Accelerating Dynamic Itemset Counting on Intel Many-core Systems

Mikhail Zymbler
South Ural State University, Chelyabinsk, Russia
mzym@susu.ru

***Abstract*—The paper presents a parallel implementation of a Dynamic Itemset Counting (DIC) algorithm for many-core systems, where DIC is a variation of the classical Apriori algorithm. We propose a bit-based internal layout for transactions and itemsets with the assumption that such a representation of the transaction database fits in main memory. This technique reduces the memory space for storing the transaction database and also simplifies support counting and candidate itemsets generation via logical bitwise operations. Implementation uses OpenMP technology and thread-level parallelism. Experimental evaluation on the platforms of Intel Xeon CPU and Intel Xeon Phi coprocessor with large synthetic database showed good performance and scalability of the proposed algorithm.**

***Index Terms*—frequent itemset mining, dynamic itemset counting, bitmap, OpenMP, many-core, Intel Xeon Phi**

Introduction

Association rule mining is one of the important problems in data mining [1]. The task is to discover the strong associations among the items from a transaction database such that the presence of one item in a transaction implies the presence of another. Association rule mining is decomposed into two subtasks [1]. The first one is to find all frequent itemsets that consist of items which often occur together in transactions. The second one is to generate all the association rules from the frequent itemsets found.

In this paper, we address the task of frequent itemset mining which can be formally described as follows. Let $\mathcal{I} = (i_1, \ldots, i_m)$ be a set of literals, called *items*. Let $\mathcal{D} = (T_1, \ldots, T_n)$ be a database of *transactions*, where each transaction $T_i \subseteq \mathcal{I}$ consists of a set of items (*itemset*). An itemset that contains k items is called a k-itemset. The *support* of an itemset $I \subseteq \mathcal{I}$ denotes the percentage of transactions in $\mathcal{D}$ that contain the itemset I. If support of an itemset $I \subseteq \mathcal{I}$ satisfies the user-specified minimum support threshold (called $minsup$) then I is *frequent itemset*. Let the set of frequent k-itemsets be denoted by $\mathcal{L}_k$ and $\mathcal{L} = \cup_{k=1}^{k_{max}} \mathcal{L}_k$ denotes a set of all frequent itemsets, where k_{max} is number of items in the longest frequent itemset. Given the transaction database $\mathcal{D}$ and minimum support threshold $minsup$ the goal of frequent itemset mining is to find the set of all frequent itemsets $\mathcal{L}$.

There is a wide spectrum of algorithms for frequent itemset mining and none of them outperforms all others for all possible transaction databases and values of $minsup$ threshold [7]. *Apriori* [1] is one of the most popular itemset mining algorithms for which many refinements and parallel implementations for various platforms were proposed. Dynamic Itemset Counting (DIC) [2] is a variation of Apriori, which tries to reduce number of passes made over a transaction database while keeping the number of itemsets counted in a pass relatively low. Despite the fact that DIC has good potential of parallelization [2] it still has not been implemented for modern many-core CPU and accelerators, to the best of our knowledge.

In this paper we propose parallel implementation of the DIC algorithm for Intel Xeon and Intel Xeon Phi (Knights Landing) many-core platforms. Intel Xeon Phi device is an x86 many-core coprocessor of 61 cores, connected by a high-performance on-die bidirectional interconnect where each core supports 4× hyperthreading and contains 512-bit wide vector processor unit. Knights Landing [13] is a second generation MIC (Many Integrated Core) architecture product from Intel. As opposed to predecessor it is an independent (bootable) device, which runs applications only in native mode.

We suggest a bit-based internal layout for transactions and itemsets assuming that such a representation of a transaction database fits in main memory. This technique has a few major merits. It reduces memory space of storing the transaction database and simplifies support counting and generation of candidate (potentially frequent) itemsets via logical bitwise operations. We parallelize the algorithm through OpenMP technology and thread-level parallelism. We conduct experiments on large synthetic database to evaluate performance and scalability of our algorithm.

The rest of the paper is organized as follows. Section I provides a brief description of an original DIC algorithm. The proposed parallel algorithm is presented in section II. In section III related work is discussed. The results of experimental evaluation of the algorithm are described in section IV. The conclusion contains summarizing remarks and directions for future research.

