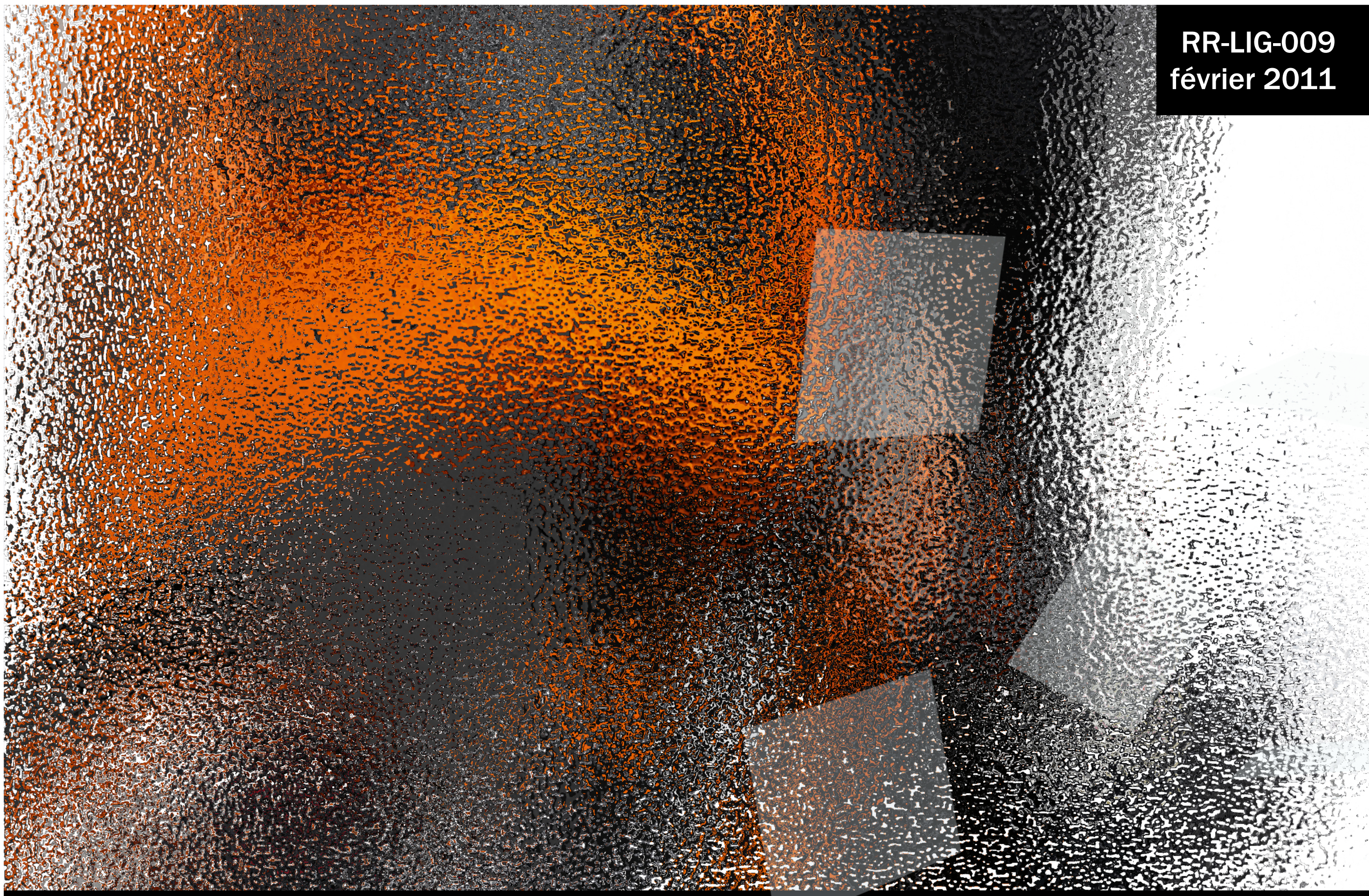
HLCM: a first experiment on parallel data mining with Haskell

ALEXANDRE TERMIER, Associate Professor, Grenoble University (UJF), France
BENJAMIN NEGREVERGNE, Drs, Grenoble University (UJF), France
SIMON MARLOW, Senior RSDE, Microsoft Research, Cambridge, UK
SATNAM SINGH, Senior Researcher, Microsoft Research, Cambridge, UK



RR-LIG-009
février 2011

http://rr.liglab.fr

ISSN 2105-0422

– DRAFT –

HLCM: a first experiment on parallel data mining with Haskell

Alexandre Termier Benjamin Négrevergne
LIG, Grenoble University
Alexandre.Termier@imag.fr
Benjamin.Negrevergne@imag.fr

Simon Marlow Satnam Singh
Microsoft Research, Cambridge
simonmar@microsoft.com
satnams@microsoft.com

Abstract

We present a parallel implementation in Haskell of the most efficient closed frequent itemset mining algorithm called LCM. We show that Haskell allows us to conveniently express the complex code of the LCM algorithm. We also present a thorough experimental study about the influence of run time system parameters on the parallel performance of our implementation.

