УДК 004.272.25, 004.421, 004.032.24 **DOI: 10.14529/cmse190104**

ПАРАЛЛЕЛЬНЫЙ ПОИСК ЧАСТЫХ НАБОРОВ НА МНОГОЯДЕРНЫХ УСКОРИТЕЛЯХ INTEL MIC

© 2019 М.Л. Цымблер

Южно-Уральский государственный университет
(454080 Челябинск, пр. им. В.И. Ленина, д. 76)
E-mail: mzym@susu.ru

Поступила в редакцию: 26.12.2018

Поиск ассоциативных правил предполагает нахождение устойчивых корреляций между наборами элементов в больших базах транзакционных данных и является одной из основных задач интеллектуального анализа данных. Ассоциативные правила генерируются на основе множества всех наборов, в которых элементы часто встречаются совместно. Алгоритм DIC (Dynamic Itemset Counting) является модификацией классического алгоритма Apriori поиска частых наборов. В отличие от предшественника DIC пытается сократить количество проходов по базе транзакций и сохранить при этом относительно небольшое количество наборов, поддержка которых подсчитывается в рамках одного прохода. В статье рассмотрена проблема ускорения алгоритма DIC на многоядерной архитектуре Intel Many Integrated Core (MIC) для случая, когда база транзакций помещается в оперативную память. Разработанная с помощью технологии OpenMP параллельная реализация алгоритма DIC использует битовое представление транзакций и наборов, что позволяет ускорить и векторизовать подсчет поддержки наборов, реализуемый посредством логических побитовых операций. Проведенные эксперименты с синтетическими и реальными данными подтвердили хорошую производительность и масштабируемость предложенного алгоритма.

Ключевые слова: интеллектуальный анализ данных, поиск ассоциативных правил, OpenMP, Intel Many Integrated Core.

ОБРАЗЕЦ ЦИТИРОВАНИЯ

Цымблер М.Л. Параллельный поиск частых наборов на многоядерных ускорителях Intel MIC // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 1. С. 54–70. DOI: 10.14529/cmse190104.

Введение

Поиск *ассоциативных правил (association rule mining)* предполагает нахождение часто повторяющихся зависимостей в заданном наборе объектов. Наглядным примером применения ассоциативных правил является *анализ рыночной корзины (market basket analysis)*, когда ставится задача нахождения наборов товаров в супермаркете, которые часто покупаются совместно. Ассоциативное правило представляет собой импликацию вида $A \rightarrow B$ $[support = x\ \%, confidence = y\ \%]$, где A и B — непустые и непересекающиеся наборы товаров, а $support$ и $confidence$ — меры полезности правила для аналитика. Поддержка (support) правила показывает, что в x % от общего числа покупок наборы A и B присутствовали одновременно. Достоверность (confidence) правила показывает, что y % покупателей, которые приобрели набор A, приобрели также набор B. Поиск ассоциативных правил применяется в медицине (например, нахождение побочных эффектов лекарств), генной инженерии (поиск часто повторяющихся цепочек ДНК) и других предметных областях.

Поиск ассоциативных правил может быть разбит на две последовательно выполняемые задачи: поиск всех частых наборов и генерация устойчивых ассоциативных правил на основе найденных частых наборов [2]. Формальное определение задачи поиска всех

частых наборов выглядит следующим образом. Пусть дано множество *объектов* $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$, любое непустое его подмножество называют *набором*. Набор из k объектов ($1 \leqslant k \leqslant m$) называют *$k$-набором*. Пусть имеется множество *транзакций* $\mathcal{D}$, в котором каждая транзакция представляет собой пару $(tid; I)$, где tid — уникальный идентификатор транзакции, $I \subseteq \mathcal{I}$ — набор. *Поддержкой набора* $I \subseteq \mathcal{I}$ является доля транзакций $\mathcal{D}$, содержащих данный набор:

$$support(I) = \frac{|\{T \in \mathcal{D} \mid I \subseteq T.I\}|}{|\mathcal{D}|}. \quad (1)$$

Наперед задаваемый *порог поддержки* $minsup$ является параметром задачи. Набор, имеющий поддержку не ниже $minsup$, называют *частым*, иначе набор называют *редким*. Множество всех частых k-наборов обозначают $\mathcal{L}_k$. Решением задачи *поиска частых наборов* будет множество $\mathcal{L} = \cup_{k=1}^{k_{max}} \mathcal{L}_k$, где k_{max} — максимальное количество объектов в частом наборе.

В данной статье предлагается параллельный алгоритм поиска всех частых наборов для многоядерных ускорителей Intel Many Integrated Core (MIC) [10]. Ускорители Intel MIC основаны на архитектуре Intel x86 и поддерживают соответствующие модели и инструменты параллельного программирования. Устройства MIC обеспечивают большое количество энергоэффективных вычислительных ядер (до 72) с малой (относительно обычных многоядерных процессоров Intel) тактовой частотой, обладающих высокой пропускной способностью локальной памяти и 512-битными векторными регистрами. Системы MIC, как правило, дают наибольшую производительность в приложениях, предполагающих вычислительные операции над большими объемами данных (десятки миллионов элементов и более), векторизуемые компилятором [22]. Под векторизацией понимается замена нескольких скалярных операций в теле цикла с фиксированным количеством повторений на одну векторную операцию [3].

Остаток статьи организован следующим образом. В разделе 1 приведен обзор работ по тематике исследования. Описание последовательного алгоритма DIC приведено в разделе 2. Раздел 3 содержит описание предложенного параллельного алгоритма поиска частых наборов. Результаты экспериментального исследования разработанного алгоритма приведены в разделе 4. В заключении обсуждается область применения разработанного алгоритма и суммируются полученные результаты.

1. Обзор работ

Классическим алгоритмом решения задачи поиска частых наборов является алгоритм *Apriori* [2]. Идея *Apriori* заключается в итеративной генерации множества *кандидатов* в частые наборы и последующем отборе кандидатов с подходящим значением поддержки. Итерация осуществляется по k, количеству объектов в наборах-кандидатах, начиная с 1. В алгоритме используется следующее свойство *антимонотонности поддержки* (принцип *a priori*), которое позволяет исключать из рассмотрения заведомо редкие наборы: если k-набор является редким, то содержащий его $(k+1)$-набор также является редким.

Алгоритм *Dynamic Itemset Counting (DIC)* [5] является модификацией *Apriori*. В отличие от предшественника *DIC* пытается сократить количество проходов по множеству транзакций и сохранить при этом относительно небольшое количество наборов, поддержка которых подсчитывается в рамках одного прохода. Авторы отметили большой потенциал

распараллеливания предложенного алгоритма посредством разделения транзакций между вычислителями.

