На правах рукописи

ЦЫМБЛЕР Михаил Леонидович

ИНТЕЛЛЕКТУАЛЬНЫЙ АНАЛИЗ ДАННЫХ В СУБД

05.13.11 — математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Автореферат
диссертации на соискание ученой степени
доктора физико-математических наук

Челябинск — 2019

Работа выполнена на кафедре системного программирования
ФГАОУ ВО «Южно-Уральский государственный университет
(национальный исследовательский университет)»

Научный консультант: СОКОЛИНСКИЙ Леонид Борисович,
доктор физ.-мат. наук, профессор,
проректор по информатизации ФГАОУ ВО
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
(г. Челябинск)

Официальные оппоненты: ВОЕВОДИН Владимир Валентинович,
доктор физ.-мат. наук, член-корреспондент РАН,
директор Научно-исследовательского
вычислительного центра ФГБОУ ВО «Московский
государственный университет
имени М.В. Ломоносова» (г. Москва)

КУЗНЕЦОВ Сергей Дмитриевич,
доктор техн. наук, профессор,
главный научный сотрудник ФГБУН «Институт
системного программирования
имени В.П. Иванникова РАН» (г. Москва)

ЕЛИЗАРОВ Александр Михайлович,
доктор физ.-мат. наук, профессор,
профессор кафедры программной инженерии
ФГАОУ ВО «Казанский (Приволжский)
федеральный университет» (г. Казань)

Ведущая организация: ФГАОУ ВО «Санкт-Петербургский национальный
исследовательский университет информационных
технологий, механики и оптики» (г. Санкт-Петербург)

Защита состоится 12 февраля 2020 г. в 11:00 часов на заседании диссертационного совета Д 212.298.18 при Южно-Уральском государственном университете по адресу: 454080, г. Челябинск, пр. Ленина, 76, ауд. 1001.

С диссертацией можно ознакомиться в библиотеке Южно-Уральского государственного университета и на сайте:
https://www.susu.ru/ru/dissertation/d-21229818/cymbler-mihail-leonidovich.

Автореферат разослан «_____» _____ 20_____ г.

И.о. ученого секретаря дис. совета В.В. Мокеев

Общая характеристика работы

Актуальность темы. *Интеллектуальный анализ данных* направлен на извлечение из данных доступных для понимания знаний, необходимых для принятия решений в различных сферах человеческой деятельности. В современном информационном обществе феномен *Больших данных* оказывает существенное влияние на технологии обработки данных. Эксперты компании IDC прогнозируют, что объем мировых данных, удваиваясь каждые два года, к 2020 г. достигнет 44 Зеттабайт (44 триллиона Гигабайт). В 2016 г. в Бекманском отчете ведущие мировые специалисты в области технологий обработки данных констатировали, что для перехода к умному (управляемому данными) обществу необходимо организовать интегрированный процесс от получения данных до извлечения из них знаний, в рамках которого алгоритмы анализа данных полноценно масштабируются на современных многопроцессорных и многоядерных аппаратных платформах.

Процессы очистки и структурирования Больших данных приводят к образованию сверхбольших баз и хранилищ данных. Один из наиболее авторитетных ученых в области баз данных М. Стоунбрейкер (Michael Stonebraker) указывает, что для решения проблем обработки сверхбольших данных необходимо использовать технологии систем управления базами данных (СУБД). Несмотря на появление большого количества NoSQL СУБД, основным инструментом управления базами данных попрежнему остаются *реляционные СУБД*.

Одним из перспективных направлений развития реляционных СУБД является внедрение в них средств интеллектуального анализа данных. *Интеграция интеллектуального анализа данных в реляционные СУБД* позволяет как избежать накладных расходов по экспорту анализируемых данных из хранилища и импорту результатов анализа обратно в хранилище, так и использовать при анализе данных системные сервисы, заложенные в архитектуре СУБД. Однако в этой области имеется целый ряд нерешенных научных задач, связанных с применением высокопроизводительных вычислительных систем для параллельной обработки и анализа данных.