I. Serial DIC Algorithm

Dynamic Itemset Counting (DIC) [2] is a variation of the most well-known Apriori algorithm [1]. Apriori is an iterative, level-wise algorithm, which uses a bottom-up search. At the

This work was financially supported by the Russian Foundation for Basic Research (grant No. 17-07-00463), by Act 211 Government of the Russian Federation (contract No. 02.A03.21.0011) and by the Ministry of education and science of Russian Federation (government order 2.7905.2017).

first pass over transaction database it processes 1-itemsets and finds $\mathcal{L}_1$ set. A subsequent pass k consists of two steps, namely candidate generation and pruning. At the *candidate generation* step Apriori combines elements of $\mathcal{L}_{k-1}$ set to form candidate (potentially frequent) k-itemsets. At the *pruning* step it gets rid of infrequent candidates using the *a priori* principle, which states that any infrequent $(k-1)$-itemset cannot be a subset of a frequent k-itemset. Apriori counts support of candidates which have not been pruned and proceeds with such passes so forth until no candidates remain after pruning.

Algorithm 1. DIC(in $\mathcal{D}$, in $minsup$, in M, out $\mathcal{L}$)

```
                                          ▷ Initialize sets of itemsets
 2: SolidBox ← ∅; SolidCircle ← ∅; DashedBox ← ∅
    DashedCircle ← $\mathcal{I}$
 4: while DashedCircle ∪ DashedBox ≠ ∅ do
                                   ▷ Scan database and rewind if necessary
 6:     Read($\mathcal{D}, M, Chunk$)
        if EOF($\mathcal{D}$) then
 8:         Rewind($\mathcal{D}$)
        for all $T \in Chunk$ do
10:                                       ▷ Count support of itemsets
            for all $I \in$ DashedCircle ∪ DashedBox do
12:             if $I \subseteq T$ then
                    support($I$) ← support($I$) + 1
14:                                       ▷ Generate candidate itemsets
            for all $I \in$ DashedCircle do
16:             if support($I$) ≥ $minsup$ then
                    MoveItemset($I$, DashedBox)
18:                 for all $i \in \mathcal{I}$ do
                        $C \leftarrow I \cup i$
20:                     if $\forall s \subseteq C\ s \in$ SolidBox∪DashedBox then
                            MoveItemset($C$, DashedCircle)
22:                        ▷ Check full pass completion for itemsets
            for all $I \in$ DashedCircle do
24:             if IsPassCompleted($I$) then
                    MoveItemset($I$, DashedBox)
26:         for all $I \in$ DashedBox do
                if IsPassCompleted($I$) then
28:                 MoveItemset($I$, SolidBox)
    $\mathcal{L}$ ← SolidBox
```

The DIC algorithm tries to reduce the number of passes made over the transaction database while keeping the number of itemsets counted in a pass relatively low. Alg. 1 depicts pseudo-code of the DIC algorithm. DIC processes database with stops at equal-length intervals between transactions (parameter M of the algorithm). At the end of the transaction database it is necessary to rewind to its beginning.

DIC maintains four sets of itemsets, namely *Dashed Circle*, *Dashed Box*, *Solid Circle* and *Solid Box*. Itemsets in the "dashed" sets are subjects for support counting while itemsets in the "solid" sets do not need to be counted. "Circles" contain infrequent itemsets while "boxes" contain frequent itemsets.

Thus, *Dashed Circle* and *Dashed Box* contain itemsets that are suspected infrequent and are suspected frequent respectively while *Solid Circle* and *Solid Box* contain itemsets that are confirmed infrequent and are confirmed frequent respectively. At start *Dashed Box*, *Solid Circle* and *Solid Box* are assumed to be empty and *Dashed Circle* contains all the 1-itemsets.

Before the stop, DIC counts support of itemsets from "dashed" sets for each transaction. At any stop DIC performs as follows. Itemsets whose support exceeds $minsup$ are moved from *Dashed Circle* to *Dashed Box*. New itemsets are added into *Dashed Circle*, they are immediate supersets of those itemsets from *Dashed Box* with all of its subsets from "box" lists. Itemsets that have completed one full pass over the transaction database are moved from the "dashed" set to "solid" set. DIC proceeds if any itemset in "dashed" sets remains.