Алгоритм *DIC-OPT* [19] является параллельной версией алгоритма *DIC* для вычислительных систем с распределенной памятью. Основная идея алгоритма заключается в том, что каждый вычислительный узел выполняет рассылку сообщений, содержащих значения поддержки наборов, всем остальным узлам по завершении обработки очередного блока из M транзакций. Авторы провели эксперименты на вычислительной системе из 12 узлов, где *DIC-OPT* показал сублинейное ускорение.

В работе [8] представлен алгоритм *APM*, который является модификацией алгоритма *DIC* для SMP-систем. Процессоры SMP-системы динамически генерируют наборы-кандидаты независимо друг от друга и не требуют синхронизации. Транзакции распределяются по узлам системы. Эксперименты, проведенные на вычислительной системе Sun Enterprise 4000 из 12 узлов, показали, что APM опережает параллельные реализации классического алгоритма Apriori. Однако, ускорение APM постепенно снижается до 4, когда количество задействованных узлов больше четырех.

Алгоритм *mcEclat* [21] является параллельной версией *Eclat* [23] для сопроцессора Intel Xeon Phi. *mcEclat* использует представление транзакций в виде вертикальных битовых карт: *tid* всех транзакций, в которых присутствует данный объект, преобразуются в битовую карту этого объекта, где в соответствующих позициях биты установлены в 1. Для вычисления поддержки набора над битовыми картами входящих в набор объектов выполняется логическая побитовая операция `AND` и затем подсчитывается количество битов результата, установленных в 1. Эксперименты показали ускорение алгоритма до 100 на 240 нитях сопроцессора, однако реализация не в полной мере использует возможности векторизации вычислений Xeon Phi и не опережает себя на платформе двухпроцессорной системы Intel Xeon.

В работах [6, 9, 14] предложены различные последовательные алгоритмы поиска частых наборов на основе использования битовых карт: *MAFIA*, *BitTableFI* и *BitwiseDIC* (версия алгоритма *DIC* [5]) соответственно.

2. Последовательный алгоритм поиска частых наборов

Алгоритм *DIC* (см. алг. 1) разбивает множества транзакций $\mathcal{D}$ на логические блоки, состоящие из $\lceil \frac{|\mathcal{D}|}{M} \rceil$ транзакций, где число транзакций в блоке M ($1 \leqslant M \leqslant |\mathcal{D}|$) является параметром алгоритма.

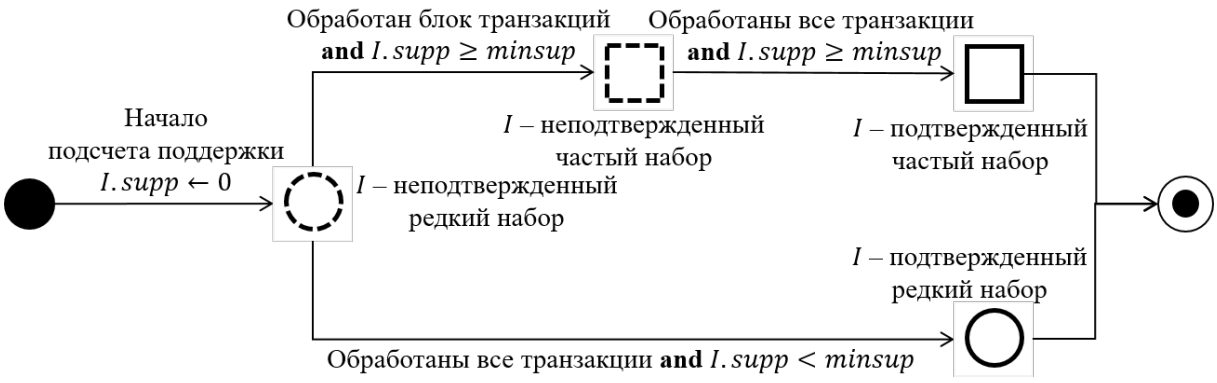


Рис. 1. Жизненный цикл набора в алгоритме *DIC*

Алгоритм 1 DIC(IN $\mathcal{D}$, $minsup$, M, OUT $\mathcal{L}$)

▷ Инициализация множеств наборов

1: $SolidBox \leftarrow \varnothing$; $SolidCircle \leftarrow \varnothing$; $DashedBox \leftarrow \varnothing$; $DashedCircle \leftarrow \mathcal{I}$
2: **while** $DashedCircle \cup DashedBox \neq \varnothing$ **do**

▷ Чтение транзакций

3: Read($\mathcal{D}, M, Chunk$)
4: **if** EOF($\mathcal{D}$) **then**
5: Rewind($\mathcal{D}$)
6: **for all** $T \in Chunk$ **do**

▷ Подсчет поддержки наборов

7: **for all** $I \in DashedCircle \cup DashedBox$ **do**
8: **if** $I \subseteq T$ **then**
9: $support(I) \leftarrow support(I) + 1$

▷ Генерация наборов-кандидатов

10: **for all** $I \in DashedCircle$ **do**
11: **if** $support(I) \geqslant minsup$ **then**
12: $DashedBox \leftarrow DashedBox \cup I$
13: **for all** $i \in \mathcal{I}$ **do**
14: $C \leftarrow I \cup i$
15: **if** $\forall s \subseteq C \ s \in SolidBox \cup DashedBox$ **then**
16: $DashedCircle \leftarrow DashedCircle \cup C$

▷ Проверка завершения полного прохода

17: **for all** $I \in DashedCircle \cup SolidBox$ **do**
18: **if** IsPassCompleted(I) **then**
19: **switch** Shape(I)
20: *dashed*: $DashedBox \leftarrow DashedBox \cup I$
21: *solid*: $SolidBox \leftarrow SolidBox \cup I$
22: $\mathcal{L} \leftarrow SolidBox$

DIC различает четыре вида наборов, для названия которых используются метафоры: *пунктирные окружности*, *пунктирные квадраты*, *сплошные окружности* и *сплошные квадраты*. Жизненный цикл набора в алгоритме *DIC* представлен на рис. 1. Для «пунктирных» наборов необходимо выполнить подсчет поддержки, в то время как для «сплошных» наборов подсчет поддержки закончен. «Квадрат» соответствует частому набору, «окружность» — редкому. В соответствии с этим определяются четыре непересекающихся множества наборов: *DashedCircle*, *DashedBox*, *SolidCircle* и *SolidBox*. Множества *DashedCircle* и *DashedBox* содержат неподтвержденные редкие и неподтвержденные частые наборы соответственно, а множества *SolidCircle* и *SolidBox* — подтвержденные редкие и подтвержденные частые наборы соответственно.