Categories and Subject Descriptors H.2.8 [*Information Systems/Database applications*]: Data mining

General Terms Data mining, parallelism

Keywords Closed frequent pattern mining, parallel program, performance evaluation

1. Introduction

Frequent pattern mining is a major component of data mining consisting of the discovery of patterns occurring more than a given number of times in a data-set. This research originated with the analysis of market-basket data [1], and has been extended to the analysis of sequential [10], tree [2] and graph data [5]. Typical application domains include corporate data analysis, log analysis, chemistry or bioinformatics among many others. Frequent pattern mining is a highly computationally intensive task which attracts research to discover new and more efficient algorithms, or novel ways to improve existing algorithms.

With the advent of multicore computers, we could expect that pattern mining algorithm research quickly shifts to parallel algorithm, in order to exploit the power of these computers. However except some pioneering works [3], this is not yet the case. We propose two reasons to explain this situation. First, the computations in pattern mining algorithms are very irregular and unpredictable, making it difficult to design an efficient parallelization strategy. Second, pattern mining algorithms implementations are usually large and complex C programs with many specific optimizations, both high and low level. Understanding and modifying these programs is a time consuming process. Even if a good parallization strategy is applied, performance is not guaranteed, as assumptions from the sequential era such as *"store intermediate data in memory to avoid redundant computations"* that lead the design of these programs can give subpar parallel performance. [12] for example shown a 4-time run time improvement when focusing on reducing memory footprint and avoiding dynamic data structures in order to alleviate bandwidth pressure.

Functional languages allow us to write higher level code and these languages exhibit good properties for parallel programming. One of the most promising languages for parallel applications is Haskell, whose design heavily emphasises ease of program development and efficient parallelism [13].

In this paper, we describe how to use Haskell as a prototyping language for parallel frequent pattern mining algorithms. We show that it allows us to write concise and readable code, while being between 6 and 50 times slower than C++. This simplies the experimentation on different parallelism stategies and algorithmic ideas, while being fast enough to analyze real-world datasets.

Our contributions are the following:

- We implemented LCM [14], the state-of-the-art algorithm for mining closed frequent itemsets, in Haskell. For the rest of this paper, this implementation will be refered to as HLCM [1]. Our implementation is limited to 500 lines compared to the 3500 lines of the parallel C++ implementation [8], and is the first implementation to make readily understandable the most advanced features of the algorithm. The LCM algorithm and our Haskell implementation are described in Section 2.
- We parallelized our Haskell implementation with Haskell semi-explicit parallelism (i.e. `parMap`-like constructs). In Section 3, we present the parallel performance of HLCM, an through a detailed experimental study we show the influence of the RTS parameters on performance. We show that a correct parametering is necessary for parallel programs, leading at best to a 5 times speedup over default parameters.
- We also compare HLCM with the C++ implementation of LCM, and show that HLCM with correct RTS parameters takes better advantage of parallelism than this implementation.

2. HLCM: itemset mining in Haskell

2.1 Problem settings

We start with a few definitions in order to state the closed frequent itemset mining problem.

Let $\mathcal{I} = \{i_0, ..., i_n\}$ be the set of *items*. Any subset $I \subseteq \mathcal{I}$ is called an *itemset*. Input data is a set of *transactions* $\mathcal{T} = \{t_0, ..., t_m\}$, where the transactions $t_i, i \in [0..m]$ are itemsets. An itemset I occurs in a transaction t_k iff $I \subseteq t_k$. The set of occurrences of I, also called $tidlist$ of I, is the set $tidlist(I) = \{k \mid k \in [0..m], I \subseteq t_k\}$. The support of an itemset I is defined by $support(I) = |tidlist(I)|$. Lets consider a frequency threshold

[1] Our implementation will soon be available on Hackage as the `hlcm` package.

$\varepsilon \in [0..n]$. An itemset I is *frequent* iff $support(i) \geq \varepsilon$. Let $\mathcal{F}$ be the set of all frequent itemsets.