II. Parallel DIC algorithm

A. Internal Data Layout

In this work we suggest *direct bit representation* for both transactions and itemsets. For a transaction $T \subseteq \mathcal{D}$ (for an itemset $I \subseteq \mathcal{I}$, respectively) this means that it is represented by a word where each p-th bit is set to one if an item $i_p \in T$ ($i_p \in I$, respectively) and all other bits are set to zero. The word's length W in bytes depends on system environment and it is calculated as $W = \lceil \frac{m}{sizeof(\texttt{byte})} \rceil$. In our implementation we use C++ and `unsigned long long int` data type, so we have $W = 8$ and $m = 64$. This could be extended through an open-source library for arbitrary precision arithmetic, for instance, GNU Bignum Library[1].

Let us denote by $BitMask$ a function that returns direct bit representation of a given itemset or transaction as a word, i.e. $BitMask : \mathcal{I} \to \mathbb{Z}_+$. Then direct bit representation of transaction database $\mathcal{D}$ is an n-element array $\mathcal{B}$, where $\forall j,\ 1 \leq j \leq n\ \mathcal{B}[j] = BitMask[T_j]$.

Direct bit representation has several major merits. It often requires less space than byte-based representation for dense transaction database with long transactions. In fact, $\mathcal{B}$ requires $n \cdot W$ bytes to store and allows $\mathcal{B}$ to fit in main memory. For instance, *netflix*[2], one of the most referenced datasets, contains $n = 17,771$ transactions consisting of $m = 480,189$ distinct items. Hence, direct bit representation of the *netflix* dataset takes about 1 Gb. Thus, in what follows we assume that $\mathcal{B}$ has been preliminary produced from $\mathcal{D}$ and it is available in main memory.

Direct bit representation simplifies support counting as well. The fact of $I \subseteq T$ can be checked by the predicate with one logical bitwise operation, that is $BitMask(I)$ AND $BitMask(T) = BitMask(I)$.

Thereby, we implement an itemset as a record structure with the following basic fields, namely $mask$ to provide direct bit representation, k as number of items in the itemset, $stop$ as counter to determine when full pass for the given itemset is completed, and $supp$ to store support count.

[1]The GNU Multiple Precision Arithmetic Library
[2]http://www.netflixprize.com

To implement a set of itemsets, we use vector, which represents an array of elements belonging to the same type and provides random access to its elements with an ability to automatically resize when appending elements. Such a data structure is implemented in C++ Standard Template Library as a class with iterator and methods for inserting an element and removing an element with complexity of $O(1)$ and $O(s)$ respectively, where s is the current size of a vector.

To reduce costs of moving elements across vectors, we establish a *DASHED* vector for "dashed box" and "dashed circle" itemsets and a *SOLID* vector for "solid box" and "solid circle" itemsets and provide the itemset's record structure with fig field to indicate an appropriate set the given itemset belongs to.

B. Parallelization of the Algorithm

The proposed parallel version of DIC algorithm is presented in Alg. 2 and basic sub-algorithms are depicted in Alg. 3–5.

Algorithm 2. `ParalDIC(in` $\mathcal{B}$, `in` $minsup$, `in` M, `out` $\mathcal{L}$)

```
                                              ▷ Initialize sets of itemsets
 2: SOLID.$init()$; DASHED.$init()$
    $k \leftarrow 1$
 4: for all $i \in 0..m-1$ do
        $I.fig \leftarrow$ NIL; $I.bitmask \leftarrow 0$
 6:     $I.mask \leftarrow$ SetBit($I.mask, i$)
        $I.stop \leftarrow 0$; $I.supp \leftarrow 0$; $I.k \leftarrow k$
 8:     SOLID.$push\_back(I)$
    $stop_{max} \leftarrow \lceil \frac{n}{M} \rceil$; $stop \leftarrow 0$
10: FirstPass(SOLID, DASHED)
    while not DASHED.$empty()$ do
12:                              ▷ Scan database and rewind if necessary
        $stop \leftarrow stop + 1$
14:     if $stop > stop_{max}$ then
            $stop \leftarrow 1$
16:     $first \leftarrow (stop - 1) \cdot M$; $last \leftarrow stop \cdot M - 1$
        $k \leftarrow k + 1$
18:     CountSupport(DASHED)
        CutDashedCircle(DASHED)
20:     GenCandidates(DASHED)
        CheckFullPass(DASHED)
22: $\mathcal{L} \leftarrow \{I \in$ SOLID$, I.fig =$ BOX$\}$
```

We enhance the classical DIC algorithm by adding two more stages, namely *FirstPass* and *CutDashedCircle* where each of them is aimed to reducing the number of itemsets to perform support counting of.