При инициализации множества *DashedBox*, *SolidCircle* и *SolidBox* полагаются пустыми, а множество *Dashed Circle* заполняется 1-наборами из множества объектов $\mathcal{I}$. При обработке транзакций блока *DIC* вычисляет поддержку «пунктирных» наборов из множеств *DashedCircle* и *DashedBox*. По завершении обработки блока наборы, поддержка которых сравнялась или превысила порог $minsup$ перемещаются из множества *DashedCircle* в *DashedBox*. В множество *DashedCircle* добавляется каждый набор, который является

надмножеством таких наборов из *DashedBox*, любое подмножество которых является «квадратом». После обработки последнего блока выполняется переход к первому блоку транзакций. Алгоритм завершается, когда множества *DashedCircle* и *DashedBox* становятся пустыми.

3. Поиск частых наборов на ускорителе Intel Xeon Phi

3.1. Проектирование структур данных

Параллельный алгоритм *PhiDIC* [24] использует *битовое представление* наборов и транзакций. Транзакция $T \subseteq \mathcal{D}$ (набор $I \subseteq \mathcal{I}$, соответственно) представляется в виде слова, в котором каждый $(p-1)$-й бит установлен в 1, если элемент $i_p \in T$ ($i_p \in I$, соответственно), и остальные биты установлены в 0. Количество битов в слове W зависит от используемой системы программирования и определяется как $W = \lceil \frac{m}{sizeof(byte)} \rceil$. В предлагаемой реализации используется система программирования C++, а набор и транзакция представлены значением типа данных `unsigned long long int`, таким образом, в данной реализации $W = 8$ и $m = 64$.

Введем *функцию битовой маски* $BitMask : \mathcal{I} \to \mathbb{N}$, которая возвращает натуральное число, являющееся битовым представлением указанной транзакции либо набора. Например, для набора $\{i_1, i_4, i_5\}$ функция $BitMask$ возвратит целое число 25, имеющее такое битовое представление, в котором 0-й, 3-й и 4-й биты установлены в 1, а остальные биты равны 0. Тогда *битовой картой* базы транзакций $\mathcal{D}$ назовем n-элементный одномерный массив $\mathcal{B}$, где $\mathcal{B}_j = BitMask(T_j)$, $1 \leqslant j \leqslant n$.

Использование битовых масок наборов и битовой карты базы транзакций упрощает подсчет поддержки наборов и обеспечивает векторизацию этой операции. Действительно, факт вхождения набора I в транзакцию T ($I \subseteq T$), установление которого необходимо при подсчете поддержки данного набора, может быть определен с помощью одной логической побитовой операции $BitMask(I)$ `AND` $BitMask(T)$=$BitMask(I)$, а циклическое выполнение данной операции может быть автоматически векторизовано компилятором.

Набор, таким образом, реализуется как структура со следующими полями:

- *mask* — битовая маска набора;
- k — количество элементов в наборе;
- *stop* — счетчик части транзакций, в которых проверено наличие данного набора;
- *supp* — поддержка данного набора;
- *shape* — вид данного набора (`BOX` или `CIRCLE`, либо `NULL`).

Множество наборов реализуется с помощью класса `vector` из стандартной библиотеки классов C++ (Standard Template Library) [16]. Данный класс реализует массив элементов, принадлежащих одному типу данных и обеспечивает операции добавления, удаления элементов, доступ к элементу по его индексу и итерацию элементов массива. Каждый из векторов `DASHED` и `SOLID` обеспечивает хранение двух множеств наборов: *DashedCircle*, *DashedBox* и *SolidCircle*, *SolidBox* соответственно. Хранение двух множеств в рамках одного вектора обеспечивает меньшие накладные расходы, чем выделение одного вектора на каждое множество элементов: для перемещения элемента из одного множества в другое достаточно изменить значение поля *shape*.

Алгоритм 2 PhiDIC(in $\mathcal{B}$, $minsup$, M; out $\mathcal{L}$)

```
1: SOLID.init(); DASHED.init()
2: for all i ∈ 0..m − 1 do
3:     I.shape ← NIL; I.mask ← 0; I.mask ← SetBit(I.mask, i)
4:     I.stop ← 0; I.supp ← 0; I.k ← 1
5:     SOLID.push_back(I)
6: k ← 1; stop ← 0; stop_max ← ⌈n/M⌉
7: while not DASHED.empty() do
8:     k ← k + 1; stop ← stop + 1
9:     if stop > stop_max then
10:        stop ← 1
11:    first ← (stop − 1) · M; last ← stop · M − 1
12:    CountSupport(DASHED)
13:    Prune(DASHED)
14:    MakeCandidates(DASHED)
15:    CheckFullPass(DASHED)
16: L ← {I | I ∈ SOLID ∧ I.shape = BOX}
```

3.2. Распараллеливание поиска частых наборов

Разработанная параллельная реализация поиска частых наборов представлена в алг. 2. Распараллеливанию подвергаются следующие стадии алгоритма: подсчет поддержки (см. алг. 3), отбрасывание заведомо редких наборов (см. алг. 4) и проверка завершения просмотра наборов-кандидатов по всей базе транзакций (см. алг. 5).

Реализация подсчета поддержки показана в алг. 3. Подсчет выполняется посредством двух вложенных циклов: внешний распараллеливается и выполняется по наборам-кандидатам, а внутренний — по транзакциям. Это позволяет избежать гонок данных при обновлении поддержки одного и того же набора разными нитями, которое может быть выполнено одновременно. Алгоритм различает два случая в зависимости от того, больше ли мощность множества наборов-кандидатов, чем количество нитей, или нет. В первом случае внешний цикл распараллеливается на все доступные нити. Во втором случае алгоритм активирует режим вложенного параллелизма, и внешний цикл распараллеливается на количество нитей, равное количеству наборов-кандидатов.

Внутренний цикл по транзакциям распараллеливается таким образом, что каждая нить внешнего цикла инициирует (*fork*) одинаковое количество подчиненных нитей, выполняющих подсчет поддержки. Указанная техника позволяет сбалансировать нагрузку нитей на финальной стадии алгоритма, когда общее количество наборов-кандидатов последовательно уменьшается, что позволяет повысить общую производительность алгоритма.