На сегодняшний день *параллельные СУБД* на основе концепции *фрагментного параллелизма* являются одним из наиболее эффективных подходов к обработке сверхбольших баз данных. Параллельная СУБД организует разбиение таблиц базы данных на горизонтальные фрагменты, которые распределяются по узлам *вычислительного кластера* и обра-

батываются независимо. Коммерческие параллельные СУБД (Teradata, IBM DB2 Parallel Edition и др.) имеют высокую стоимость и ориентированы на специфические аппаратно-программные платформы. *Свободные СУБД* (MySQL, PostgreSQL и др.) зарекомендовали себя надежной альтернативой проприетарным решениям и предоставляют открытый исходный код, который может быть модернизирован любым разработчиком. Это делает *актуальной* задачу построения параллельной СУБД путем внедрения фрагментного параллелизма в код свободной СУБД (при этом модернизация производится без масштабных изменений, что было бы равнозначно разработке «с нуля»).

Кроме того, разработка отечественных параллельных СУБД и систем анализа данных на основе решений с открытым исходным кодом является важным фактором обеспечения технологической независимости Российской Федерации в области информационных технологий[1].

Важной тенденцией развития современной процессорной техники является увеличение количества вычислительных ядер. Многоядерные ускорители и сопроцессоры, такие как Intel Many Integrated Core (MIC), IBM Cell, NVIDIA GPU (Graphic Processing Units) и FPGA (Field Programmable Gate Arrays), включают в себя сотни ядер, поддерживающих векторную обработку данных, и опережают классические центральные процессоры по производительности на определенных классах приложений. Однако разработка аналитических алгоритмов для ускорителей требует новых схем организации вычислений и хранения данных в памяти, допускающих эффективную векторизацию.

В соответствии с вышеизложенным является *актуальной* задача разработки новых подходов к интеграции интеллектуального анализа в реляционные системы баз данных, а также разработка и реализация в рамках предлагаемых подходов новых параллельных алгоритмов интеллектуального анализа данных для кластерных вычислительных систем с узлами на базе современных многоядерных ускорителей.

Цель и задачи исследования. *Цель* данной работы состояла в разработке комплекса масштабируемых методов и параллельных алгоритмов для создания программной платформы интеллектуального анализа данных средствами СУБД с открытым кодом. Данная цель предполагает решение следующих *задач*.

[1]Постановление Правительства РФ № 1236 от 16 ноября 2015 г. «Об установлении запрета на допуск программного обеспечения, происходящего из иностранных государств, для целей осуществления закупок для обеспечения государственных и муниципальных нужд».

1. Разработать методы и алгоритмы для внедрения фрагментного параллелизма в свободную последовательную реляционную СУБД. Проверить эффективность предложенных решений на СУБД PostgreSQL.
2. Разработать методы и алгоритмы для внедрения интеллектуального анализа данных в параллельную СУБД для современных многопроцессорных платформ с многоядерными ускорителями.
3. Разработать параллельные алгоритмы решения задач кластеризации, поиска шаблонов и анализа временных рядов средствами параллельной реляционной СУБД.
4. Провести вычислительные эксперименты с синтетическими и реальными данными, подтверждающие эффективность предложенных методов и алгоритмов.

Методы исследования. Проведенные в работе исследования базируются на реляционной модели данных, методах интеллектуального анализа данных и теории временных рядов. При разработке программных комплексов применялись методы объектно-ориентированного проектирования и язык UML, а также методы системного, модульного и объектно-ориентированного программирования. В реализации параллельных алгоритмов использованы методы параллельного программирования для общей и распределенной памяти на основе стандартов MPI и OpenMP, а также методы параллельных систем баз данных.

Научная новизна работы заключается в следующем:

1. Разработан оригинальный метод интеграции интеллектуального анализа данных в реляционную СУБД на основе пользовательских функций, инкапсулирующих параллельные аналитические алгоритмы для современных многоядерных процессоров.
2. Разработан оригинальный метод интеграции фрагментного параллелизма в последовательную свободную СУБД, не требующий масштабных изменений в исходном коде.
3. Впервые разработаны параллельные алгоритмы анализа временных рядов для вычислительных кластеров с многоядерными ускорителями.
4. Разработаны новые параллельные алгоритмы кластеризации данных сверхбольших объемов для параллельной реляционной СУБД.