We parallelize the following stages of the algorithm, namely support counting (cf. Alg. 3), reduction of *Dashed Circle* set (cf. Alg. 4) and checking full pass completion for itemsets (cf. Alg. 5) through OpenMP technology and thread-level parallelism.

In the classical DIC algorithm, the *Dashed Circle* set is initialized by all the 1-itemsets (cf. Alg. 1, line 3). In contrast with classical DIC, we use the technique of full first pass [4]. This means that we initially perform one full pass over $\mathcal{D}$ to find $\mathcal{L}_1$, the set of frequent 1-itemsets (this done similarly to Alg. 3). Then candidate 2-itemsets are computed from $\mathcal{L}_1$ through the Apriori join procedure [1]. This done via logical bitwise `OR` operation on each pair of frequent 1-itemsets and candidates are inserted in the *Dashed Circle* set. This technique helps to reduce cardinality of the *Dashed Circle* set in further computations because infrequent 1-itemsets and their supersets have been pruned according to the *a priori* principle.

Algorithm 3. `CountSupport(in out` $DASHED$)

```
    if DASHED.$size() \geq num\_of\_threads$ then
 2:     #pragma omp parallel for
        for all $I \in$ DASHED do
 4:         $I.stop \leftarrow I.stop + 1$
            for all $T \in \mathcal{B}[first]$ .. $\mathcal{B}[last]$ do
 6:             if $I.mask$ AND $T = I.mask$ then
                    $I.supp \leftarrow I.supp + 1$
 8: else
        omp_set_nested(true)
10:     #pragma omp parallel for
         num_threads(DASHED.$size()$)
12:     for all $I \in$ DASHED do
            $I.stop \leftarrow I.stop + 1$
14:         #pragma omp parallel for reduction(+:$I.supp$)
             num_threads($\lceil \frac{num\_of\_threads}{\text{DASHED}.size()} \rceil$)
16:         for all $T \in \mathcal{B}[first]$ .. $\mathcal{B}[last]$ do
                if $I.mask$ AND $T = I.mask$ then
18:                 $I.supp \leftarrow I.supp + 1$
```

In the original algorithm support counting is performed through two nested loops (cf. Alg. 1, lines 9–13) where the outer loop takes transactions and the inner loop takes the "dashed" itemsets. As opposed to the classical DIC algorithm we change the order of these loops to parallelize outer loop through omp parallel for pragma (cf. Alg. 3). This shuffle avoids data races when threads process different transactions but need to change support count of the same itemsets simultaneously.

Additionally, our algorithm balances the load of threads depending on the current total number of elements in both *Dashed Circle* and *Dashed Box* sets. If the number of available threads does not exceed current total number of "dashed" itemsets, we parallelize the outer loop (along itemsets) using all the threads. Otherwise, we enable nested parallelism and parallelize the outer loop using a number of threads equal to the current total number of "dashed" itemsets. Then we parallelize the inner loop (along transactions) so that each outer thread forks an equal-sized set of descendant threads where descendants perform counting through reduction of summing operation. This balancing technique allows to processing data effectively in the final stage of counting when the number of candidate itemsets tends to zero and increases overall performance of the algorithm.

After the support counting, in addition to moving appropriate itemsets from *Dashed Circle* set to *Dashed Box* set as in classical DIC (cf. Alg. 1, line 17), we reduce *Dashed Circle* set pruning clearly infrequent itemsets as follows [9]. We compute an itemset's highest possible support by adding

Algorithm 4. `CutDashedCircle(in out` $DASHED$`)`

```
    #pragma omp parallel for
 2: for all $I \in$ DASHED and $I.fig =$ CIRCLE do
        if $I.supp \geq minsup$ then
 4:         ▷ Move appropriate itemsets to Dashed Box set
            $I.fig \leftarrow$ BOX
 6:     else
                                ▷ Prune clearly infrequent itemset
 8:         $supp_{max} \leftarrow I.supp + M \cdot (stop_{max} - I.stop)$
            if $supp_{max} < minsup$ then
10:             $I.fig \leftarrow$ NIL
                            ▷ Prune supersets of infrequent itemset
12:             for all $J \in$ DASHED and $J.fig =$ CIRCLE do
                    if $I.mask$ AND $J.mask = I.mask$ then
14:                     $J.fig \leftarrow$ NIL
    DASHED.$erase(\forall I, I.fig =$ NIL$)$
```

its current support to the number of transactions have not been processed yet (cf. Alg. 4). If the value of the itemset's highest possible support is less than *minsup* threshold, then the itemset is pruned and after that we prune all its supersets according to the *a priori* principle.