Реализация отбрасывания заведомо редких наборов представлена в алг. 4. Максимально возможная поддержка набора вычисляется путем сложения текущего значения поддержки данного набора с количеством транзакций, которые еще не были обработаны. Если вычисленная максимально возможная поддержка меньше, чем порог *minsup*, то данный набор заведомо редкий и может быть исключен из дальнейшего рассмотрения. Далее

Алгоритм 3 CountSupport(in out $DASHED$)

```
1: if DASHED.size() ⩾ num_of_threads then
2:     #pragma omp parallel for
3:     for all I ∈ DASHED do
4:         I.stop ← I.stop + 1
5:         for all T ∈ B_first .. B_last do
6:             if I.mask AND T = I.mask then
7:                 I.supp ← I.supp + 1
8: else
9:     omp_set_nested(true)
10:    #pragma omp parallel for num_threads(DASHED.size())
11:    for all I ∈ DASHED do
12:        I.stop ← I.stop + 1
13:        #pragma omp parallel for reduction(+:I.supp) num_threads(⌈num_of_threads / DASHED.size()⌉)
14:        for all T ∈ B_first .. B_last do
15:            if I.mask AND T = I.mask then
16:                I.supp ← I.supp + 1
```

Алгоритм 4 Prune(in out $DASHED$)

```
1: #pragma omp parallel for
2: for all I ∈ DASHED and I.shape = CIRCLE do
3:     if I.supp ⩾ minsup then
4:         I.shape ← BOX
5:     else
6:         supp_max ← I.supp + M · (stop_max − I.stop)
7:         if supp_max < minsup then
8:             I.shape ← NIL
9:             for all J ∈ DASHED and J.shape = CIRCLE do
10:                if I.mask AND J.mask = I.mask then
11:                    J.shape ← NIL
12: DASHED.erase({I | I.shape = NIL})
```

Алгоритм 5 CheckFullPass(in out $DASHED$)

```
1: #pragma omp parallel for
2: for all I ∈ DASHED do
3:     if I.stop = stop_max then
4:         if I.supp ⩾ minsup then
5:             I.shape ← BOX
6:         SOLID.push_back(I)
7:         I.shape ← NIL
8: DASHED.erase({I | I.shape = NIL})
```

в соответствии с принципом *a priori* отбрасывается каждый набор-кандидат, который является надмножеством отброшенного набора.

По завершении отбрасывания подпрограмма `MakeCandidates` выполняет генерацию наборов-кандидатов для их обработки на следующей итерации алгоритма. Новые кандидаты получаются посредством применения логической побитовой операции `OR` к каждой паре наборов из множества `DASHED`, помеченных как `BOX` (частый). Финальным шагом обработки наборов-кандидатов является проверка завершения просмотра этих наборов по всей базе данных транзакций (см. алг. 5). Если просмотр завершен, набора перемещается в множество `SOLID`. Если при этом набор имеет поддержку не ниже порогового значения, он помечается как `BOX` (частый). Цикл обработки наборов-кандидатов распараллеливается с помощью директивы компилятора `#pragma omp parallel for`. По завершении обработки все искомые частые наборы будут перемещены в множество `SOLID` и помечены как `BOX`.

4. Вычислительные эксперименты

4.1. Цели, аппаратная платформа и наборы данных

Для исследования эффективности разработанного алгоритма были проведены вычислительные эксперименты на вычислительном узле кластерной системы «Торнадо ЮУрГУ» [13], характеристики которого приведены в табл. 1.

Таблица 1

Аппаратная платформа экспериментов

Характеристика	Хост	Сопроцессор
Модель, Intel Xeon	X5680	Phi (KNC), SE10X
Количество физических ядер	2×6	61
Гиперпоточность	2	4
Количество логических ядер	24	244
Частота, ГГц	3,33	1,1
Размер VPU, бит	128	512
Пиковая производительность, TFLOPS	0,371	1,076

В работе [24] были проведены эксперименты, показавшие, что при поиске частых наборов в базах из сотен тысяч транзакций масштабируемость *PhiDIC* деградирует, поскольку такие объемы не обеспечивают достаточную вычислительную нагрузку на ускоритель при подсчете поддержки. В данной статье в качестве наборов данных для экспериментов использовались базы из десятков миллионов транзакций, представленные в табл. 2.

Таблица 2

Наборы данных для экспериментов

Набор данных	Вид	Транзакции			Частые наборы (при $minsup = 0.1$)	
		n	m	Ср. длина	Количество	k_{max}
20M	Синтетический	$2 \cdot 10^7$	64	40	4 606	6
Tornado20M	Реальный	$2 \cdot 10^7$	64	15	346	4

Синтетический набор данных 20M подготовлен с помощью генератора IBM Quest Data Generator [12], использованного в экспериментах с оригинальным алгоритмом *DIC* [5]. В итоге набор данных 20M содержит 4 606 частых набора, самый длинный из которых состоит из 6 элементов.

Набор Tornado20M представляет собой журнал с показаниями датчиков напряжения в вычислительных узлах суперкомпьютера «Торнадо ЮУрГУ» [13], снятых в течение одного месяца. Данный журнал используется для нахождения устойчивых ассоциативных правил, связующих вычислительные шкафы, полки, узлы суперкомпьютера и опасные значения напряжения. «Торнадо ЮУрГУ» состоит из 8 шкафов, каждый шкаф состоит из 8 полок, каждая полка состоит из 6 узлов. Рассматривается 4 возможных значения измеряемого напряжения, для каждого из которых различают 4 статуса («меньше нормы», «норма», «больше нормы», «ошибка измерения»). В соответствии с этим транзакция журнала может быть закодирована с помощью 64 бит (8 бит на номер шкафа, 8 бит на номер полки, 48 бит на 6 узлов, где каждая пара битов отражает статус измеренного напряжения). В итоге набор данных Tornado20M содержит 340 частых наборов, самый длинный из которых состоит из 4 элементов.

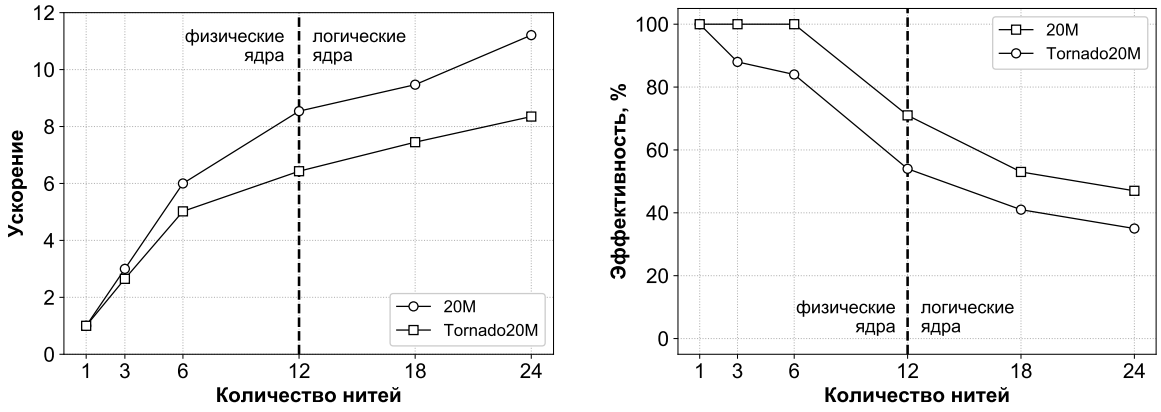
В экспериментах в качестве порога поддержки взято значение $minsup = 0,1$ как наиболее типичное. В качестве параметра количества транзакций в блоке взято значение $M = \lceil n/2 \rceil$, которое минимизирует накладные расходы на создание нитей для подсчета поддержки.

