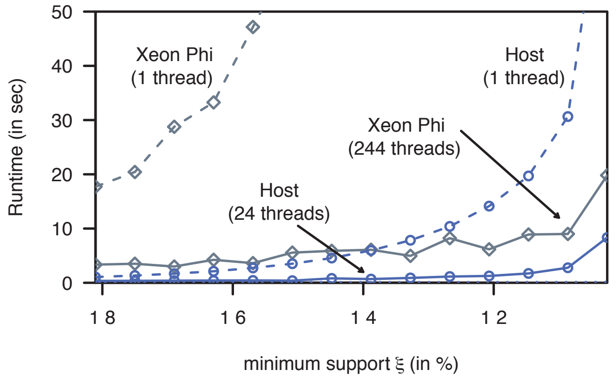


(b) BMS-POS^T and $\xi = 1.0\%$

Figure 5: Scalability of mcEclat's offload version on the Xeon Phi coprocessor

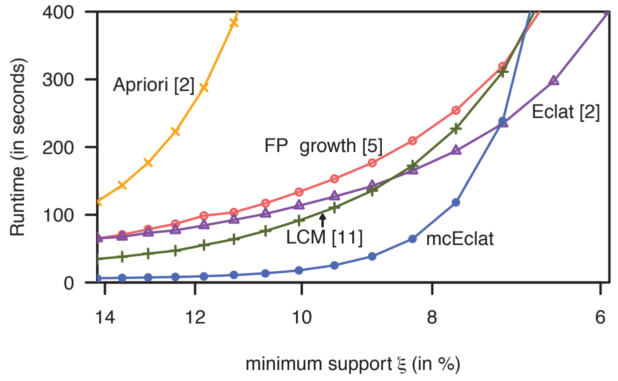


(a) webdocs

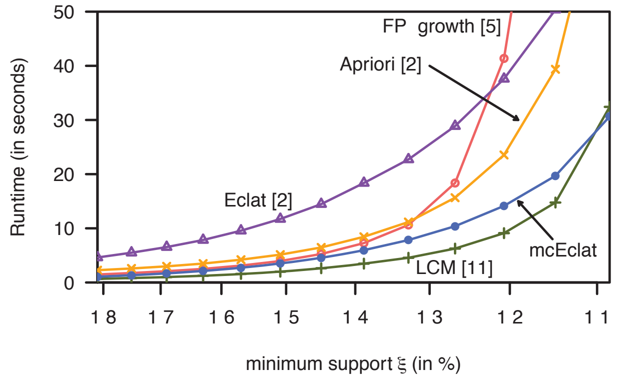


(b) BMS-POS^T

Figure 6: Execution times on both platforms



(a) webdocs



(b) BMS-POS^T

Figure 7: Comparison with existing algorithms (single-threaded and host only)