After the reduction of *Dashed Circle* set we generate afresh itemsets to be inserted in that set performing Apriori join procedure [1] via logical bitwise `OR` operation between all the itemsets marked as "boxes".

Algorithm 5. `CheckFullPass(in out` $DASHED$`)`

```
    #pragma omp parallel for
 2: for all $I \in$ DASHED do
        if $I.stop = stop_{max}$ then
 4:         if $I.supp \geq minsup$ then
                $I.fig \leftarrow$ BOX
 6:         SOLID.$push\_back(I)$
            $I.fig \leftarrow$ NIL
 8: DASHED.$erase(\forall I, I.fig =$ NIL$)$
```

Finally, for all itemsets in the *Dashed Circle* set we check if an itemset has been counted through all the transactions and if yes, we make the itemset "solid" and stop counting it (cf. Alg. 5). This activity is also parallelized along itemsets through omp parallel for pragma.

In the end *DASHED* vector contains "box" itemsets as a result of the algorithm.

III. Related Work

The Original DIC algorithm was presented by Brin et al. in [2], where the authors briefly discuss a way to parallelize DIC using the distribution of the transaction database among the nodes so that each node counts all the itemsets for its own data segment. The authors noticed that it is unnecessary to perform synchronization and load balancing in parallel version of DIC.

Paranjape-Voditel et al. proposed *DIC-OPT* [10], a parallel version of DIC for distributed memory systems. The key idea is that each node sends messages to other nodes after every M transactions have been read regarding the counts of potentially frequent itemsets. This initiates the early counting of the itemsets on other nodes without waiting for synchronization with other nodes. Authors carried out experiments on up to 12 nodes where their implementation showed sub-linear speedup.

Cheung et al. suggested *APM* [4], a DIC-based parallel algorithm for SMP systems. APM is an adaptive parallel mining algorithm, where all CPUs generate candidates dynamically and count itemset supports independently without synchronization. The transaction database is partitioned across CPUs with a highly homogeneous itemset distributions. This technique addresses to the problem of a large number of candidates because of the low homogeneous itemset distribution in most cases. The experiments on the Sun Enterprise 4000 server with up to 12 nodes showed that APM outperforms Apriori-like parallel algorithms. However, APM's speedup gradually drops down to 4 when the number of nodes is grater than 4. This is because APM suffers from the SMP's inherent problem of I/O contention when the number of nodes is large.

Schlegel et al. proposed *mcEclat* [12], a parallel version of the well-known mining algorithm Eclat [14] for the Intel Xeon Phi coprocessor. mcEclat converts a dataset being mined into a set of tid-bitmaps, which are repeatedly intersected to obtain the frequent itemsets. *Tid-bitmap* maps the IDs of transactions in which an itemset occurs to bits in a bitmap at certain positions. For instance, if the itemset i exists in 4-th and 7-th transactions then the respective bits of i's tid-bitmap are set to one while all its other bits are set to zero. Tid-bitmaps are intersected via logical bitwise `AND` operation and then support of an itemset is obtained by counting the one bits in its respective tid-bitmap. Experiments showed up to $100\times$ speedup of mcEclat on the Intel Xeon Phi. However, the algorithm's performance on the Intel Xeon Phi coprocessor is similar or slightly worse (for smaller values of $minsup$) than on system with two Intel Xeon CPUs when the maximum number of threads is employed on both systems. The reason is that mcEclat does not fully exploit the Intel Xeon Phi's powerful vector processing capabilities.

Kumar et al. presented *Bitwise DIC* [9], a serial version of the DIC algorithm based upon tid-bitmap technique mentioned above. Bitwise DIC outperforms the original DIC. Unfortunately, the authors poorly supported their study by experiments and discussion of the results (only five runs of the algorithms on one dataset with $5,000$ transactions for fixed value of $minsup$ were conducted and only runtime was presented).