5. Разработаны новые параллельные алгоритмы поиска частых наборов и кластеризации данных для многоядерных ускорителей.

Теоретическая ценность диссертационной работы состоит в следующем. В работе предложены методы, архитектурные решения и алгоритмы, позволяющие интегрировать параллельную обработку и анализ данных в последовательные реляционные СУБД: предложен подход к интеграции интеллектуального анализа данных в СУБД, предполагающий встраивание в СУБД аналитических алгоритмов, которые инкапсулируют параллельное исполнение на современных многоядерных ускорителях; предложен подход к разработке параллельной СУБД, предполагающий интеграцию фрагментного параллелизма в СУБД с открытым исходным кодом. В работе предложены параллельные алгоритмы решения различных задач интеллектуального анализа данных (кластеризация, поиск частых наборов, поиск похожих подпоследовательностей и диссонансов во временных рядах) для современных многоядерных ускорителей, обеспечивающих ускорение, близкое к линейному.

Практическая ценность работы заключается в том, что предложенные методы интеграции параллелизма применены к свободной СУБД PostgreSQL и разработаны прототипы библиотеки интеллектуального анализа данных и прототип параллельной СУБД PargreSQL. Результаты, полученные в работе, могут быть использованы в создании коммерческих и свободно распространяемых программных продуктов, ориентированных на параллельную обработку и анализ данных с использованием свободной реляционной СУБД.

Степень достоверности результатов. Разработанные методы и алгоритмы подтверждены вычислительными экспериментами над реальными и синтетическими данными, проведенными в соответствии с общепринятыми стандартами.

Апробация работы. Основные положения диссертационной работы, разработанные методы, алгоритмы и результаты вычислительных экспериментов докладывались на следующих международных и всероссийских научных конференциях:

— ПаВТ'2019: «Параллельные вычислительные технологии 2019» (2–4 апреля 2019 г., Калининград);

— DAMDID'2018: 20th International Conference on Data Analytics and Management in Data Intensive Domains (9–12 октября 2018 г., Москва);

— ПаВТ'2018: «Параллельные вычислительные технологии 2018» (3–5 апреля 2018 г., Ростов-на-Дону);

— DAMDID'2017: 19th International Conference on Data Analytics and Management in Data Intensive Domains (10–13 октября 2017 г., Москва);

— ПаВТ'2016: «Параллельные вычислительные технологии 2016» (29–31 марта 2016 г., Архангельск);

— SISAP'2015: 18th International Conference on Similarity Search and Applications (12–14 октября 2015 г., Глазго, Великобритания);

— AICT'2015: 9th International Conference on Application of Information and Communication Technologies (14–16 октября 2015 г., Ростов-на-Дону);

— «Суперкомпьютерные дни в России 2015» (28–29 сентября 2015 г., Москва);

— ADBIS'2015: 19th East-European Conference on Advances in Databases and Information Systems (8–11 сентября 2015 г., Пуатье, Франция);

— ПаВТ'2015: «Параллельные вычислительные технологии 2015» (31 марта – 2 апреля 2015 г., Екатеринбург);

— «Научный сервис в сети Интернет 2014: многообразие суперкомпьютерных миров» (22–27 сентября 2014 г., Новороссийск);

— ПаВТ'2014: «Параллельные вычислительные технологии 2014» (1–3 апреля 2014 г., Ростов-на-Дону);

— DEXA'2013: 24th International Conference on Database and Expert Systems Applications (26–30 августа 2013 г., Прага, Чехия);

— ADBIS'2013: 17th East-European Conference on Advances in Databases and Information Systems (1–4 сентября 2013 г., Генуя, Италия);

— «Научный сервис в сети Интернет 2013: все грани параллелизма» (23–28 сентября 2013 г., Новороссийск);

— ПаВТ'2011: «Параллельные вычислительные технологии 2011» (29–31 марта 2011 г., Москва);

— ПаВТ'2010: «Параллельные вычислительные технологии 2010» (30 марта – 1 апреля 2010 г., Уфа).