В экспериментах исследовались производительность, ускорение и параллельная эффективность алгоритма *PhiDIC*. Под *производительностью* понимается время работы алгоритма без учета времени загрузки данных в память и выдачи результата. *Ускорение* и *параллельная эффективность* параллельного алгоритма [1], запускаемого на k нитях, вычисляются как $s(k) = \frac{t_1}{t_k}$ и $e(k) = \frac{s(k)}{k}$ соответственно, где t_1 и t_k — время работы алгоритма на одной и k нитях соответственно.

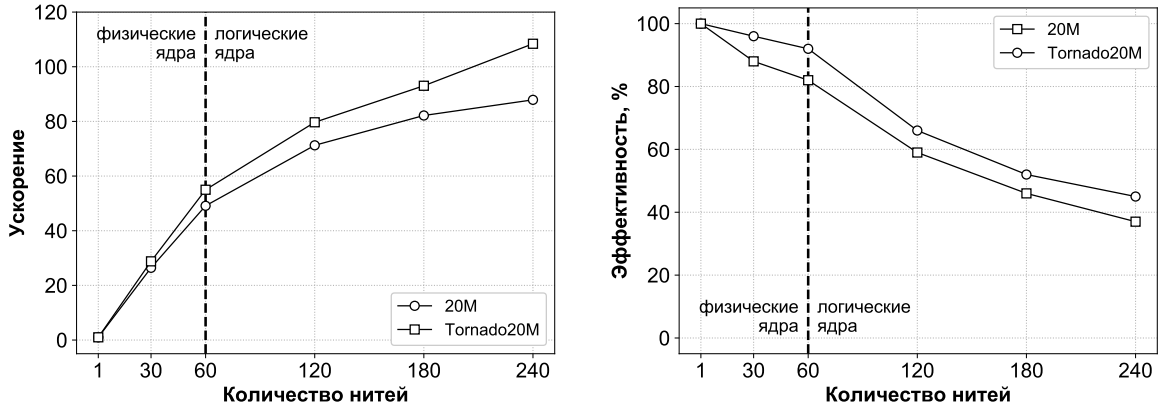
4.2. Результаты и обсуждение

Результаты экспериментов по исследованию ускорения и параллельной эффективности алгоритма *PhiDIC* представлены на рис. 2. На платформе Intel Xeon Phi алгоритм *PhiDIC* показывает ускорение, близкое к линейному и параллельную эффективность, близкую к 100 %, когда количество нитей, на которых запущен алгоритм, совпадает с количеством физических ядер процессора. Если при запуске алгоритм использует более одной нити на физическое процессорное ядро, то ускорение становится сублинейным (его значение уменьшается до 88 и 108 соответственно для наборов данных 20M и Tornado20M), а параллельная эффективность снижается соответствующим образом (до 37 % и 45 % для соответствующих наборов данных). На платформе вычислительного узла с двумя процессорами Intel Xeon наблюдается схожая картина, хотя и с несколько более скромными результатами для набора данных Tornado20M. Ускорение и параллельная эффективность алгоритма в экспериментах на наборе данных Tornado20M уменьшаются до значений 8 и 35 % соответственно, когда алгоритм запускается на максимальном количестве физических ядер.

Результаты экспериментов по исследованию производительности алгоритма *PhiDIC* представлены на рис. 3. Можно видеть, что алгоритм *PhiDIC* работает до полутора раз быстрее на платформе многоядерного процессора Intel Xeon Phi, чем на платформе

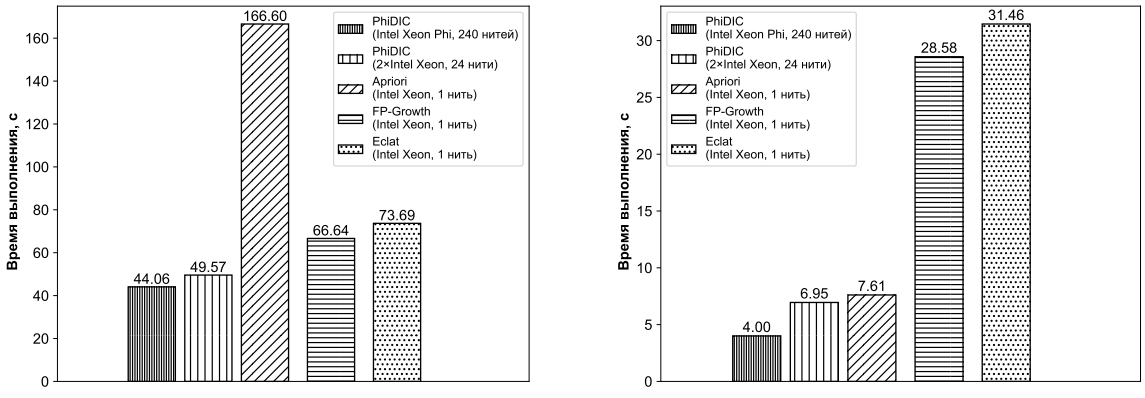


а) Платформа 2×Intel Xeon



б) Платформа Intel Xeon Phi

Рис. 2. Ускорение и параллельная эффективность алгоритма



а) Набор 20M б) Набор Tornado20M

Рис. 3. Производительность алгоритма

вычислительного узла, состоящего из двух обычных процессоров Intel Xeon. Алгоритм *PhiDIC* на платформе многоядерного процессора Intel Xeon Phi опережает в два раза лучшие результаты последовательных алгоритмов-конкурентов (*Apriori* [2], *Eclat* [23] и FP-Growth [11] в реализации [4]) на платформе процессора Intel Xeon.

