Discovering Closed Frequent Itemsets on Multicore: Parallelizing Computations and Optimizing Memory Accesses

Benjamin Negrevergne,
Alexandre Termier and
Jean-François Méhaut
Laboratoire d'Informatique de Grenoble
{firstname}.{name}@imag.fr

Takeaki Uno
National Institute of Informatics
uno@nii.jp

ABSTRACT

The problem of closed frequent itemset discovery is a fundamental problem of data mining, having applications in numerous domains. It is thus very important to have efficient parallel algorithms to solve this probem, capable of efficiently harnessing the power of multicore processors that exists in our computers (notebooks as well as desktops). In this paper we present PLCM$_{QS}$, a parallel algorithm based on the LCM algorithm, recognized as the most efficient algorithm for sequential discovery of closed frequent itemsets. We also present a simple yet powerfull parallelism interface based on the concept of Tuple Space*, which allows an efficient dynamic sharing of the work. Thanks to a detailed experimental study, we show that PLCM$_{QS}$ is efficient on both on sparse and dense databases.*

KEYWORDS : pattern mining, frequent closed itemset, multicore, memory accesses

1. INTRODUCTION

Frequent pattern discovery is one of the major domains of data mining. This domain was pioneered by the works of Agrawal et al. on the Apriori algorihm [1] for discovering frequent itemsets. The problem of discovering frequent itemsets consists, given a transaction database where each transaction is made of items and given a minimum support threshold $minsup$, in discovering all the itemsets that occur at least $minsup$ times in the database. This problem is the simplest in the domain of pattern mining, where patterns to be mined can also be sequences, trees or graphs. However, it has many applications, for example in the analysis of sales data. Moreover, thanks to its relative simplicity, all the significant algorithmic improvements to frequent pattern mining have been discovered first for frequent itemset mining, before being adapted to more complex patterns. Let us cite for example the case of closure [2] or algorithms without candidate generation [3].

In 2003 and 2004, the FIMI workshop [4] confronted frequent itemset mining algorithms in order to select the best solutions. The winner of FIMI 2004, LCM2 [5], combines high level improvements coming from enumeration theory and low level improvements optimizing the tradeoffs between computation time and memory usage. Since 2004, no sequential algorithm gave better performances.

However, the sequential algorithms presented at FIMI in 2004 are not able to exploit the parallelism capabilities of modern processors. Since 2005, processor design have undergone a radical shift. Physical limits have been reached, preventing further increase of clock speed and thus sequential performance. In order to enable a performance increase, chip makers now integrate several processing cores on a single chip. To harness the power of these multicore processors, it is necessary to design parallel algorithms.

Because they set up the main issues of pattern mining, frequent itemset mining algorithms are good candidates for parallelization. This paper is an extended version of [6] where we present a parallel closed frequent itemset mining algorithm based on LCM and capable of scaling up efficiently with the number of threads on multicore processors. We present a new parallelism environment, Melinda, based on the concept of Tuple Space. Melinda relies on two simple primitives, but allows to use internally efficient computation distribution models. We present the PLCM algorithm which uses

this interface to parallelize LCM. In addition to [6], we present a detailed study and highlight several issues with the memory accesses of PLCM, coming from its sequential inheritance. We propose two optimizations specific to the parallel case, integrated into the algorithm PLCM_{QS}. Experiments show that PLCM_{QS} is the most efficient and generic solution for parallel mining of closed frequent itemsets.

The outline of the paper is as follows: In section 2, we briefly define the problem of extracting closed frequent itemsets, and we present the state of the art in parallel pattern mining on multicore processors. In section 3, we explain the basics for understanding the LCM algorithm. We then describe in section 4 the Tuple Space parallel framework and the PLCM algorithm designed with this framework, as well as the optimizations leading to PLCM_{QS}. Section 5 gives a detailed experimental study comparing PLCM and PLCM_{QS} with the state of the art. Last, in section 6, we conclude and give directions for future work.