In serial algorithms *MAFIA* [3] and *BitTableFI* [6] Burdick et al. and Dong et al., respectively, used vertical bitmap to compress the transaction database for quick candidate itemsets generation and support count. *Vertical bitmap* is a set of integer in which every bit represents an item. If an item i appears in a transaction j, then bit j of the bitmap for item i is set to one; otherwise, the bit is set to zero. This idea is applied to transactions and itemsets. In cases where itemsets appear in a significant number of transactions, the vertical bitmap is the smallest representation of the information. However, the weakness of a vertical representation is the sparseness of the bitmaps, especially at the lower support levels.

In this paper we suggested a parallel version of the DIC algorithm for Intel Xeon and Xeon Phi many-core systems (which was done for the first time, to the best of our knowledge) where we use direct bit representation of both transaction database and itemsets.

IV. Experimental Evaluation

To evaluate the developed algorithm, we performed experiments on the Tornado SUSU [8] supercomputer's node (cf. Tab. I for its specifications).

TABLE I: Specifications of hardware

Specifications	CPU	Coprocessor
Model, Intel Xeon	X5680	Phi SE10X
Cores	6	61
Frequency, GHz	3.33	1.1
Threads per core	2	4
Peak performance, TFLOPS	0.371	1.076
Memory, Gb	24	8
Cache, Mb	12	30.5

We compiled source code using Intel `icpc` compiler (version 15.0.3). Experiments have been performed on realistic and synthetic[3] datasets summarized in Tab. II.

TABLE II: Specifications of datasets

Dataset	Category	# transactions	# items
SKIN [5]	Real	245,057	11
RECORDLINK [11]	Real	574,913	29
20M	Synthetic	$2 \cdot 10^7$	64

In the experiments, we studied the following aspects of the developed algorithm. We compared the performance of parallel DIC with serial implementations of DIC[4] and Apriori[5] algorithms. We also evaluated the scalability of our algorithm depending on the value of M (the number of transactions that should be processed before stop) and on $minsup$ threshold.

Fig. 1 illustrates the results of the first set of experiments where we compare the performance of parallel DIC with serial DIC and Apriori on CPU. As was seen, serial DIC performs the worst for all the datasets we have tested on[6] and this is in accordance with testing results of B. Goethals. For datasets with relatively small number of short transactions, serial Apriori performs best whereas parallel DIC demonstrates degradation of the performance. However, in case of a large dataset, parallel DIC outperforms serial Apriori. Hence, our algorithm behaves the best way when the transaction database provides sufficient amount of work in support counting, which is the heaviest part of the algorithm. This is why we use 20M dataset in the next set of experiments.

[3]We use IBM Quest Synthetic Data Generator similar to original paper [2].

[4]Frequent Pattern Mining Implementations by Bart Goethals

[5]Apriori – Frequent Item Set Mining by Christian Borgelt

[6]For 20M dataset Serial DIC was stopped after 30 hours without output.

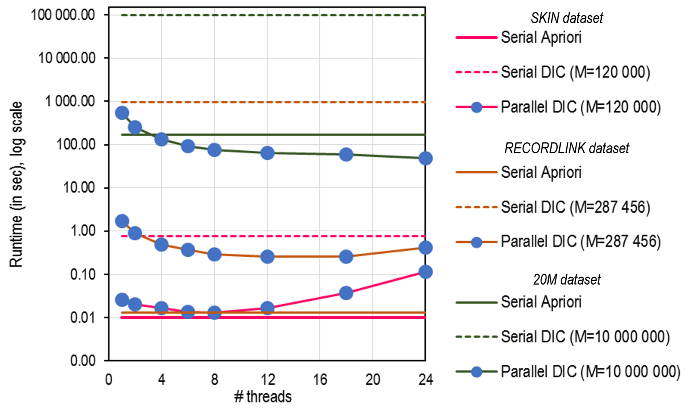
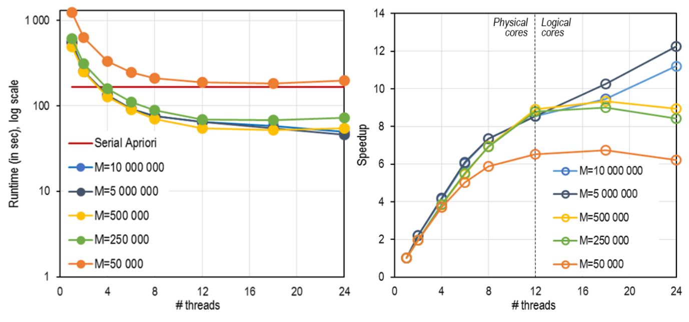
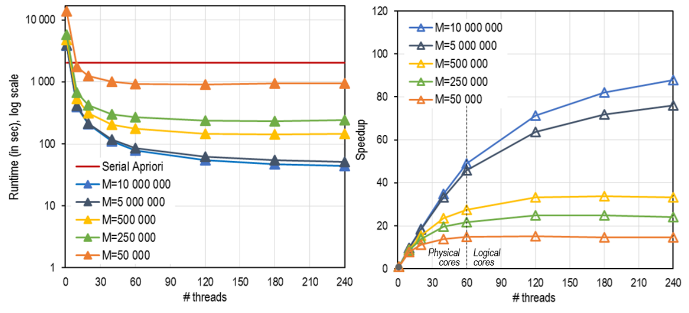