We give an example transaction database below:

Transaction id	Transaction items
t_1	[1,2,3,4,5,6]
t_2	[2,3,5]
t_3	[2,5]
t_4	[1,2,4,5,6]
t_5	[2,4]
t_6	[1,4,6]
t_7	[3,4,6]

With $\varepsilon = 3$, the frequent itemsets are:

Frequent itemset	Support	tidlist
[4]	5	$[t_1, t_4, t_5, t_6, t_7]$
[2]	5	$[t_1, t_2, t_3, t_4, t_5]$
[6]	4	$[t_1, t_4, t_6, t_7]$
[5]	4	$[t_1, t_2, t_3, t_4]$
$[4, 6]$	4	$[t_1, t_4, t_6, t_7]$
$[2, 5]$	4	$[t_1, t_2, t_3, t_4]$
[3]	3	$[t_1, t_2, t_7]$
$[2, 4]$	3	$[t_1, t_4, t_5]$
$[1, 4]$	3	$[t_1, t_4, t_6]$
$[1, 6]$	3	$[t_1, t_4, t_6]$
$[1, 4, 6]$	3	$[t_1, t_4, t_6]$

As you can see, the frequent itemsets usually contains a lot of redundant informations: for example, knowing that $[1, 6]$ is frequent with tidlist $[t_1, t_4, t_6]$ does not add any new information when one knows that $[1, 4, 6]$ is frequent *with the same tidlist*. Closed frequent itemsets get rid of this redundancy without losing information. A frequent itemset $I \in \mathcal{F}$ is *closed* if there exists no frequent itemset $I' \in \mathcal{F}$ such that $I \subset I'$ and $tidlist(I) = tidlist(I')$. Intuitively, closed frequent itemsets are the biggest itemsets (w.r.t. set inclusion) for a given tidlist. Let $\mathcal{C}$ be the set of closed frequent itemsets. In our previous example, the closed frequent itemsets are:

Closed frequent itemset	Support	tidlist
[4]	5	$[t_1, t_4, t_5, t_6, t_7]$
[2]	5	$[t_1, t_2, t_3, t_4, t_5]$
$[4, 6]$	4	$[t_1, t_4, t_6, t_7]$
$[2, 5]$	4	$[t_1, t_2, t_3, t_4]$
[3]	3	$[t_1, t_2, t_7]$
$[2, 4]$	3	$[t_1, t_4, t_5]$
$[1, 4, 6]$	3	$[t_1, t_4, t_6]$

The problem of closed frequent itemset mining is thus, given a set of transactions $\mathcal{T}$ and a frequency threshold ε, to discover all the closed frequent itemsets of $\mathcal{T}$.

2.2 The LCM algorithm

Many algorithms capable of solving the problem of mining closed frequent itemsets have been proposed, e.g. [6, 9, 15]. The FIMI'04 competition [4] crowned LCM [14] as the most efficient of these algorithms.

LCM efficiency mainly comes from a theoretical progress: LCM authors showed that it is possible to build a covering tree of all the closed frequent itemests, and that the edges of this tree can be determined efficiently during execution. The other algorithms [6, 9, 15] have to maintain a memory of all the closed frequent itemsets found, and when a new "potential" closed frequent itemset arrives they have to verify that it is not invalidated by an already found closed frequent itemset, or that it does not invalidates some already found closed frequent itemsets in the memory. With LCM there is no more need for this memory: the closed frequent itemsets can be outputted as soon as they are found. Thanks to this result, the complexity of LCM is linear in the number of closed frequent itemset to find, hence its name: ***L**inear time **C**losed itemset **M**iner*.

We give in Algorithm 1 the pseudo-code of the LCM algorithm. Note that this pseudo-code is different from the pseudo-code given by the original authors of LCM in [14]. The algorithm that we give here follows exactly the algorithm of the C code of the original authors, which implements a more efficient traversal of the search space than shown in the papers.

Algorithm 1 LCM

```
Input: Pattern $P$, Database $DB$, Item $limit$, Threshold $\varepsilon$
Output: All the closed frequent patterns of $DB$ that contain $P$
    and that contain no bigger item than $limit$.
 1: $CDB$ = project and reduce $DB$ w.r.t. $P$ and $limit$
 2: Compute frequencies of items in $CDB$
 3: $CP$ = 100% frequent items in $CDB$
 4: if $max(CP) = limit$ then
 5:     $P' = P \cup CP$
 6:     $Cand$ = frequent items of $CDB$ that are not in $CP$
 7:     Output $P'$
 8:     for all $e \in Cand, e \leq limit$ do
 9:         $LCM(P', CDB, e, \varepsilon)$
10:     end for
11: end if
```

The main function described in Algorithm 1 is called as described in Algorithm 2.

Algorithm 2 LCMmain

```
Input: Database $DB$, set of items $\mathcal{I}$, Threshold $\varepsilon$
Output: All the closed frequent patterns of $DB$
 1: Reorder items in $DB$ by descending order of frequency
 2: for all $e \in \mathcal{I}, support(e) \geq \varepsilon$ do
 3:     $LCM(\perp, DB, e, \varepsilon)$
 4: end for
```

The LCM algorithm is a tree-recursive algorithm, whose recursion structure is based on a DFS exploration of the search tree of closed frequent itemsets. Each node of the tree (i.e. each recursive call) corresponds to a closed frequent itemset of the solution or to an empty leaf with no solution. It is assumed that the items have been relabeled in descending order of frequency: item 0 is the most frequent, item 1 is the second most frequent and so on (line 1 in Algorithm 2).