2. PRELIMINARIES

In this section, we give the main notations used throughout the paper and define the problem of extracting closed frequent itemsets We then review the state of the art.

2.1. Problem Definition

Let $\mathcal{I} = \{1, ..., n\}$ be a set of items. A transaction database on $\mathcal{I}$ is a set $T = \{t_1, ..., t_m\}$ such that each t_i is included in $\mathcal{I}$. A t_i is called a transaction. A subset P of $\mathcal{I}$ is called an itemset. For an itemset P, a transaction containing P is called an occurrence of P. The tid-list of P, denoted by $tidlist(P)$, is the set of all the occurrences of P. $|tidlist(P)|$ is called the support or frequency of P, and is denoted by $support(P)$. For a given minimum support threshold $minsup$, an itemset P is frequent if $support(P) \geq minsup$. We denote by $\mathcal{F}$ the set of all frequent itemsets. For any pair of itemsets P and Q, we say that P and Q are equivalents if $tidlist(P) = tidlist(Q)$. This relationship induces equivalence classes on itemsets. Itemsets maximal w.r.t. inclusion in each equivalence class are called closed itemsets. We denote by $\mathcal{C}$ the set of closed frequent itemsets.

The problem we are interested in is, given a transaction database $\mathcal{T}$ and a minimum support threshold $minsup$, to extract all the closed frequent itemsets of $\mathcal{T}$.

2.2. State of the Art

At the end of the 1990s, numerous works have studied parallel frequent pattern mining for clusters [7, 8]. These works were interested both on the capacity to increase the size of the data handled and on improving performance. We only mention them, and focus this state of the art on parallel frequent pattern mining on multicore architectures, whose characteristics are slighlty different.

Parallel frequent pattern mining on multicore processors was pioneered by Buehrer et al. [9]. Based on gSpan [10], they proposed a parallel frequent graph mining algorithm with excellent scale-up properties. Their contribution comprises an efficient way to decompose work and to explore in a depth-first way the search space. They also propose a way to exploit temporal locality of the cache.

Lucchese et al. [11] proposent similar strategies for mining closed frequent itemsets, using as Buehrer et al. work stealing techniques. When a thread A has a lot of work and another thread B do not have work, B can "steal" part of the work of A in order to improve load balance. The solution of Lucchese et al. contains optimizations for improving cache usage when creating conditional databases (called projections in their paper), a classic technique in closed frequent itemset mining.

The most recent works are from Tatikonda et al. [12] on parallel frequent tree mining. Their algorithm scales up very well with the number of cores, with a quasi-linear speed-up on a lot of real-world databases. Tatikonda et al. show in their paper that conversely to traditional monocore architectures, the problem on multicore architectures lies more on memory accesses than on computations. Memory is a shared resource for all the cores: if all the cores make numerous simultaneous memory accesses, data transfer bus between the memory and the processor will be saturated and the latency of memory accesses will drastically increase, with a high impact on program performances. Indeed, bandwith of most current multicore processors is not enough to support numerous and simultaneous memory accesses from all cores. Thus new algorithmic choices must be made, that can be opposite to those that gave good results in the sequential case. Another important contribution is the importance of reducing bandwidth pressure. They do so by reducing as much as possible the working set size of their algorithm, with excellent results. One of the techniques for reducing working set size in the cache is to avoid pointer based data struc-

tures, which have a very bad locality in the cache and thus force the algorithm to make too many memory accesses.

3. THE LCM ALGORITHM

We describe in this section the basic principles of the Linear time Closed itemset Miner (LCM) algorithm. Our explanations are based on version 2 of this algorithm, described in [5]. The main characteristic of this algorithm is to have a linear complexity in the number of closed frequent itemsets to find (proved in [13]). This distinguishes LCM from all other closed frequent itemset mining algorithms, and allowed it to win the challenge organized in FIMI'2004.