Публикации. По теме диссертации опубликована 21 печатная работа. Работы [1–12] опубликованы в журналах, включенных ВАК в перечень изданий, в которых должны быть опубликованы основные результаты диссертаций на соискание ученой степени доктора и кандидата наук. Работы [13–21] опубликованы в изданиях, индексируемых в Scopus и Web of Science. В рамках выполнения диссертационной работы получено три свидетельства Роспатента об официальной регистрации программ для ЭВМ и баз данных [22–24].

Личный вклад. Все совместные публикации, кроме работы [2], выполнены М.Л. Цымблером в соавторстве с аспирантами и студентами, у которых он являлся научным руководителем. В статье [1] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, обзор работ, описание методов внедрения фрагментного параллелизма и архитектуры СУБД PargreSQL, заключение, стр. 18–28, 30), К.С. Пану принадлежит раздел 4 (описание результатов экспериментов, стр. 28–30). В статье [2] М.Л. Цымблеру принадлежит описание архитектуры СУБД PargreSQL (стр. 57), Л.Б. Соколинскому принадлежит вводная часть (стр. 56), К.С. Пану принадлежит описание результатов тестирования СУБД PargreSQL (стр. 58). В статье [3] М.Л. Цымблеру принадлежат введение, разделы 2–4 и заключение (описание архитектуры и принципов реализации СУБД PargreSQL, стр. 112, 113–118), К.С. Пану принадлежит раздел 1 (обзор работ, стр. 112–113). В статье [19] М.Л. Цымблеру принадлежат разделы 1–3, 5–6 (введение, описание принципов проектирования и реализации СУБД PargreSQL, обзор работ, заключение, стр. 153–162, 163), К.С. Пану принадлежит раздел 4 (описание результатов экспериментов, стр. 162). В статье [20] М.Л. Цымблеру принадлежит разделы 1–4, 6 (введение, обзор работ, описания СУБД PargreSQL и алгоритма, заключение, стр. 388–396, 397), К.С. Пану принадлежит раздел 5 (описание результатов экспериментов, стр. 396–397). В статье [6] М.Л. Цымблеру принадлежат введение, разделы 1–3, 5 (формальная постановка задачи, обзор работ и описание алгоритма, заключение, стр. 48–54, 57), К.С. Пану принадлежит раздел 4 (описание результатов экспериментов, стр. 54–56). В статье [4] М.Л. Цымблеру принадлежат описания алгоритма и результатов экспериментов (стр. 43), А.В. Мовчану принадлежит вводная часть (стр. 42). В статье [16] М.Л. Цымблеру принадлежит разделы 1–3, 5 (введение, формальная постановка задачи, описание алгоритма, заключение, стр. 65–69, 71), А.В. Мовчану принадлежит раздел 4 (описание результатов экспериментов, стр. 69–71). В статье [8] М.Л. Цымблеру принадлежат разделы 1–4 (введение, формальная постановка задачи, описание

алгоритма, заключение, стр. 33–36, 40), Я.А. Краевой принадлежит раздел 4 (описание результатов экспериментов, стр. 36–40). В статье [15] М.Л. Цымблеру принадлежат разделы 1–4, 6 (введение, формальная постановка задачи, обзор работ, описание алгоритма, заключение, стр. 149–156, 161–162), Я.А. Краевой принадлежит раздел 5 (описание результатов экспериментов, стр. 156–161). В статье [5] М.Л. Цымблеру принадлежат разделы 1–3 (введение, формальная постановка задачи, описание алгоритма, заключение, стр. 47–50, 51), Р.М. Миниахметову принадлежит раздел 4 (описание результатов экспериментов, стр. 50–51). В статье [18] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, обзор работ, описание подхода к интеграции анализа данных в СУБД, заключение, стр. 230–238, 242–243), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 238–242). В статье [7] М.Л. Цымблеру принадлежат введение, разделы 1–2, заключение (обзор работ, описание алгоритма, стр. 65–71, 77–78), Т.В. Речкалову принадлежит раздел 3 (описание результатов экспериментов, стр. 71–77). В статье [21] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, обзор работ, описание алгоритма, заключение, стр. 200–206, 213–214), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 206–213). В статье [10] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, формальная постановка задачи и обзор работ, описание алгоритма, заключение, стр. 104–109, 111, 113), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 110, 112). В статье [17] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, формальная постановка задачи, описание алгоритма, обзор работ, заключение, стр. 413–415, 416–417), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 415–416).