In each iteration (Algorithm 1), the algorithm receives a closed frequent itemset P which is the solution of its caller, and an item $limit$ with which P must be extended. No item bigger than $limit$ are allowed in the output.

The database DB received in input is a projection of the original database restricted to the transactions containing P. It is further reduce to eliminate infrequent items and items bigger than $limit$ that are not 100 % frequent. Once these items are removed, there may be duplicate transactions, which are merged into a single transaction with a weight indicating how many original transactions it represents. The resulting database CDB is called a *conditional database*. Reducing the databases is a very important optimization, as the search tree tend to have a very large branching factor but a rather low depth. This leads to a "bottom-wide" tree with a lot of leaves. Reduced databases allow to speed-up the computations in these leaves, which speeds up considerabily the algorithm.

The actual computation is done in line 2: the conditional database is scanned in order to compute the frequency of each item. All the items that appear in each transaction are stored in CP: these are the item of CDB that always co-occur with the items of P, so they are the extension of P we are looking for. If $max(CP) > limit$, then the closed frequent itemset $P \cup CP$ does not belong to this branch of computation and the current iteration ends. Else, we output the solution which is $P \cup CP$, and we iterate recursively with the new pattern $P \cup CP$. The items used as limit are all the frequent items that are not in CP: they are not in the current closed frequent itemset, but can lead to more specific closed frequent itemsets (itemsets bigger than $P \cup CP$ whose supporting tidlist is included in the tidlist of $P \cup CP$).

2.3 HLCM: LCM in Haskell

We wanted to implement an efficient closed frequent itemset miner in Haskell, so LCM was a natural choice. Moreover, the incremental behavior of the algorithm is well adapted to the laziness of Haskell: writing

```
take N (hlcm ...)
```

will only compute the N first closed frequent itemsets of the search tree, without requesting any more computations. This would not have been possible with other algorithms such as [6, 9, 15], which would have had to compute all the closed frequent itemsets first and then return the first N.

Haskell also make very easy to parallelize the LCM algorithms with strategies. The **for all** in Algorithms 1 and 2 can easily be implemented by a `map` in Haskell. Replacing this `map` by a `parMap` (a parallel implementation of `map`) immediately parallelizes the program. We will see in Section 3 the effectiveness of this parallelization.

The most delicate part in implementing LCM in Haskell was to handle efficiently the conditional databases. These structures support many accesses and are rewritten at each iteration, so they are the key for good performances. In the original C implementation of T. Uno, he initially allocates a large memory chunk, and then manages himself this memory. The new conditional databases are not really created, instead he cleverly reuses parts of the "parent database", knowing which parts will be reused and which will not. This is possible because this implementation is purely sequential, but for parallel implementations the new conditional databases must be allocated for real. This is what did [8] in their parallel C++ implementation (PLCM), with a 2-4x performance hit with one thread when compared to the original C implementation. For HLCM, our first experiments with arrays did not gave satisfactory results. The projection/reduction step involves a lot of fiddling with the indexes of the array, for operations such as transaction sorting, transaction elimination, item elimination. We found ourselves writing array code in Haskell with `unsafe` operation which was much more difficult to write than C array code, with a poor execution speed. This was clearly not our goal.

We thus completely rewrote the database data structure as a lexicographic tree with strict values in the nodes. This structure has the advantage to perform automatically two of the most important database reduction operations: sorting transactions and eliminating duplicate transactions. In facts, it can be interesting to note that the original C implementation performs the transaction sorting with a radix sort, whose computation structure is a lexicographic tree. Using a lexicographic tree for transactions databases is thus a natural choice: we encode the data in HLCM by following the computation structure of the original C implementation. We saw a big improvement in code readability and in our comprehension of the algorithm. We also saw an important performance improvement w.r.t. our array based solution, further enhanced by adding strictness annotation in the lexicographic tree data structure.

Parallelization: As said above, we parallelized HLCM by using the `Control.Strategies` module. We wanted to see if it was possible to get improve the algorithm execution time on multicore processors, with minimal modifications in the code. We thus limited the map to replace the `map` implementing the **for all** of line 8 with either a `parMap` or a `parBuffer`. In both cases we had to use the `rdeepseq` strategy[2], `rwhnf` always gave us a near sequential behavior with only one thread working.

As reported in other parallel experimentations in Haskell [13], creating too many sparks can in some cases be detrimental to performance. We thus put a threshold on the depth of iterations where new sparks are created. For deeper iterations, the program reverts to a standard `map` in order to keep some control on the spark granularity.

3. Experiments

In this section, we investigate the performance of HLCM. We measure its execution time for different number of cores, and the speedup that could be reached by parallel execution. We also compare HLCM to the parallel C++ implementation of LCM named PLCM, and to the sequential C implemention of LCM named LCM2.5.