LCM is an algorithm from the family of backtracking algorithms. These algorithms are based on the following recursive structure, given for the simple case of frequent itemsets:

1. The algorithm receives a previous solution as input, i.e. a frequent itemset P
2. This solution is outputed as a result.
3. For each item e that is greater than the greatest item of P, the algorithm generates an itemset $P \cup \{e\}$ and tests its frequency
    - If $P \cup \{e\}$ is frequent then the algorithm make a recursive call on $P \cup \{e\}$

This kind of algorithm generates a covering tree on the frequent itemsets. In the case of LCM, each recursive iteration receives a closed frequent itemset P as input. LCM computes all the closed frequent itemset proceeding directly from P. We thus obtain a covering tree structure on all the closed frequent itemsets, which allows to enumerate them without duplication. LCM thus does not need to keep in memory informations on the closed frequent itemsets it has previously generated.

This closed frequent itemset enumeration is optimal, but further optimizations are needed in order to improve the efficiency of the algorithm by reducing computations time and memory requirements.
Database reduction: To compute frequency and closure, the algorithm makes numerous accesses to the database. It is thus especially interesting to reduce significantly the size of the database. The technique used is to:

- eliminate infrequent items
- eliminate items included in all transactions (and store them in a separate place)
- merge all identical transactions after all these eliminations, and give a weight to the unique resulting transaction in order to have correct frequency computations. The merging is performed after sorting transactions with a radix sort.

This technique can be used at each iteration in order to have reduced databases, that will be processed faster. These databases are called conditional databases, because they are also restricted to the transactions supporting the closed frequent itemset given as input of the recursive call.
Occurrence deliver: Frequency counting step is one of the most costly steps of (closed) frequent itemset mining. Uno et al. proposed a technique called occurrence deliver, which allows to compute the occurrences of all the extensions $P \cup \{e\}$ of the closed frequent itemset given as input of the recursive call, in a single pass over the database.
Frequent items reordering: The LCM algorithms renames the items so that each item has a number corresponding to its rank in decreasing frequency order. By exploring the branches from the less frequents items, LCM is thus assured to have the best database reduction.

We will not give more details on the LCM algorithm and its optimizations. We highly recommend the interested reader to read [13], which describes in depth the algorithm and gives all the theoretical material necessary to its understanding, as well as their proofs.

4. PLCM : PARALLEL LCM FOR MULTI-CORE PROCESSORS

In this section's first part, we present Melinda, a parallel execution model based on work sharing. Melinda is a simple paradigm that can easily be adopted by programmers with no skills in parallel programming. We then describe our PLCM algorithm implemented with Melinda.

4.1. Melinda: a Work Sharing Model

The Melinda execution model is based on the idea that the computation work must be shared between processors and cores. It directly derives from the Linda model defined by D. Gelernter[14]. In Melinda, the work is dropped in a global space (the Tuple Space) by the threads. When threads are idle they can extract work as Tuples from the Tuple Space and process them. The insertion primitive (put) adds a tuple in the Tuple Space. The extraction primitive (get) extracts a tuple from the Tuple Space. The get primitive may be blocking if the Tuple Space is empty.

The application completes when the Tuple Space is empty and all threads are blocked trying to get a Tuple.

Melinda hides low-level details of data sharing and synchronization thous suits programmers with no experience with parallel programming.

Melinda is implemented on SMP machines with multicore processors. The memory for the Tuple Space is allocated in the shared memory. To reduce the synchronization cost and improve data locality, the Tuple Space is composed of several memory banks. We count the total number of tuples in the tuple space with a single integer that can be incremented and decremented with atomic operations available in modern processors. No other global synchronization is needed. Each bank is individually synchronized with mutex, but there is generally no contention. A memory bank is attached to one thread. Each thread put and get its tuples in its bank thus the data is not migrated from one core to another, unless it is needed to balance the work.

4.2. Parallelization of LCM

In the case of PLCM, a Melinda tuple contains the parameters of a recursive call. The parallelization is thus done on the tree formed by the recursive calls. However, the creation of a tuple by node of the recursive calls tree can generate a lot of tuples, a lot of which will have very few work associated. Creating and managing tuples comes with some overheads, tuples number must not be too high. We chose to restrict the creation of tuples to the first depth levels of the recursive calls tree. The recursive calls tree having usually a high breadth factor and a small depth, this technique gives good results.

