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

I. SERIAL DIC ALGORITHM

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 II provides a brief description of an original DIC algorithm. The proposed parallel algorithm is presented in section III. In section IV related work is discussed. The results of experimental evaluation of the algorithm are described in section V. 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
first pass over transaction database it processes 1-itemsets and finds \( L_1 \) set. A subsequent pass \( k \) consists of two steps, namely candidate generation and pruning. At the candidate generation step Apriori combines elements of \( L_{k-1} \) set to form candidate (potentially frequent) \( k \)-itemsets. At the pruning step it gets rid of infrequent candidates using the \textit{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 \( D \), in \( minsup \), in \( M \), out \( L \))**

1. \( \text{Initialize sets of itemsets} \)
2. \( \text{SolidBox} \leftarrow \emptyset; \text{SolidCircle} \leftarrow \emptyset; \text{DashedBox} \leftarrow \emptyset \)
3. \( \text{DashedCircle} \leftarrow I \)
4. \( \text{while} \ \text{DashedCircle} \cup \text{DashedBox} \neq \emptyset \text{ do} \)
5. \( \quad \text{Read}(D, M, \text{Chunk}) \)
6. \( \quad \text{if} \ \text{EOF}(D) \text{ then} \)
7. \( \quad \quad \text{Rewind}(D) \)
8. \( \quad \quad \text{for all} \ T \in \text{Chunk} \)
9. \( \quad \quad \quad \text{Count support of itemsets} \)
10. \( \quad \quad \quad \text{for all} \ I \in \text{DashedCircle} \cup \text{DashedBox} \)
11. \( \quad \quad \quad \quad \text{if} \ I \subseteq T \text{ then} \)
12. \( \quad \quad \quad \quad \quad \text{support}(I) \leftarrow \text{support}(I) + 1 \)
13. \( \quad \quad \quad \text{Generate candidate itemsets} \)
14. \( \quad \quad \quad \text{for all} \ I \in \text{DashedCircle} \)
15. \( \quad \quad \quad \quad \text{if} \ \text{support}(I) \geq \text{minsup} \text{ then} \)
16. \( \quad \quad \quad \quad \quad \text{MoveItemset}(I, \text{DashedBox}) \)
17. \( \quad \quad \quad \text{Check full pass completion for itemsets} \)
18. \( \quad \quad \quad \text{for all} \ I \in \text{DashedCircle} \text{ do} \)
19. \( \quad \quad \quad \quad \text{if} \ \text{IsPassCompleted}(I) \text{ then} \)
20. \( \quad \quad \quad \quad \quad \text{MoveItemset}(I, \text{DashedBox}) \)
21. \( \quad \quad \quad \text{for all} \ I \in \text{DashedBox} \text{ do} \)
22. \( \quad \quad \quad \quad \text{if} \ \text{IsPassCompleted}(I) \text{ then} \)
23. \( \quad \quad \quad \quad \quad \text{MoveItemset}(I, \text{SolidBox}) \)
24. \( \quad \quad \quad \text{for all} \ I \in \text{SolidBox} \text{ do} \)
25. \( \quad \quad \quad \quad \text{if} \ \text{IsPassCompleted}(I) \text{ then} \)
26. \( \quad \quad \quad \quad \quad \text{MoveItemset}(I, \text{SolidBox}) \)

**Algorithm 1. DIC(in \( D \), in \( minsup \), in \( M \), out \( L \))**

\( \mathcal{L} \leftarrow \text{SolidBox} \)

The DIC algorithm tries to reduce the number of passes 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 \( \text{mins}\) 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 \textit{direct bit representation} for both transactions and itemsets. For a transaction \( T \subseteq D \) (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 \( W = \left\lceil \frac{m}{\text{sizeof(byte)}} \right\rceil \). In our implementation we use C++ and \textit{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 \textit{BitMask} a function that returns direct bit representation of a given itemset or transaction as a word, i.e. \( \text{BitMask} : \mathcal{I} \rightarrow \mathbb{Z}_4 \). Then direct bit representation of transaction database \( D \) is an \( n \)-element array \( B \), where \( \forall j, 1 \leq j \leq n \ B[j] = \text{BitMask}[T_j] \).

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, \( B \) requires \( n \cdot W \) bytes to store and allows \( B \) 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 \( B \) has been preliminary produced from \( D \) 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 \( \text{BitMask}(I) \land \text{BitMask}(T) = \text{BitMask}(I) \).

Thereby, we implement an itemset as a record structure with the following basic fields, namely \textit{mask} to provide direct bit representation, \( k \) as number of items in the itemset, \textit{stop} as counter to determine when full pass for the given itemset is completed, and \textit{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 $\text{fig}$ 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 $B$, in $\text{minsup}$, in $M$, out $L$)