Fig. 1: Comparison of performance ($minsup = 0.1$)

Fig. 2 depicts the results of the second set of experiments where we studied the scalability of parallel DIC w.r.t. M parameter on both platforms using 20M dataset. Experimental results show that parallel DIC outperforms serial Apriori much more often than not. A greater value of M results in less runtime and greater speedup. On both platforms at greater value of M our algorithm shows speedup closer to linear when the number of threads matches the number of physical cores the algorithm is running on and speedup becomes sub-linear when the algorithm uses more than one thread per physical core. Parallel DIC achieves up to 12× and 90× speedup on CPU and Xeon Phi, respectively.

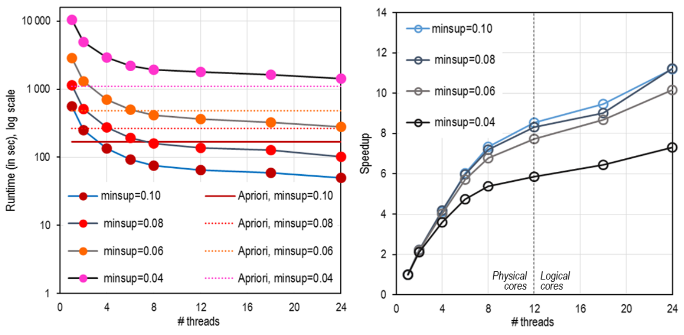


(a) Intel Xeon CPU

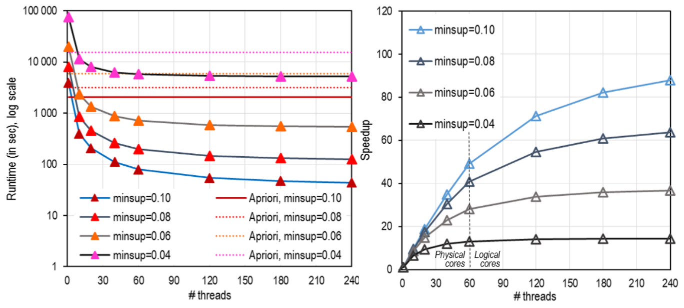


(b) Intel Xeon Phi coprocessor

Fig. 2: Scalability w.r.t. M (20M dataset, $minsup = 0.1$)



(a) Intel Xeon CPU



(b) Intel Xeon Phi coprocessor

Fig. 3: Scalability w.r.t. $minsup$ (20M dataset, $M = 10^7$)

Fig. 3 presents the results of the third set of experiments where we evaluated scalability of parallel DIC w.r.t. $minsup$ threshold on both platforms. Our algorithm still shows better speedup when only physical cores are involved. Runtime and speedup of the algorithm expectedly suffer from decreasing of $minsup$ value. Parallel DIC outperforms serial Apriori with the exception of hard case $minsup = 0.02$ on CPU where our algorithm shows almost the same runtime as Apriori when the maximum number of threads is employed. On the Intel Xeon Phi platform it is enough for our algorithm to use 10 threads to overtake serial Apriori.

Summing up, parallel DIC demonstrates good performance and scalability on both platforms.