4.3. Optimizations

Table 1 gives preliminary results for the PLCM algorithm described above. The machine is a quad-socket Intel Xeon 7460 at 2.66 GHz, with 6 cores per socket, for a total of 24 cores. There are 64 GB of RAM. The databases used are connect, a dense database, and T40I10D100K, a sparse database (see section 5 for a detailed description of the databases). The second column of the table gives the mining time when only one thread is used (sequential time), and the next columns give the speedup obtained when using more threads. These figures confirm that using parallelism allows to have a smaller mining time. However the algorithm performance do not scale well with the number of cores.

Table 1. Speedups for Initial Version of PLCM

Database/#cores	1	4	8	16	24
connect@4.4%	*127s*	3.48	5.22	5.41	5.66
T40I10D100K@0.8%	*128s*	3.7	5.6	6.1	6.4

Running a profiler on the code shows that most of the mining time is spent during the database reduction. The database reduction drastically reduce the size of the transactions by sorting the transactions with a radix sort, processing a prefix intersection(cf. [15]), and merging the transactions that became the same. We isolated the database reduction in a test program. This test program spawns N threads, where each thread executes the same code for the database reduction, and is given the same input data. On a perfect architecture the running time, should stay the same whether N is equal to 1 or to the number of cores. We give below the steps of the test program:

1. load the database
2. make a "warm up" sequential sort on the database
3. perform N_S database reductions, and get and average time
4. spawn N threads, make a "warm up" database reduction on each thread
5. perform N_S database reduction on each thread, and get an average time

We executed the test program in the Intel VTune performance analyzer. VTune is a very powerfull profiling tool, which allows to access a wealth of information for the program to profile. One of its most interesting feature is the easy access to the processor performance counters. Figure 1 shows the chronogram for the $INST_RETIRED.ANY$ counter, which counts the number of instructions completely executed per sample of time, with a sample time set to 1ms. There are 12 threads running. The steps of the test program are indicated in the figure. The X-axis represents time. On the Y-axis, there is one line per thread. For each thread, one histogram bar represents the number of instructions retired per sample of time. Higher bar means more instructions retired.

Step 3 gives the reference time for the database reduction test: 200 ms. The time of the tests made in parallel, given by the duration of step 5, is 800 ms. So it was 4 times longer to make the sequential sorting done in step 3 when 12 processors are doing it simultaneously. This surprising result cannot be explained by a problem of load balance or computation granularity: the same code is executed, and the threads finish almost together as confirmed by Figure 1.

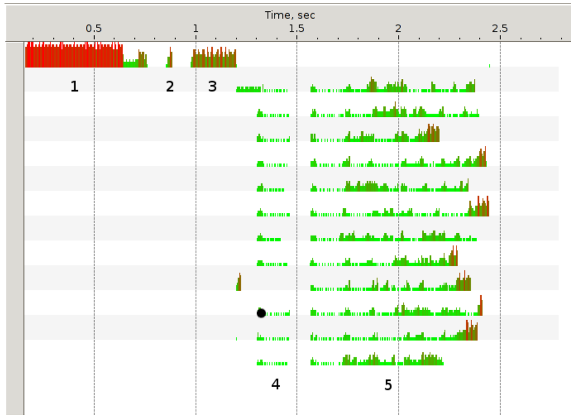

Figure 1. Instruction Retired for Database Reduction (radix sort), 12 Simultnateous Instances