We use well known benchmarking databases for frequent itemset mining[3]. These databases can be divided into *sparse* databases with a small average number of items per transactions, and *dense* databases with many items per transaction. Sparse databases are represented in our experiments by `retail` and `T40I10D100K`, and dense databases are represented by `connect` and `accidents`.

3.1 Exploring parallel RTS behavior

We implemented HLCM and compiled it with a HEAD version of GHC (`ghc-6.13.20100525`). We used the recommended options for good parallel performance: `ghc -O2 -threaded -feager-blackholing`. Our test machine is a dual Intel Xeon-5520 (Gainestown) at 2.26 GHz with 24 GB of RAM and 8 cores.

The baseline performance for HLCM, using `parMap` as strategy and 8 cores (`+RTS -N8`), is given below. In this experiment, sparks are created for all depthes of the search tree.

	retail	T40I10D100K	connect	accidents
	$\varepsilon = 50$	$\varepsilon = 3000$	$\varepsilon = 15000$	$\varepsilon = 95000$
Time (s)	18.2s	19.5s	231.5s	79.2s
Speedup	3.3	3.1	2.6	2.5

There is a speedup for all the datasets, which is encouraging. However, on a machine with 8 cores one would expect speedups better than 3. Note that the loading of data is sequential in HLCM, but the support threshold ε has been chosen low enough for loading time to be small w.r.t. computation time.

We now study the influence of RTS parameters and parallel strategies on performance.

Heap size: Data mining programs such as HLCM take as input datasets of moderate size (hundreds of MB at most). However the exploration of these datasets will lead to the construction of a large number of intermediary structures in memory, especially when several threads explore different branches of the search tree in parallel. Allocating more memory in the heap with the `+RTS -H` option reduces GC time, which could improve run time. We report in Figure 1 the speedup over the baseline obtained when giving more heap size to HLCM with `+RTS -H`. All other parameters remain unchanged.

For the dense datasets, which are very complex and lead to many closed frequent itemsets, giving more heap allow impressive speedups: execution on connect gets 5 times faster than baseline ! The sparse dataset T40I10D100K, which is of moderate complexity, is unaffected by heap size. However, the retail dataset sees a catastrophic degrade in performance with more heap size. For this dataset, reducing the heap size to `200M` is the only way to get slighly better results than the baseline. This dataset differs from

[2] `rnf` with the old version of `Control.Strategies`

[3] Available at `http://fimi.cs.helsinki.fi/data/`

T40I10D100K in that it is a *very sparse* dataset, with very few closed frequent itemsets. The needed memory is thus much lower than for processing the other datasets. Figure 2 shows the percentage of time spent doing GC in the above experiment.

For the retail dataset only, GC percentage skyrockets with more heap size. The most probable reason is that retail's data requires little memory and with the correct settings stays most of the time in cache. Increasing the heap size has a detrimental effect on locality, which is most important on datasets like retail which can have a very good locality.

We also investigate the heap size influence w.r.t. the number of threads. Figure 3 shows for the accidents dataset the speedup versus the baseline run for executions with 1, 2, 4 and 8 threads. The "baseline" is the run without any RTS parameter other than `-N` for each of these number of threads.

For 1 thread, as soon as there is 1G of memory the best run time is reached. Then the more the threads, the more the heap size needed to reach the best level of performance. It is especially interesting to note that the more the threads, the more the heap size has an impact on performance. For 2 threads, with the best heap size the speedup w.r.t. default setting is 2.6. But for 4 threads it reaches a speedup of 3, and for 8 threads the speedup peaks at 3.4.

It is such of uttermost importance to fine tune the heap size for parallel Haskell programs, with the `+RTS -H` option. The best heap sizes for each dataset are reported below, in the case of 8 threads:

	retail	T40I10D100K	connect	accidents
Best -H	-H200M	-H5G	-H10G	-H10G
Time (s)	17s	15.5s	42.7s	23.4s

GC load balancing: The GHC RTS allows to control the amount of load balancing done by the parallel GC, with the `-qb[gen]` option. `-qb` desactivates totally GC load balancing, while `-qb1` allows load balancing for generation 1 and `-qb0` allows load balancing for generations 0 and 1 (in the case of 2 generations, which is the default). These parameters can have an influence on execution on runtime, as shown in [7]. We did new experiments, where the baseline is now the execution with 8 cores and the best heap settings.