Conclusion

In this paper we presented a parallel implementation of Dynamic Itemset Counting (DIC) algorithm for Intel many-core systems, where DIC is a variation of classical Apriori algorithm for frequent itemset mining.

We enhance the DIC algorithm by adding two more stages, which are devoted to reducing the number of itemsets to perform support counting of. We also propose direct bit representation for transactions and itemsets with the assumption that such a representation of the transaction database fits in main memory. This technique reduces memory space for storing the transaction database and also simplifies support counting and candidate itemsets generation via logical bitwise operations. We parallelize the DIC algorithm through of OpenMP technology and thread-level parallelism. Our algorithm balances support counting between threads depending on the current total number of candidate itemsets. We performed experimental evaluation on the platforms of the Intel Xeon CPU and the Intel Xeon Phi coprocessor with large synthetic database, showing the good performance and scalability of the proposed algorithm.

In continuation of the presented research, we plan to implement the developed parallel algorithm for the case of cluster systems based on nodes with the Intel Xeon Phi many-core coprocessor on-board.

References

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 1994, pp. 487–499.

[2] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, J. Peckham, Ed. ACM Press, 1997, pp. 255–264.

[3] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A maximal frequent itemset algorithm," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 11, pp. 1490–1504, 2005.

[4] D. W. Cheung, K. Hu, and S. Xia, "An adaptive algorithm for mining association rules on shared-memory parallel machines," *Distributed and Parallel Databases*, vol. 9, no. 2, pp. 99–132, 2001.

[5] A. Dhall, G. Sharma, R. Bhatt, and G. M. Khan, "Adaptive digital makeup," in *Advances in Visual Computing, 5th International Symposium, ISVC 2009, Las Vegas, NV, USA, November 30 - December 2, 2009, Proceedings, Part II*, ser. Lecture Notes in Computer Science, G. Bebis, R. D. Boyle, B. Parvin, D. Koracin, Y. Kuno, J. Wang, R. Pajarola, P. Lindstrom, A. Hinkenjann, L. M. Encarnação, C. T. Silva, and D. S. Coming, Eds., vol. 5876. Springer, 2009, pp. 728–736.

[6] J. Dong and M. Han, "Bittablefi: An efficient mining frequent itemsets algorithm," *Knowl.-Based Syst.*, vol. 20, no. 4, pp. 329–335, 2007.

[7] M. HooshSadat, H. W. Samuel, S. Patel, and O. R. Zaïane, "Fastest association rule mining algorithm predictor (FARM-AP)," in *Fourth International C* Conference on Computer Science & Software Engineering, C3S2E 2011, Montreal, Quebec, Canada, May 16-18, 2011, Proceedings*, B. C. Desai, A. Abran, and S. P. Mudur, Eds. ACM, 2011, pp. 43–50.

[8] P. Kostenetskiy and A. Safonov, "Susu supercomputer resources," in *PCT'2016, International Scientific Conference on Parallel Computational Technologies, Arkhangelsk, Russia, March 29–31, 2016*, L. Sokolinsky and I. Starodubov, Eds. CEUR Workshop Proceedings. vol. 1576, 2016, pp. 561–573.

[9] P. Kumar, P. Bhatt, and R. Choudhury, "Bitwise dynamic itemset counting algorithm," in *Proceedings of the ICCIC 2015, IEEE International Conference on Computational Intelligence and Computing Research, December 10–12, 2015, Madurai, India*, N. Krishnan and M. Karthikeyan, Eds. IEEE, 2015, pp. 1–4.

[10] P. Paranjape-Voditel and U. Deshpande, "A dic-based distributed algorithm for frequent itemset generation," *JSW*, vol. 6, no. 2, pp. 306–313, 2011.

[11] M. Sariyar, A. Borg, and K. Pommerening, "Controlling false match rates in record linkage using extreme value theory," *Journal of Biomedical Informatics*, vol. 44, no. 4, pp. 648–654, 2011.

[12] B. Schlegel, T. Karnagel, T. Kiefer, and W. Lehner, "Scalable frequent itemset mining on many-core processors," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, R. Johnson and A. Kemper, Eds. ACM, 2013, p. 3.

[13] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[14] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), Newport Beach, California, USA, August 14-17, 1997*, D. Heckerman, H. Mannila, and D. Pregibon, Eds. AAAI Press, 1997, pp. 283–286.