A better hypothesis is that the threads are making a heavy usage of the cache, and are either polluting each others cache or making too many memory accesses, causing high bandwidth pressure. The increasing number of cache misses at the Last Level Cache (LLC) shows a bad cache usage, and implies a heavy bandwidth pressure when a large number of cores (for instance 24) are performing database reduction simultaneously. As a consequence, the latency of the memory accesses increases dramatically, which explains the degradation of performances when the number of threads increases. This can be further confirmed by running the test program on an Intel Core i7 920 with 4 cores. This recent processor contains a lot of performance monitoring events. Some of these events, for instance *MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_X*, can record the latency of the memory instructions performed. We thus recorded the memory latencies for the sequential sorting test (step 3), and for the parallel sortings with 2,3 and 4 cores (step 5). The results are reported in Figure 2. The X-axis represents the number of threads, and the Y-axis represents the percentages of memory accesses having a given latency in a cumulative way. For example in the case of 1 thread, 67% of all memory accesses have a latency of 16 cycles or less, 87% of all memory accesses have a latency of 32 cycles or less, and so on. This figure clearly shows that as the number of threads increase, the proportion of memory accesses with low latencies (less or equal to 32 cycles) diminishes, while the proportion of accesses with high latencies (more or equal to 64 cycles) increases.

The previous experiments have allowed to isolate the problem with database reduction: it makes too many cache misses, quickly saturating the bus when there are a lot of threads. Bus saturation conducts to higher

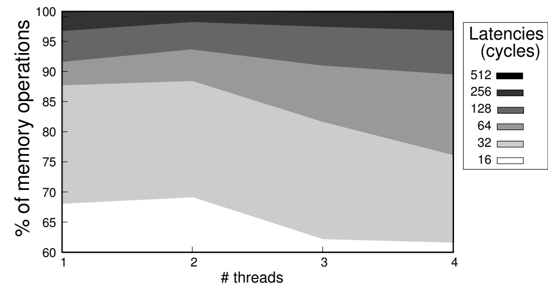

Figure 2. Latencies of Database Reduction, Core i7

memory access latencies and ultimately longer execution times. Analyzing the code for database reduction is necessary to understand this behavior. Internal algorithm like radix sort for sorting transactions has been carefully selected by the authors of LCM because it is the fastest sequential algorithm for sorting transactions. This speed is obtained by using a lot of additional memory during the execution of radix sort. In facts, the transaction database to sort is entirely rebuilt in a temporary structure, in a vertical format (for each item, lists of transaction ids are stored). The memory traffic thus comprises : reads in the original database, writes in the temporary vertical database, and reads in the temporary vertical database. When a lot of radix sort algorithms operate in parallel, the total amount of memory traffic generated by these operations is too heavy for current processor's bandwidth, as we have demonstrated before.

In the parallel case, when a lot of threads are running, a more "bandwidth-friendly" algorithm must be used. We thus replaced radix sort with quicksort. Quicksort can use some stack memory for its variables, but it does not use any byte in the heap, contrarily to radix sort. We present in Figure 3 the VTune graph when replacing radix sort with quicksort in the test program. This time, execution time for step 3 (sequential) and step 5 (parallel) are almost identical, confirming that quicksort makes a better usage of the cache.

However the running time for quicksort is much higher than the running time of radix sort, and our testings shown that the run times for PLCM with this version of quicksort were not lower than with radix sort, even if there were better scale up properties. The reason is that radix sort is tighly integrated with one of the most important optimizations of LCM for reducing run time. One of our contributions is thus a modified quicksort algorithm completely integrated with suffix intersection, without using extra heap space compared to a standard quicksort.

Using this new version of quicksort improves the scale

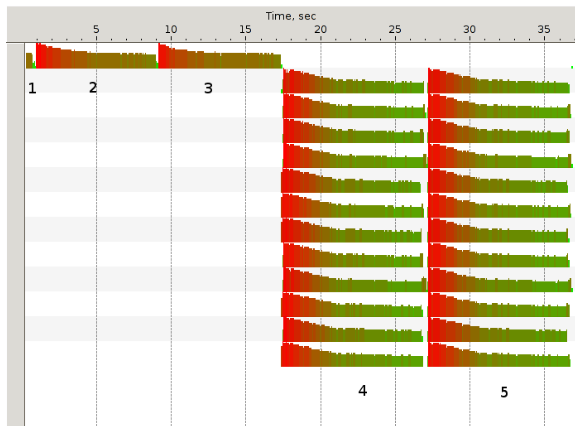

Figure 3. Instruction Retired for Database Reduction (quick sort), 12 Simultaneous Instances