The results are reported below. For each parameter, we report the average run time on 5 executions, and the difference with the best heap baseline.

	retail	T40I10D100K	connect	accidents
`-qb`	17.4s (+2.4%)	16.5s (+6.4%)	44.2s (+3.5%)	23.4s (0%)
`-qb0`	17.3s (+1.8%)	14.5s (-6.6%)	44.1s (+3.3%)	23.9s (+2%)
`-qb1`	17.5s (+3%)	15.6s (0%)	43.3s (+1.4%)	23.5s (+0.5%)

Here this option does not offer a significant difference, except for the T40I10D100K dataset. In most cases, the best setting is `-qb` (no load balancing) or `-qb1` (load balancing restricted to generation 1). This is consistant with the tree-recursive nature of HLCM: too much load balancing can move subtrees on a processor different from their root, leading to numerous cache misses when the databases have to be reloaded on the new processor. This is however the opposite for T40I10D100K and retail, which prefers the full load balancing offered by `-qb0`. This could mean that the search tree for these sparse datasets is largely unbalanced, and that the cache miss price of load balancing is compensated by keeping all processors busy.

The best GC load balancing parameters and new times for each datasets are summed up below:

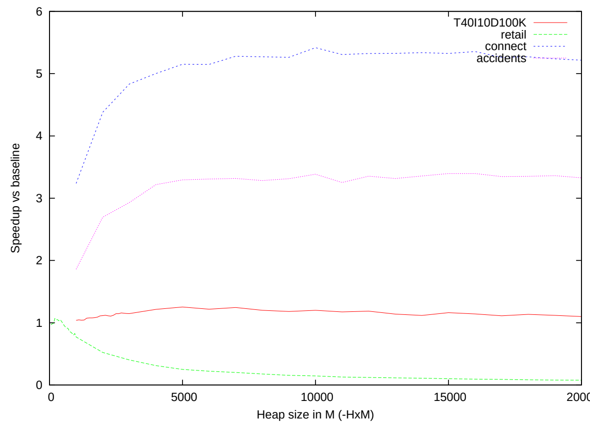


Figure 1. Speedup vs baseline w.r.t. heap size

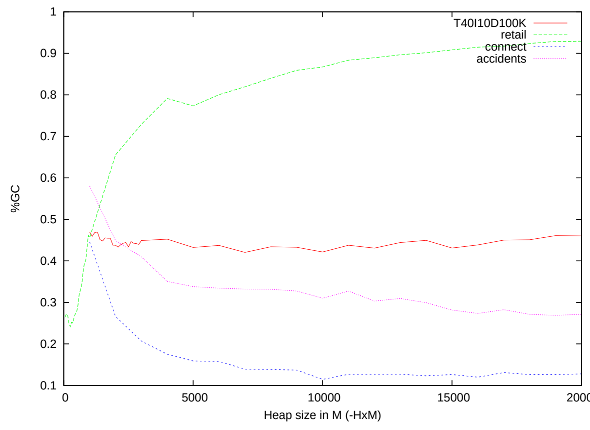


Figure 2. % GC w.r.t. heap size

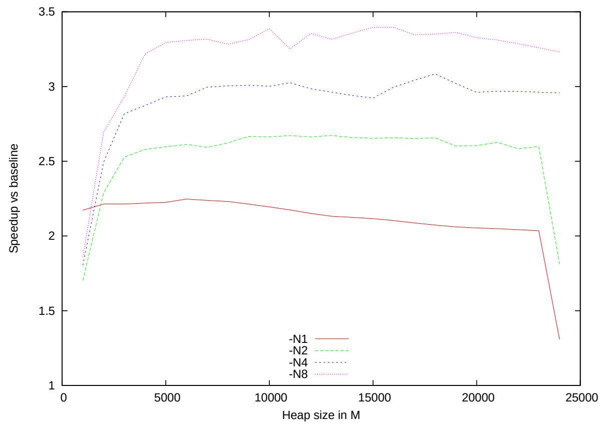


Figure 3. Impact of heap size w.r.t. nb of threads, accidents dataset

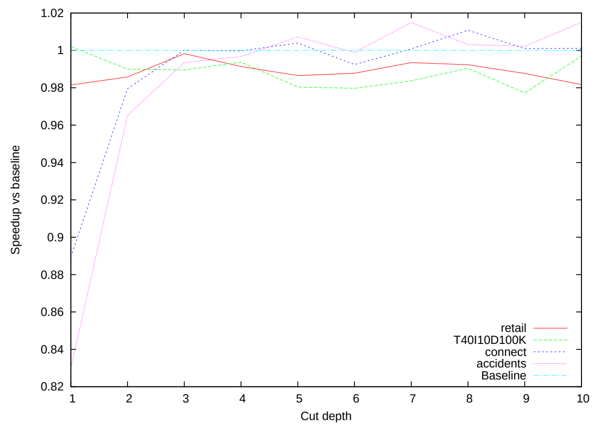


Figure 4. Speedup vs baseline w.r.t. depth for switching to sequential

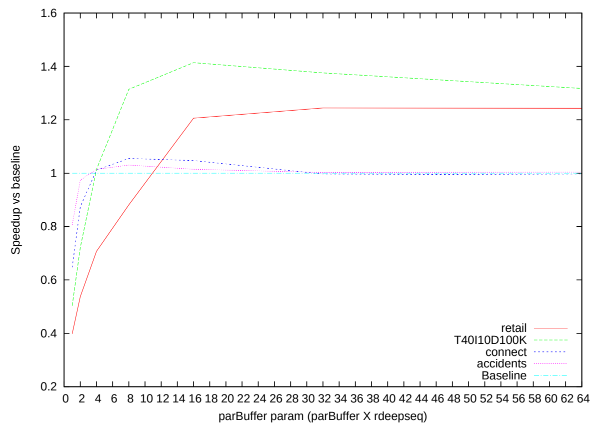


Figure 5. Speedup vs baseline w.r.t. parBuffer parameter

	retail	T40I10D100K	connect	accidents
-qb	-qb0	-qb0	-qb1	-qb
Time (s)	17.3s	14.5s	43.3s	23.4s

Controlling granularity via depth: The deeper the iteration in the search tree, the simpler the computations. The computations can become so simple and so numerous that the overheads of spark creation for each of them outweight their benefit. As shown in Section 2, we have a threshold that prevents the creation of sparks for iteration below a certain depth. The speedup vs baseline w.r.t. depth for preventing spark creation is shown in Figure 4.

For the dense datasets connect and accidents, creating more sparks is beneficial until depth 3-4, without negative impact for higher depthes. These datasets are complex and have many closed frequent itemsets, so there is always enough work to keep the sparks busy. For the sparse datasets however depth of spark creation seem to have a much weaker influence.