Структура и объем работы Диссертация состоит из введения, пяти глав, заключения и библиографии. Объем диссертации составляет 260 страниц, объем библиографии — 274 наименования.

Содержание работы

Во введении приводится обоснование актуальности темы и степень ее разработанности; формулируются цели и задачи исследования; раскрываются новизна, теоретическая и практическая значимость полученных результатов; формулируется методологическая основа диссертационного исследования; дается обзор содержания диссертации.

В первой главе, «Интеллектуальный анализ данных», рассмотрены типовые задачи анализа данных и дается обзор методов и подходов к интеграции интеллектуального анализа данных в реляционные СУБД. Рассматриваются следующие типовые задачи интеллектуального анализа данных: кластеризация, поиск шаблонов и анализ временных рядов[2].

Кластеризация заключается в разбиении множества объектов сходной структуры на заранее неизвестные группы (кластеры) в зависимости от похожести свойств объектов и формально определяется следующим образом. Пусть заданы конечные множества: $X = \{x_1, \ldots, x_n\}$ — множество объектов d-мерного метрического пространства, для которых задана функция расстояния $\rho(x_i, x_j)$, и $C = \{c_1, \ldots, c_k\}$ — набор уникальных идентификаторов кластеров, где $1 < k \ll n$. *Алгоритм (четкой) кластеризации* представляет собой функцию $\alpha : X \to C$, которая каждому объекту назначает уникальный идентификатор кластера. Алгоритм кластеризации выполняет разбиение множества X на непересекающиеся непустые подмножества (*кластеры*) таким образом, чтобы каждый кластер состоял из объектов, близких по метрике ρ, а объекты разных кластеров существенно отличались. *Алгоритм нечеткой кластеризации* позволяет одному и тому же объекту принадлежать одновременно всем кластерам, но с различной степенью принадлежности.

Разбиение графа представляет собой обособленную задачу кластеризации, предполагающую распределение вершин графа по кластерам на основе весов ребер, трактуемых как расстояния между соседними вершинами, и формально определяется следующим образом. Пусть имеется граф $G = (N, E)$, где N — множество взвешенных вершин, E — множество взвешенных ребер, и дано целое число $p > 0$. Тогда *p-разбиением* графа G называются подмножества вершин N_1, …, N_p ($N_i \subseteq N$), для которых выполняются следующие условия:

1) $\cup_{i=1}^{p} N_i = N$ и $N_i \cap N_j = \varnothing \ \forall i \neq j$;
2) $w(N_i) \approx \frac{w(N)}{p}$;
3) величина *разреза* $W_{cut} = \sum_{(u,v) \in E_{cut}} w(u, v)$ минимальна, где $E_{cut} := \{(u, v) \in E \mid u \in N_i,\ v \in N_j,\ i \neq j\}$.

Поиск шаблонов (ассоциативных правил) заключается в нахождении часто повторяющихся зависимостей в заданном конечном наборе объек-

[2]В приводимых ниже формальных определениях и далее в изложении используются обозначения, соответствующие традициям, устоявшимся в теории решения каждой из указанных задач. В случае коллизии обозначений контекстом полагается соответствующая задача.