up capabilities of PLCM. However we quickly realized that when databases to sort become too large, even when running in parallel, there are not enough processors to absorb the execution time penalty coming from the higher complexity of quicksort. We thus need an other algorithmic improvement to keep the size of the databases low. Our solution was to extend the item reordering strategy explained in section 3, in order to perform it at each recursive iteration (instead of only at load time in original LCM). In sequential LCM, this operation is too costly to bring benefit. But in the parallel case, the additional database reduction obtained by this optimization greatly helps quicksort which performs better on small databases.

We call PLCM$_{QS}$ the final algorithm obtained by integrating the previous optimizations into PLCM. The next section will give detailed comparisons between PLCM and PLCM$_{QS}$.

5. EXPERIMENTS

We compare in this section PLCM, PLCM$_{QS}$, and MT_Closed [11], which is the only other parallel approach for mining closed frequent itemsets. The MT_Closed implementation is the original C++ implementation. It is designed to be efficient on dense datasets.

The implementations of PLCM and PLCM$_{QS}$ are our C++ implementations. They do not share code with the original C implementation of LCM: they have been rewritten from scratch, using Tuple Spaces for parallelism management. The Tuple Spaces API is implemented in C++ and Posix Threads.

The experiments are conducted on a quad-socket Intel Xeon at 2.66 GHz, each socket having 6 cores, for a total of 24 cores. There are 64 GB of RAM. Databases are the well known databases of the FIMI workshop, available on the website of the worksop [4]. These databases can be divided into four categories, each category giving similar results:

- Very sparse databases: BMS-WebView1, BMS-WebView2, T10I4D100K and retail.
- Sparse databases: mushroom, BMS-POS, kosarak, Webdoc and T40I10D100K.
- Structured dense databases: connect, chess, pumsdb and pumsdb-star.
- Dense database : accidents.

In our experiments the four categories are represented the following datasets, respectively : retail, T40I10D100K, connect and accidents. Experiments with other datasets can be found on our website[1]

We give in Figure 4 the mining times and speedups w.r.t. mining times for the representative databases. The mining time does not take into account loading of input data and writing of final results to disk. This is the metric which is currently used in the state of the art ([9, 11, 12]). However the final user will be happy to learn that PLCM can output one pattern as soon as it is found. Therefore the time needed to write outputs can be covered with computations. Thanks to this, the wall clock time for a total execution of PLCM is slightly the same than the mining time.

Dense datasets: As expected, MT_Closed exhibits the best mining times and speedups for the dense datasets. MT_Closed however uses bitmaps representations and SIMD instructions to be efficient on such datasets whereas PLCM uses a generic approach. The optimizations of PLCM$_{QS}$ allow it to exihibit better performances than PLCM, and to have near identical results with MT_Closed on accidents, which is a complex real world dataset.

Sparse datasets: The optimizations in PLCM$_{QS}$ give good results on the T40I10D100K dataset, allowing it to have the best performances of all three algorithms. For the retail dataset, PLCM and PLCM$_{QS}$ have similar results, with speedups slighly better for PLCM for a high number of threads. Retail is a very sparse dataset with very few solutions and thus a small search space tree. Hence the additional computations made at each recursive iteration by PLCM$_{QS}$ does not pay off in this case. Notice that for retail no results are reported for MT_Closed, because all the runs timed out.