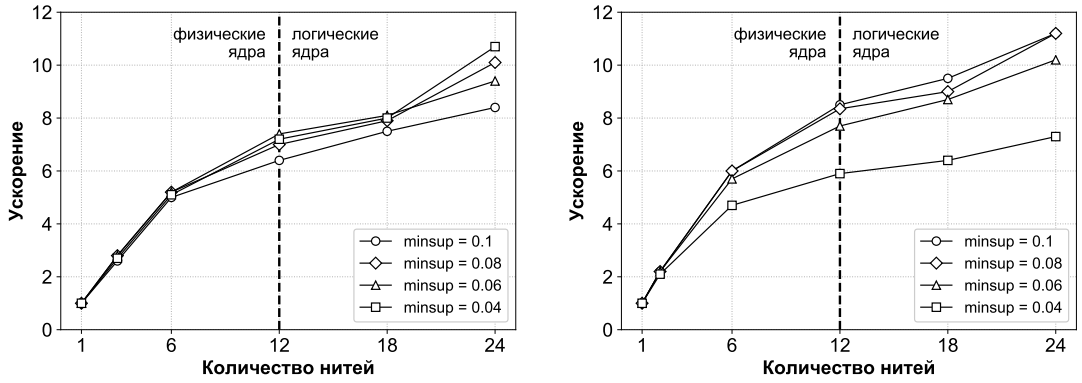
В табл. 3 представлены результаты экспериментов с набором данных Tornado20M, показывающих выгоду векторизации вычислений. Можно видеть, что производительность алгоритма *PhiDIC* на платформе Intel Xeon Phi увеличивается в два раза, если обеспечивается векторизация вычислений.

Таблица 3

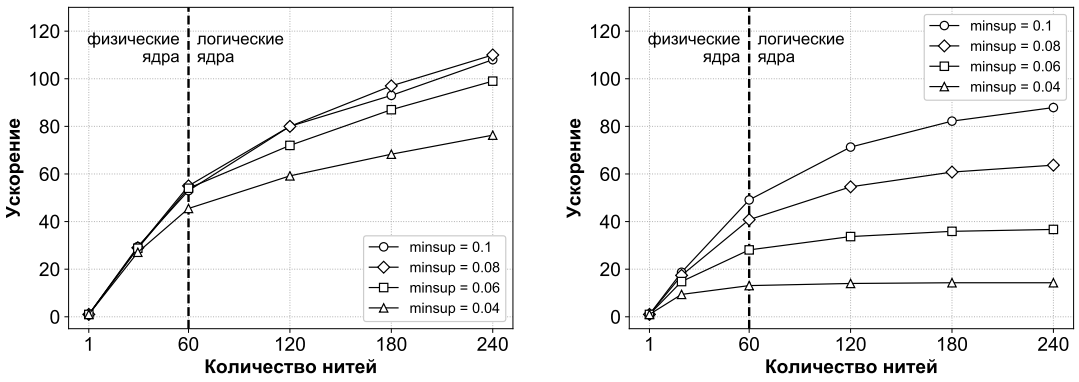
Влияние векторизации на производительность

Платформа	Время выполнения алгоритма, с	
	векторизация включена	векторизация отключена
Intel Xeon Phi	4,00	10,36
Intel Xeon	6,95	8,55

На рис. 4 показано влияние порога *minsup* на ускорение алгоритма. На обеих платформах и для обоих наборов данных ускорение алгоритма ожидаемо страдает от уменьшения значения *minsup*, поскольку это существенно увеличивает количество наборов-кандидатов для вычисления поддержки. Алгоритм *PhiDIC* все еще показывает лучшее ускорение, когда задействованы только физические ядра, и лучшее ускорение на платформе Intel Xeon Phi, чем на двухпроцессорной системе Intel Xeon.



а) Платформа 2×Intel Xeon



б) Платформа Intel Xeon Phi

Рис. 4. Влияние параметра *minsup* на ускорение алгоритма (слева — набор Tornado20M, справа — набор 20M)

Заключение

В статье предложен параллельный алгоритм решения задачи поиска частых наборов *PhiDIC* для многоядерного ускорителя архитектуры Intel MIC. Алгоритм предполагает битовое представление транзакций и наборов. Использование битовых масок наборов и битовой карты базы транзакций упрощает подсчет поддержки наборов и обеспечивает векторизацию этой операции.

С помощью технологии OpenMP распараллеливаются следующие стадии алгоритма: подсчет поддержки, отбрасывание заведомо редких наборов-кандидатов и проверка завершения просмотра наборов-кандидатов по всей базе транзакций. Подсчет поддержки выполняется посредством двух вложенных циклов: внешний распараллеливается и выполняется по наборам-кандидатам, а внутренний — по транзакциям, что позволяет избежать гонок данных. Если мощность множества наборов-кандидатов больше, чем количество нитей, внешний цикл распараллеливается на все доступные нити. В противном случае алгоритм активирует режим вложенного параллелизма, и внешний цикл распараллеливается на количество нитей, равное количеству наборов-кандидатов, а каждая нить внешнего цикла инициирует одинаковое количество подчиненных нитей, выполняющих подсчет поддержки. Указанная техника позволяет сбалансировать нагрузку нитей на финальной стадии алгоритма, когда общее количество наборов-кандидатов последовательно уменьшается, и повысить общую производительность алгоритма.

Результаты проведенных вычислительных экспериментов на синтетических и реальных наборах, состоящих из десятков миллионов транзакций, показывают, что разработанный алгоритм демонстрирует ускорение, близкое к линейному и параллельную эффективность, близкую к 100 %, когда количество нитей, на которых запущен алгоритм, совпадает с количеством физических ядер процессора.

Битовое представление наборов и транзакций, используемое в алгоритме *PhiDIC*, означает, что каждые набор и транзакция, представляемые целым положительным числом, не могут состоять из более чем 64 объектов. Данное ограничение, очевидно, неприемлемо для поиска частых наборов в транзакциях покупок в супермаркете [5], ДНК-микрочипов [7] и др. Однако алгоритм *PhiDIC* потенциально применим для поиска шаблонов в медицинских данных, что подтверждается следующими научными публикациями. В работе [15] описан поиск шаблонов риска заболевания у пациентов клиники, где количество атрибутов не превышает 30. В работах [17, 18] описан механизм трансформации данных медицинских карт в формат транзакций для последующего поиска частых наборов. В экспериментах авторы взяли не более 25 атрибутов из более чем 100 атрибутов медицинской карты пациента, поскольку выбранные атрибуты могут дать полную картину течения болезни. Кроме того, опыт авторов показал, что шаблоны, в которые входит более чем пять медицинских атрибутов, трудно поддаются интерпретации врачами. В работе [20] описан поиск шаблонов в базе данных больницы с более чем 2,5 млн. транзакций с данными пациентов, включая атрибуты, касающиеся демографии, диагностики и употребления лекарств.

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 17-07-00463), Правительства РФ в соответствии с Постановлением № 211 от 16.03.2013 (соглашение № 02.A03.21.0011) и Министерства образования и науки РФ (государственное задание 2.7905.2017/8.9).