тов. Поиск шаблонов разбивается на две последовательно выполняемые задачи: поиск всех частых наборов и генерация на их основе устойчивых ассоциативных правил. В данном исследовании рассматривается *задача поиска всех частых наборов*, формально определяемая следующим образом. Пусть дано конечное множество *объектов* $\mathcal{I} = \{i_1, \ldots, i_m\}$, любое непустое его подмножество называют *набором*. Набор из k объектов ($1 \leqslant k \leqslant m$) называют *$k$-набором*. Дано множество *транзакций* $\mathcal{D} = \{T_1, \ldots, T_n\}$, 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:

$$supp(I) := \frac{|\{T \in \mathcal{D} \mid I \subseteq T.I\}|}{|\mathcal{D}|}.$$

Набор с поддержкой не ниже заданного порога $minsup$, который является параметром задачи, называют *частым*, иначе набор называют *редким*. Решением задачи является множество $\mathcal{L} := \cup_{k=1}^{k_{max}} \mathcal{L}_k$, где $\mathcal{L}_k$ — множество всех частых k-наборов и k_{max} — максимальная мощность частого набора.

Временной ряд T представляет собой хронологически упорядоченную последовательность числовых значений с отметками времени: $T = (t_1, \ldots, t_m)$, где $t_i \in \mathbb{R}$ и $|T| = m$ — длина ряда. *Подпоследовательность* $T_{i,n}$ — подмножество T из n элементов, взятых без пропусков, начиная с позиции i: $T_{i,n} = (t_i, t_{i+1}, \ldots, t_{i+n-1}), 1 \leqslant n \ll m, 1 \leqslant i \leqslant N$, где $N := m - n + 1$. Множество всех подпоследовательностей длины n ряда T обозначается как S_T^n.

Поиск похожих подпоследовательностей предполагает нахождение участков ряда, форма которых похожа на заданный ряд существенно меньшей длины. Подпоследовательность $T_{i,n} \in S_T^n$ является самой похожей на *поисковый запрос* Q ($|Q| = n \ll |T|$), если $\forall j \ DTW(Q, T_{i,n}) \leqslant DTW(Q, T_{j,n}),\ 1 \leqslant i, j \leqslant N$, где мера схожести *DTW (Dynamic Time Warping)* определяется следующим образом. Пусть заданы поисковый запрос $Q = (q_1, \ldots, q_n)$, подпоследовательность $C = (c_1, \ldots, c_n)$ и целое число $1 \leqslant r < n$ *(ограничение Сако—Чиба)*. Тогда

$$DTW(Q, C) := d(n, n),$$

$$d(i, j) := (q_i - c_j)^2 + \min \begin{cases} d(i-1, j) \\ d(i, j-1) \\ d(i-1, j-1) \end{cases},$$

$$d(0, 0) = 0,\ d(i, 0) = d(0, j) = \infty,\ j - r \leqslant i \leqslant j + r.$$

Поиск диссонансов предполагает нахождение аномальных (наиболее непохожих на остальные) участков временного ряда и формально определяется следующим образом. Подпоследовательности $T_{i,n}$ и $T_{j,n}$ называются *непересекающимися*, если $|i-j| \geqslant n$. Подпоследовательность, не пересекающаяся с подпоследовательностью C, обозначается как M_C. Подпоследовательность D является *диссонансом* ряда T, если $\forall C, M_C \in T$ $\min\big(\mathrm{ED}(D, M_D)\big) > \min\big(\mathrm{ED}(C, M_C)\big)$, где ED — расстояние Евклида.

Далее в первой главе рассматриваются известные способы распараллеливания СУБД, методы интеграции интеллектуального анализа данных в СУБД и дается обзор работ, наиболее близких к теме диссертационного исследования.

Во второй главе, «Кластеризация и поиск шаблонов» рассмотрены следующие задачи: кластеризация данных сверхбольшого объема, которые не могут быть размещены в оперативной памяти, в реляционных СУБД и параллельный поиск шаблонов на многоядерных ускорителях. Предложены алгоритмы *dbParGraph* кластеризации графа и *pgFCM* нечеткой кластеризации данных для параллельной СУБД на основе фрагментного параллелизма. Разработаны параллельные алгоритмы *PDIC* и *DDCapriori* поиска частых наборов для многоядерных ускорителей (архитектур Intel MIC и IBM Cell соответственно). Представлены результаты вычислительных экспериментов, исследующих эффективность разработанных алгоритмов.