[1] http://membres-liglab.imag.fr/negrevergne/HPPDDM10/

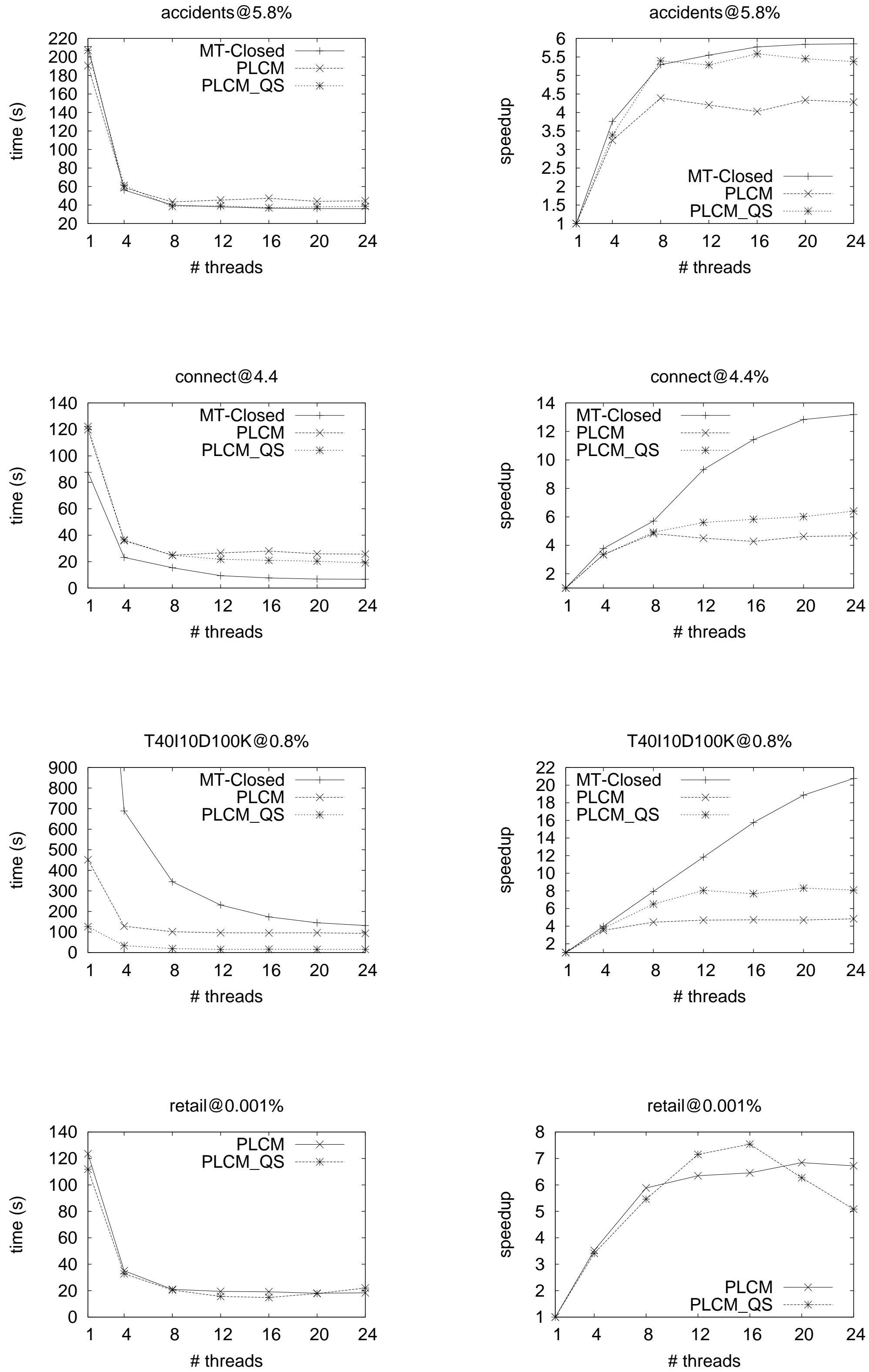

Figure 4. Comparative Experiments Results for Mining Times

Discussion: These experiments have shown that our $PLCM_{QS}$ algorithm could obtain good performances both on sparse datasets and dense datasets. It is thus a generic approach for the parallel mining of closed frequent itemsets, which is an improvement over MT_Closed which is specialized on dense datasets.