Литература

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.
2. Agrawal R., Srikant R. Fast Algorithms for Mining Association Rules in Large Databases // VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago de Chile, Chile. 1994. P. 487–499.
3. Bacon D.F., Graham S.L., Sharp O.J. Compiler Transformations for High-Performance Computing // ACM Computing Surveys. 1994. Vol. 26, No. 4. P. 345–420. DOI: 10.1145/197405.197406.
4. Borgelt C. Frequent Item Set Mining // Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery. 2012. Vol. 2, No. 6, P. 437–456. DOI: 10.1002/widm.1074.
5. Brin S., Motwani R., Ullman J.D., Tsur S. Dynamic Itemset Counting and Implication Rules for Market Basket Data // SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13–15, 1997, Tucson, Arizona, USA. 1997. P. 255–264. DOI: 10.1145/253260.253325.
6. Burdick D., Calimlim M., Flannick J. *et al.* MAFIA: A Maximal Frequent Itemset Algorithm // IEEE Transactions on Knowledge and Data Engineering. 2005. Vol. 17, No. 11. P. 1490–1504. DOI: 10.1109/TKDE.2005.183.
7. Chang Y., Chen J., Tsai Y. Mining Subspace Clusters from DNA Microarray Data Using Large Itemset Techniques // Journal of Computational Biology. 2009. Vol. 16, No. 5. P. 745–768. DOI: 10.1089/cmb.2008.0161.
8. Cheung D.W., Hu K., Xia S. An Adaptive Algorithm for Mining Association Rules on Shared-Memory Parallel Machines // Distributed and Parallel Databases. 2001. Vol. 9, No. 2. P. 99–132. DOI: 10.1023/A:1018951022124.
9. Dong J., Han M. BitTableFI: An Efficient Mining Frequent Itemsets Algorithm // Knowledge-Based Systems. 2007. Vol. 20, No. 4. P. 329–335. DOI: j.knosys.2006.08.005.
10. Duran A., Klemm M. The Intel Many Integrated Core Architecture // 2012 International Conference on High Performance Computing and Simulation, HPCS 2012, Madrid, Spain, July 2–6, 2012. 2012. P. 365–366. DOI: 10.1109/HPCSim.2012.6266938.
11. Han J., Pei J., Yin Y. Mining Frequent Patterns without Candidate Generation // Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16–18, 2000, Dallas, Texas, USA. P. 1–12. DOI: 10.1145/342009.335372.
12. IBM Quest Synthetic Data Generator. URL: https://ibmquestdatagen.sourceforge.io/ (дата обращения: 03.11.2018).
13. Kostenetskiy P., Safonov A. SUSU Supercomputer Resources // PCT'2016, International Scientific Conference on Parallel Computational Technologies, Arkhangelsk, Russia, March 29–31, 2016. CEUR Workshop Proceedings. Vol. 1576, CEUR-WS.org, 2016. P. 561–573.
14. Kumar P., Bhatt P., Choudhury R. Bitwise Dynamic Itemset Counting Algorithm // Proceedings of the ICCIC 2015, IEEE International Conference on Computational Intelligence and Computing Research, December 10–12, 2015, Madurai, India. 2015. P. 1–4. DOI: 10.1109/ICCIC.2015.7435752.

15. Li J., Fu A.W., Fahey P. Efficient Discovery of Risk Patterns in Medical Data // Artificial Intelligence in Medicine. 2009. Vol. 45, No. 1. P. 77–89. DOI: 10.1016/j.artmed.2008.07.008.

16. Lischner R. STL Reference: Containers, Iterators, and Algorithms. O'Reilly, 2003. 120 p.

17. Ordonez C., Ezquerra N.F., Santana C.A. Constraining and Summarizing Association Rules in Medical Data // Knowledge and Information Systems. 2006. Vol. 9, No. 3. P. 1–2. DOI: 10.1007/s10115-005-0226-5.

18. Ordonez C., Santana C.A., de Braal L. Discovering Interesting Association Rules in Medical Data // 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Dallas, Texas, USA, May 14, 2000. 2000. P. 78–85.

19. Paranjape-Voditel P., Deshpande U. A DIC-based Distributed Algorithm for Frequent Itemset Generation // Journal of Software. 2011. Vol. 6, No. 2. P. 306–313. DOI: 10.4304/jsw.6.2.306-313.

20. Pattanaprateep O., McEvoy M., Attia J., Thakkinstian A. Evaluation of Rational Nonsteroidal Anti-inflammatory Drugs and Gastro-protective Agents Use; Association Rule Data Mining Using Outpatient Prescription Patterns // BMC Medical Informatics and Decision Making. 2017. Vol. 17, No. 1. P. 96:1–96:7. DOI: 10.1186/s12911-017-0496-3.

21. Schlegel B., Karnagel T., Kiefer T., Lehner W. Scalable Frequent Itemset Mining on Many-core Processors // Proceedings of the 9th International Workshop on Data Management on New Hardware, DaMoN 2013, New York, NY, USA, June 24, 2013. 2013. P. 3. DOI: 10.1145/2485278.2485281.

22. Sokolinskaya I., Sokolinsky L. Revised Pursuit Algorithm for Solving Non-stationary Linear Programming Problems on Modern Computing Clusters with Manycore Accelerators // Supercomputing. RuSCDays 2016. Communications in Computer and Information Science. 2016. Vol. 687. P. 212–223. DOI: 10.1007/978-3-319-55669-7_17.

23. Zaki M.J. Scalable Algorithms for Association Mining // IEEE Transactions on Knowledge and Data Engineering. 2000. Vol. 12, No. 3. P. 372–390. DOI: 10.1109/69.846291.

24. Zymbler M. Accelerating Dynamic Itemset Counting on Intel Many-core Systems // Proceedings of the 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO'2017, Opatija, Croatia, May 22–26, 2017. IEEE, 2017. P. 1575–1580. DOI: 110.23919/MIPRO.2017.797363.

Цымблер Михаил Леонидович, к.ф.-м.н., доцент, кафедра системного программирования, Южно-Уральский государственный университет (национальный исследовательский университет) (Челябинск, Российская Федерация)

DOI: 10.14529/cmse190104

PARALLEL FREQUENT ITEMSET MINING ON THE INTEL MIC ACCELERATORS

© 2019 M.L. Zymbler

South Ural State University (pr. Lenina 76, Chelyabinsk, 454080 Russia)

E-mail: mzym@susu.ru

Received: 26.12.2018

Association rule mining is one of the basic problems of data mining, which supposes finding strong correlations between itemsets in large transaction database. Association rules are generated from frequent itemsets (itemset is frequent if its items frequent occur together in transactions). The DIC (Dynamic Itemset Counting) algorithm is modification of the classical Apriori algorithm of finding frequent itemsets. DIC tries to reduce the number of passes made over the transaction database while keeping the number of itemsets counted in a pass relatively low. The paper addresses the task of accelerating DIC on the Intel MIC (Many Integrated Core) systems in the case when the transaction database fits into the main memory. The paper presents a parallel implementation of DIC based on OpenMP technology and thread-level parallelism. We exploit the bit-based internal layout for transactions and itemsets. This technique simplifies the support count via logical bitwise operation, and allows for vectorization of such a step. Experiments with large synthetic and real databases showed good performance and scalability of the proposed algorithm.