1: $\text{SOLID.init}();$ $\text{DASHED.init}()$
2: $k \leftarrow 1$
3: $\text{for all } i \in 0..m - 1 \text{ do}$
4: $\quad I.\text{fig} \leftarrow \text{NIL};$ $I.\text{bitset} \leftarrow 0$
5: $\quad I.\text{mask} \leftarrow \text{SelBit}(I.\text{mask}, i)$
6: $\quad I.\text{stop} \leftarrow 0;$ $I.\text{supp} \leftarrow 0;$ $I.k \leftarrow k$
7: $\quad \text{SOLID.push_back}(I)$
8: $\text{stop}_{\text{max}} \leftarrow \lceil \frac{n}{M} \rceil;$ $\text{stop} \leftarrow 0$
9: $\text{FirstPass}(\text{SOLID}, \text{DASHED})$
10: $\text{while not } \text{DASHED.empty}() \text{ do}$
11: $\quad \text{Scan database and rewind if necessary}$
12: $\quad \text{stop} \leftarrow \text{stop} + 1$
13: $\quad \text{if } \text{stop} > \text{stop}_{\text{max}} \text{ then}$
14: $\quad \text{stop} \leftarrow 1$
15: $\text{first} \leftarrow (\text{stop} - 1) \cdot M;$ $\text{last} \leftarrow \text{stop} \cdot M - 1$
16: $\quad k \leftarrow k + 1$
17: $\quad \text{CountSupport(} \text{DASHED})$
18: $\quad \text{CutDashedCircle(} \text{DASHED})$
19: $\quad \text{GenCandidates(} \text{DASHED})$
20: $\quad \text{CheckFullPass(} \text{DASHED})$
21: $\text{L} \leftarrow \{I \in \text{SOLID}, I.\text{fig} = \text{BOX}\}$

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 first pass [4]. This means that we initially perform one full pass over $\mathcal{D}$ to find $L_1$, the set of frequent 1-itemsets (this done similarly to Alg. 3). Then candidate 2-itemsets are computed from $L_1$ 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)

1: if DASHED.size() $\geq$ num_of_threads then
2: $\quad \#pragma omp parallel for$
3: $\quad \text{for all } I \in \text{DASHED} \text{ do}$
4: $\quad \quad I.\text{stop} \leftarrow I.\text{stop} + 1$
5: $\quad \quad \text{for all } T \in B[\text{first}] .. B[\text{last}] \text{ do}$
6: $\quad \quad \quad \text{if } I.\text{mask} \text{ AND } T = I.\text{mask} \text{ then}$
7: $\quad \quad \quad \quad I.\text{supp} \leftarrow I.\text{supp} + 1$
8: $\quad \quad \text{else}$
9: $\quad \quad \quad \text{omp_set_nested(true)}$
10: $\quad \quad \quad \#pragma omp parallel for$
11: $\quad \quad \quad \text{num_threads(DASHED.size())}$
12: $\quad \quad \text{for all } I \in \text{DASHED} \text{ do}$
13: $\quad \quad \quad I.\text{stop} \leftarrow I.\text{stop} + 1$
14: $\quad \quad \text{num_threads(}$
15: $\quad \quad \quad \text{DASHED.size()}$
16: $\quad \quad \text{for all } T \in B[\text{first}] .. B[\text{last}] \text{ do}$
17: $\quad \quad \quad \text{if } I.\text{mask} \text{ AND } T = I.\text{mask} \text{ then}$
18: $\quad \quad \quad \quad I.\text{supp} \leftarrow I.\text{supp} + 1$

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
is that each node sends messages to other nodes after every 
through omp parallel for (cf. Alg. 5). This activity is also parallelized along itemsets
and if yes, we make the itemset “solid” and stop counting it
if an itemset has been counted through all the transactions
Algorithm 5.
CheckFullPass(in out DASHED)

```
#pragma omp parallel for
2: for all I ∈ DASHED and I.fig = CIRCLE do
   if I.supp ≥ minsup then
     ▷ Move appropriate itemsets to Dashed Box set
      I.fig ← BOX
   else
     ▷ Prune clearly infrequent itemset
8: suppmax ← I.supp + M · (stopmax − I.stop)
   if suppmx < minsup then
10: I.fig ← NIL
  ▷ Prune supersets of infrequent itemset
12: for all J ∈ DASHED and J.fig = CIRCLE do
      if J.mask AND I.mask = I.mask then
14:      J.fig ← NIL
DASHED.erase(∀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 the problem of a large number of candid-
dates 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 greater 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× 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

<table>
<thead>
<tr>
<th>Specifications</th>
<th>CPU</th>
<th>Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>X5680</td>
<td>Phi SE10X</td>
</tr>
<tr>
<td>Cores</td>
<td>6</td>
<td>61</td>
</tr>
<tr>
<td>Frequency, GHz</td>
<td>3.33</td>
<td>1.1</td>
</tr>
<tr>
<td>Threads per core</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Peak performance, TFLOPS</td>
<td>0.371</td>
<td>1.076</td>
</tr>
<tr>
<td>Memory, Gb</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>Cache, Mb</td>
<td>12</td>
<td>30.5</td>
</tr>
</tbody>
</table>

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

TABLE II: Specifications of datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Category</th>
<th># transactions</th>
<th># items</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIN [5]</td>
<td>Real</td>
<td>245,057</td>
<td>11</td>
</tr>
<tr>
<td>20M</td>
<td>Synthetic</td>
<td>2 · 10⁷</td>
<td>64</td>
</tr>
</tbody>
</table>

In the experiments, we studied the following aspects of the developed algorithm. We compared the performance of parallel DIC with serial implementations of DIC and Apriori 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 \( \text{minsup} \) 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 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.

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.

---

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.
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