Алгоритм dbParGraph предназначен для кластеризации сверхбольших графов социальных сетей в параллельной СУБД. Разработана схема базы данных, в которой исходные, промежуточные и результирующие данные алгоритма представляются в виде реляционных таблиц (см. табл. 1). Данные графа хранятся в реляционной таблице, представляющей список ребер, где для каждого ребра графа хранятся его вес и уникальные идентификаторы (номера) концевых вершин. Каждая таблица базы данных подвергается горизонтальной фрагментации. На каждом узле кластера набор фрагментов таблиц обрабатывается отдельно соответствующим экземпляром параллельной СУБД.

Алгоритм *dbParGraph* состоит из следующих стадий: поиск и консолидация сообществ, улучшение фрагментации ребер, огрубление, начальное разбиение и уточнение. *Поиск сообществ* выполняется в параллельной СУБД и предполагает нахождение таких подмножеств вершин данного графа, в каждом из которых вершины плотно связаны между собой и редко связаны с другими частями графа. Вводятся меры сходства соседних вершин графа и вхождения вершины графа в сообщество, опре-

Табл. 1. Схема базы данных алгоритма *dbParGraph*

№ п/п	Таблица	Семантика	Поля таблицы
1	G	Исходный граф	A, B: номера концевых вершин ребра W: вес ребра
2	$VERTEX$	Сведения о принадлежности вершин сообществам	A: номер вершины C: номер сообщества данной вершины
3	$GRAPH$	Исходный граф с улучшенной фрагментацией ребер	A, B: номера концевых вершин ребра W: вес ребра F: номер фрагмента
4	$MATCH$	Максимальное паросочетание исходного графа	A, B: номера концевых вершин ребра F: номер фрагмента
5	$COARSE\_GRAPH$	Огрубленный граф	A, B: номера концевых вершин ребра W: вес ребра F: номер фрагмента
6	$COARSE\_PARTITIONS$	Начальное разбиение огрубленного графа	A: номер вершины, P: цвет вершины
7	$PARTITIONS$	Разбиение исходного графа	A: номер вершины, P: цвет вершины, G: значение функции выгоды

деляемые следующим образом. Пусть имеется вершина $v \in N$, обозначим множество соседних с ней вершин как $\mathcal{N}_v$. Тогда *сходство (affinity)* вершины v и соседней с ней вершины $u \in \mathcal{N}_v$ определяется функцией

$$affinity(v,u) := \frac{w(v,u)}{\sum_{i \in \mathcal{N}_v} w(v,i)}. \tag{1}$$

Вершина $v \in N$ имеет *метку* (уникальный числовой идентификатор) сообщества, которому она принадлежит, обозначаемую как $\mathcal{L}_v$. Тогда *степень вхождения вершины (degree of membership)* v в сообщество C определяется функцией

$$d(v,C) := \frac{\sum_{u \in \mathcal{N}_v \wedge \mathcal{L}_u = C} aff(v,u)}{\sum_{u \in \mathcal{N}_v} aff(v,u)}. \tag{2}$$

Метка каждой вершины графа инициализируется значением номера этой вершины графа. Затем итеративно производится следующая операция «распространения меток»: метка каждой вершины заменяется на метку, которую имеет соседняя вершина с наибольшей степенью вхождения в сообщество среди всех вершин, соседних с данной. Если таких меток несколько, выбирается метка с наименьшим идентификатором. Распространение меток направлено на то, чтобы каждая вершина вошла в сообщество (подграф), в котором количество его внутренних ребер, инцидентных вершине, было бы минимальным. Процесс распространения меток завершается по достижении стабильного состояния вершин. Стабильное состояние считается достигнутым, если когда доля вершин графа, сохранивших свои метки, составляет не менее δ $(0 < \delta \leqslant 1)$, где число δ — наперед задаваемый параметр алгоритма.

Количество сообществ в сверхбольшом социальном графе, как правило, будет существенно превышать число вычислительных узлов кластерной системы. В силу этого на стадии *консолидации* выполняется укрупнение найденных сообществ посредством объединения двух сообществ в одно, если имеется хотя бы одно соединяющее их ребро. Для объединения выбираются два сообщества, которые имеют наименьшее среди остальных пар сообществ суммарное количество принадлежащих им и соединяющих их ребер. Процесс укрупнения выполняется, пока количество сообществ не окажется равным количеству вычислительных узлов кластерной системы.