parMap vs parBuffer: Next is the choice of strategy in `Control.Strategies`. The two strategies of choice are `parMap rdeepseq` and `parBuffer N rdeepseq`. The difference is that `parMap` eagerly sparks every elements in the argument list, while `parBuffer N` only sparks ahead N elements. We vary in Figure 5 the parameter for parBuffer in Algorithm 2, and report the speedup between `parBuffer N` and the baseline which used `parMap`. We chose Algorithm 2 because it is at this point that the biggest branching factor exists, and that `parBuffer` is more likely to have effect. For Algorithm 1 we took a more conservative approach, using `parBuffer 2 rdeepseq`.

It appears clearly that when correctly parametered, `parBuffer` can improve performances over `parMap`. Dense dataset tend to prefer relatively few spark creations ahead (4 for accidents, 8 for connect), while sparse dataset prefer many spark creations ahead (more than 16). This is consistant with the previous observation on depth threshold: there is less work in branches of the search tree for sparse datasets, so sparking many branches allows to reduce a too fine granularity. Oppositely, there is a lot of work in branches of the search tree for dense datasets, so fewer of these branches can be sparked together.

The results with the previously found RTS parameters and the best parBuffer parameters values are reported below, along with their improvement over the previous best times. The improvement is especially important for sparse datasets.

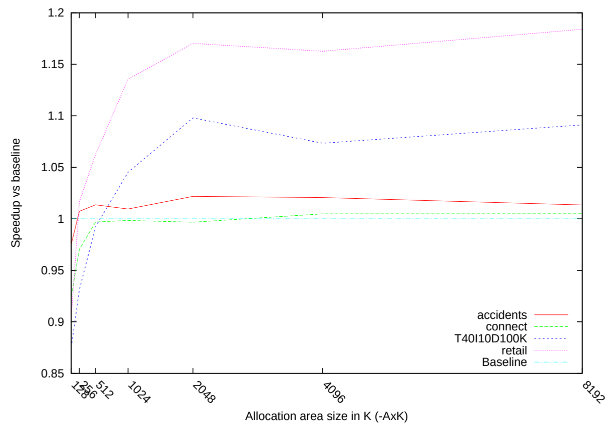


Figure 6. Speedup vs baseline w.r.t. allocation area size

	retail	T40I10D100K	connect	accidents
parBuffer N	64	16	8	4
Time (s)	13.9s (-19.5%)	10.5s (-27%)	41s (-5%)	23.1s (-1%)

Allocation area: We have found a good set of parameters, both inside our program with parBuffer and cut depth, and for the RTS. To conclude our parameter space exploration, we investigate one last RTS parameter, the GC allocation area (`+RTS -A`). This parameter directly influences the cache usage behavior of the algorithm, so it might give interesting results for a data-intensive program such as HLCM. Figure 6 shows the impact of varying the allocation area size. All the other parameters are set to give the best results as found before. We also added the `-F4` parameter in the RTS for this experiment, in order to reduce the number of garbage collections, which can be usefull for small values of `-A`.

Too small allocation area sizes such as 128K is detrimental to all datasets: they fill up too fast, and time is lost when their contents is moved to old generation. Accidents, the densest dataset, which perform a lot of work on large and complex search trees, give its best performance with mid-sized allocation area, around 2M. On the processors of the test machine, the L3 has 8M, with 4 cores on a single die. So 2M corresponds well to the share of one thread. There must be little data sharing between the threads, so bigger allocation area sizes induce too much competition on the cache on worse performances. However for the sparse datasets, which have simpler search trees, it seems that there is more data sharing, and large allocation area of 8M give excellent results.