Parallel mining of closed frequent itemsets is a challenging topic. Basic operations done on each data loaded into the cores are often very simple, especially when compared to tree or graph mining. As a result, computation time for a single data is roughly similar to the time needed for loading this data. Data loading being serialized by the bus, it is very difficult to obtain good scale up properties, even more with highly optimized algorithms like LCM. Our optimizations to reduce memory usage at the expense of more computations are an effort to improve parallel performances, and our conclusions meet those found by Tatikonda et al. [12] in their study of parallel frequent tree mining.

6. CONCLUSION AND PERSPECTIVES

We have presented in this paper a simple work sharing interface, and we have used it to parallelize the well known LCM algorithm, giving the PLCM algorithm. Based on a thorough study of parallel memory accesses of PLCM, we have proposed several algorithmic optimizations aimed at reducing memory footprint and bandwidth pressure. We have integrated these optimizations into the $PLCM_{QS}$ algorithm.

A detailed experimental study has demonstrated that these optimizations allowed to have a better speedup and assured $PLCM_{QS}$ to have the lowest execution times in most cases. $PLCM_{QS}$ is thus the only generic algorithm for parallel mining of closed frequent patterns, being adapted as well for sparse and dense databases. This extends the state of the art.

We have several perspectives for future works. One of them is to find efficient heuristics for switching on or off dynamically memory-saving optimizations. In some cases, these optimizations are too costly and do not bring benefit. Another perspective is to integrate some bitmap techniques into our algorithm in order to further improve performances on dense datasets. We would also like to investigate the solutions we developped in $PLCM_{QS}$ for other algorithms similar to LCM, such as CLOATT [16] or the more general framework described in [17].

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in VLDB, 1994, pp. 487–499.

[2] N. Pasquier, Yves, Y. Bastide, R. Taouil, and L. Lakhal, "Efficient mining of association rules using closed itemset lattices," *Information Systems*, vol. 24, pp. 25–46, 1999.

[3] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in The International Conference on Management of Data, SIGMOD, 2000, pp. 1–12.

[4] B. Goethals, "Fimi repository website," http://fimi.cs.helsinki.fi/, 2003-2004.

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

[6] B. Negrevergne, A. Termier, J.-F. Méhaut, and T. Uno, "Découverte d'itemsets fréquents fermés sur architecture multicœurs," in Extraction et Gestion de Connaissances, EGC, 2010.

[7] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 6, pp. 962–969, 1996.

[8] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "Parallel algorithms for discovery of association rules," *Data Min. Knowl. Discov.*, vol. 1, no. 4, pp. 343–373, 1997.

[9] G. Buehrer, S. Parthasarathy, and Y.-K. Chen, "Adaptive parallel graph mining for cmp architectures," in ICDM'06, 2006, pp. 97–106.

[10] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in ICDM, 2002, p. 721.

[11] C. Lucchese, S. Orlando, and R. Perego, "Parallel mining of frequent closed patterns: Harnessing modern computer architectures," in ICDM, 2007, pp. 242–251.

[12] S. Tatikonda and S. Parthasarathy, "Mining tree-structured data on multicore systems," in VLDB, 2009.

[13] T. Uno, T. Asai, Y. Uchida, and H. Arimura, "An efficient algorithm for enumerating closed patterns in transaction databases," in Discovery Science, 2004, pp. 16–31.

[14] D. Gelernter, "Generative communication in Linda," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 7, no. 1, pp. 80–112, 1985.

[15] T. Uno, "Lcm ver. 3: Collaboration of array, bitmap and prefix tree for frequent itemset mining," in Open Source Data Mining Workshop on Frequent Pattern Mining Implementations, SIGKDD, 2005, pp. 77–86.

[16] H. Arimura and T. Uno, "An output-polynomial time algorithm for mining frequent closed attribute trees," in ILP, 2005, pp. 1–19.

[17] Arimura and T. Uno, "A polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence," *Algorithms and Computation*, pp. 724–737, 2009.