Keywords: data mining, frequent itemset counting, OpenMP, Intel Many Integrated Core.

FOR CITATION

Zymbler M.L. Parallel Frequent Itemset Mining on the Intel MIC Accelerators. *Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering.* 2019. vol. 8, no. 1. pp. 54–70. (in Russian) DOI: 10.14529/cmse190104.

References

1. Voevodin V.V., Voevodin Vl.V. *Parallellnye vachisleniya* [Parallel Computing]. SPb: BHV-Petersburg, 2002. 608 p.

2. Agrawal R., Srikant R. Fast Algorithms for Mining Association Rules in Large Databases. VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago de Chile, Chile. 1994. pp. 487–499.

3. Bacon D.F., Graham S.L., Sharp O.J. Compiler Transformations for High-Performance Computing. ACM Computing Surveys. 1994. vol. 26, no. 4. pp. 345–420. DOI: 10.1145/197405.197406.

4. Borgelt C. Frequent Item Set Mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery. 2012. vol. 2, no. 6, pp. 437–456. DOI: 10.1002/widm.1074.

5. Brin S., Motwani R., Ullman J.D., Tsur S. Dynamic Itemset Counting and Implication Rules for Market Basket Data. SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13–15, 1997, Tucson, Arizona, USA. 1997. P. 255–264. DOI: 10.1145/253260.253325.

6. Burdick D., Calimlim M., Flannick J. *et al.* MAFIA: A Maximal Frequent Itemset Algorithm. IEEE Transactions on Knowledge and Data Engineering. 2005. vol. 17, no. 11. pp. 1490–1504. DOI: 10.1109/TKDE.2005.183.

7. Chang Y., Chen J., Tsai Y. Mining Subspace Clusters from DNA Microarray Data Using Large Itemset Techniques. Journal of Computational Biology. 2009. vol. 16, no. 5. pp. 745–768. DOI: 10.1089/cmb.2008.0161.

8. Cheung D.W., Hu K., Xia S. An Adaptive Algorithm for Mining Association Rules on Shared-Memory Parallel Machines. Distributed and Parallel Databases. 2001. vol. 9, no. 2. pp. 99–132. DOI: 10.1023/A:1018951022124.

9. Dong J., Han M. BitTableFI: An Efficient Mining Frequent Itemsets Algorithm. Knowledge-Based Systems. 2007. vol. 20, no. 4. pp. 329–335. DOI: j.knosys.2006.08.005.

10. Duran A., Klemm M. The Intel Many Integrated Core Architecture. 2012 International Conference on High Performance Computing and Simulation, HPCS 2012, Madrid, Spain, July 2–6, 2012. 2012. pp. 365–366. DOI: 10.1109/HPCSim.2012.6266938.

11. Han J., Pei J., Yin Y. Mining Frequent Patterns without Candidate Generation. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16–18, 2000, Dallas, Texas, USA. pp. 1–12. DOI: 10.1145/342009.335372.

12. IBM Quest Synthetic Data Generator. URL: https://ibmquestdatagen.sourceforge.io/ (accessed: 03.11.2018).

13. Kostenetskiy P., Safonov A. SUSU Supercomputer Resources. PCT'2016, International Scientific Conference on Parallel Computational Technologies, Arkhangelsk, Russia, March 29–31, 2016. CEUR Workshop Proceedings. vol. 1576, CEUR-WS.org, 2016. pp. 561–573.

14. Kumar P., Bhatt P., Choudhury R. Bitwise Dynamic Itemset Counting Algorithm. Proceedings of the ICCIC 2015, IEEE International Conference on Computational Intelligence and Computing Research, December 10–12, 2015, Madurai, India. 2015. pp. 1–4. DOI: 10.1109/ICCIC.2015.7435752.

15. Li J., Fu A.W., Fahey P. Efficient Discovery of Risk Patterns in Medical Data. Artificial Intelligence in Medicine. 2009. vol. 45, no. 1. pp. 77–89. DOI: 10.1016/j.artmed.2008.07.008.

16. Lischner R. STL Reference: Containers, Iterators, and Algorithms. O'Reilly, 2003. 120 p.

17. Ordonez C., Ezquerra N.F., Santana C.A. Constraining and Summarizing Association Rules in Medical Data. Knowledge and Information Systems. 2006. vol. 9, no. 3. pp. 1–2. DOI: 10.1007/s10115-005-0226-5.

18. Ordonez C., Santana C.A., de Braal L. Discovering Interesting Association Rules in Medical Data. 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Dallas, Texas, USA, May 14, 2000. 2000. pp. 78–85.

19. Paranjape-Voditel P., Deshpande U. A DIC-based Distributed Algorithm for Frequent Itemset Generation. Journal of Software. 2011. vol. 6, no. 2. pp. 306–313. DOI: 10.4304/jsw.6.2.306-313.

20. Pattanaprateep O., McEvoy M., Attia J., Thakkinstian A. Evaluation of Rational Nonsteroidal Anti-inflammatory Drugs and Gastro-protective Agents Use; Association Rule Data Mining Using Outpatient Prescription Patterns. BMC Medical Informatics and Decision Making. 2017. vol. 17, no. 1. pp. 96:1–96:7. DOI: 10.1186/s12911-017-0496-3.

21. Schlegel B., Karnagel T., Kiefer T., Lehner W. Scalable Frequent Itemset Mining on Many-core Processors. Proceedings of the 9th International Workshop on Data Management on New Hardware, DaMoN 2013, New York, NY, USA, June 24, 2013. 2013. pp. 3. DOI: 10.1145/2485278.2485281.

22. Sokolinskaya I., Sokolinsky L. Revised Pursuit Algorithm for Solving Non-stationary Linear Programming Problems on Modern Computing Clusters with Manycore Accelerators. Supercomputing. RuSCDays 2016. Communications in Computer and Information Science. 2016. vol. 687. pp. 212–223. DOI: 10.1007/978-3-319-55669-7_17.

23. Zaki M.J. Scalable Algorithms for Association Mining. IEEE Transactions on Knowledge and Data Engineering. 2000. vol. 12, no. 3. pp. 372–390. DOI: 10.1109/69.846291.

24. Zymbler M. Accelerating Dynamic Itemset Counting on Intel Many-core Systems. Proceedings of the 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO'2017, Opatija, Croatia, May 22–26, 2017. IEEE, 2017. pp. 1575–1580. DOI: 110.23919/MIPRO.2017.797363.