The best results are reported below:

	retail	T40I10D100K	connect	accidents
-A	8M	2M	8M	2M
t (s)	11.7s (-15%)	9.6s (-9%)	40.8s (-0.5%)	22.6s (-2%)

Last, we also run HLCM with one thread and best RTS parameters for heap, in order to give final parallel speedups for 8 threads:

	retail	T40I10D100K	connect	accidents
HLCM -N1	59.2s	46.6s	259.5s	91.3s
Final speedup	5	4.8	6.3	4

Even if the single thread run time with better parameters improved, the parallel run time improved even more, leading to much better speedups. Our worse speedup is now 4, and for the complex connect dataset we hit 6.3, which is an excellent result on a 8-core computer.

Discussion: These experiments and the improve in execution time that we got show that correctly setting the RTS parameters for the GC is mandatory for parallel performance. It is interesting to note the different behavior of dense and sparse datasets. Traditional C/C++ implementations have difficulties to be efficient for both. The advantage of Haskell is that we can give different RTS settings for each type of dataset, and easily get performance improvements.

Determining the correct RTS parameters need a lot of experiments due to the size of the parameter space. However, it is a task than could be done automatically, for example with a genetic algorithm [11]. This is not the case for C/C++ parallel programs.

3.2 Comparison with existing implementations

To conclude these experiments, we compare the run time of HLCM with the original C implementation of T. Uno (sequential) [14], and with the parallel C++ implementation PLCM [8]. Note that as we stated in Section 2, the sequential C implementation can avoid a lot of database allocations because it does not have to care for parallel execution. Its run times are given for reference only, however for fairness we only compare HLCM with PLCM. Both have the same allocation policy.

	retail	T40I10D100K	connect	accidents
lcm2.5 (1t)	0.4s	1.2s	2.4s	6s
plcm (1t)	1.7s	2.2s	5.1s	8s
plcm (8t)	0.7s	1.1s	1.4s	3.4s
HLCM/plcm (1t)	35	21.4	50.3	11.4
HLCM/plcm (8t)	15.4	9	29	6.6

These results show that even if HLCM still has a long way to go before beating the C++ implementation, it can achieve competitive results, especially for parallel execution. The HLCM/PLCM run time ratio can be as low as 6.6 for the accidents dataset. This is promising, and the ease of writing and parallelizing HLCM is without comparison with PLCM, which is a long and complex C++ code with hand written efficient data structures for itemsets based on arrays, and a hand written work sharing mechanism implemented with PThreads.

4. Conclusion

We presented HLCM, an efficient Haskell implementation of the LCM algorithm for mining closed frequent itemsets. Through detailed experiments, we showed the importance of using RTS parameters to control the heap size and about the need to influence garbage collection frequency in order to get good parallel performance. Our experiments also showed that even if our program could not beat a C++ implementation, it could stay in a 6x-30x slower range for parallel execution, which is enough to use it for analyzing real world datasets.

As a perspective, we would like to extend our work on HLCM to other pattern mining algorithms dedicated to more complex structures such as sequences, trees or graphs. We also plan to continue improving HLCM in order to provide it as a new benchmark for the `nofib` test suite. We think that this program is a good example of a real world parallel application, and depending on the dataset it can exercise different aspects of the parallel RTS of Haskell. Another interesting perspective would be to develop a run-time auto-tuning system that could monitor the behaviour of the heap and dynamically decide between reducing number of garbage collections and preserving a good locality.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, pages 487–499, 1994.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SDM*, 2002.

[3] G. Buehrer, S. Parthasarathy, and Y.-K. Chen. Adaptive parallel graph mining for cmp architectures. In *ICDM*, pages 97–106, 2006.

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

[5] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.

[6] C. Lucchese, S. Orlando, and R. Perego. Dci closed: A fast and memory efficient algorithm to mine frequent closed itemsets. In *FIMI*, 2004.

[7] S. Marlow, S. L. P. Jones, and S. Singh. Runtime support for multicore haskell. In *ICFP*, pages 65–78, 2009.

[8] B. Négrevergne, J.-F. Méhaut, A. Termier, and T. Uno. Découverte d'itemsets fréquents fermés sur architecture multicoeurs. In *EGC*, pages 465–470, 2010.

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

[10] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *ICDE*, pages 215–224, 2001.

[11] D. Stewart. Evolving faster haskell programs (now with llvm!), 2010. `http://donsbot.wordpress.com/2010/03/01/evolving-faster-haskell-programs-now-with-llvm/`.

[12] S. Tatikonda and S. Parthasarathy. Mining tree-structured data on multicore systems. *PVLDB*, 2(1):694–705, 2009.

[13] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.

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

[15] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SDM*, 2002.