На следующей стадии выполняется *улучшение фрагментации ребер графа*. Ребро, концевые вершины которого принадлежат одному и тому же сообществу, назначается для обработки в один и тот же фрагмент базы данных. Для балансировки размеров фрагментов каждое ребро с концевыми вершинами из разных сообществ назначается для обработки в тот из соответствующих фрагментов, где количество ребер меньше.

Стадия *огрубления* выполняется в параллельной СУБД и состоит из двух последовательно выполняемых шагов: поиск и стягивание. На шаге *поиска* осуществляется нахождение максимального паросочетания исходного графа, которое имеет вес, близкий к максимальному. На шаге *стягивания* осуществляется удаление ребер, найденных на шаге поиска. Концы удаляемого ребра заменяются одной вершиной, петли удаляются. Кратные ребра преобразуются в одно ребро, вес которого равен сумме весов кратных ребер. Огрубление повторяется многократно, чтобы редуцировать количество ребер исходного граф до приемлемого значения, при котором граф может быть целиком размещен в оперативной памяти.

На стадии *начального разбиения* огрубленный граф подается на вход стороннего алгоритма, который выполняет начальное разбиение графа в оперативной памяти. Результат начального разбиения импортируется в базу данных в виде реляционной таблицы.

Стадия *уточнения* выполняется в параллельной СУБД и состоит из трех последовательно выполняемых шагов: окрашивание, оценка качества разбиения и улучшение разбиения.

```
INSERT INTO PARTITIONS
  SELECT A, P
  FROM COARSE_PARTITIONS
  UNION
  SELECT MATCH.B, COARSE_PARTITIONS.P
  FROM MATCH, COARSE_PARTITIONS
  WHERE MATCH.A = COARSE_PARTITIONS.A;
```

Рис. 1. Реализация шага окрашивания в алгоритме *dbPargraph*

На шаге *окрашивания* (см. рис. 1) концам стянутых ребер исходного графа присваивается цвет соответствующей вершины огрубленного графа.

```
SELECT
  PARTITIONS.A,
  PARTITIONS.P,
  SUM(subgains.Gain) as Gain
FROM
  PARTITIONS left join (
    SELECT
      GRAPH.A, GRAPH.B,
      case when ap.P = bp.P then
        -GRAPH.W
      else
        GRAPH.W
      end as Gain
    FROM
      GRAPH left join PARTITIONS as ap on GRAPH.a = ap.A
        left join PARTITIONS as bp on GRAPH.b = bp.A
  ) as subgains
  on PARTITIONS.A = subgains.A or PARTITIONS.A = subgains.B
GROUP BY PARTITIONS.A, PARTITIONS.P
```

Рис. 2. Реализация шага подсчета функции выгоды в алгоритме *dbPargraph*

На шаге *оценки качества разбиения* (см. рис. 2) для каждой вершины графа, полученного на предыдущем шаге, вычисляется *функция выгоды*:

$$
\begin{aligned}
\mathrm{g}(v) &:= \mathrm{ext}(v) - \mathrm{int}(v), \\
\mathrm{ext}(v) &:= \sum_{(v,u)\in E \wedge P(v)\neq P(u)} w(v,u), \\
\mathrm{int}(v) &:= \sum_{(v,u)\in E \wedge P(v)= P(u)} w(v,u).
\end{aligned}
\tag{3}
$$

Функция $P : E \to \{1, \ldots, p\}$ возвращает номер подграфа, к которому относится заданная вершина. Значение функции $\mathrm{g}(v)$ показывает выгоду от инверсии цвета вершины v: если $\mathrm{g}(v) > 0$, то цвет вершины необходимо изменить на противоположный.

```
SELECT *
FROM PARTITIONS
WHERE
   P = current AND
   G = (
      SELECT max(G)
      FROM PARTITIONS
      WHERE P = current)
LIMIT 1
INTO V
```

(a) Поиск вершины

```
UPDATE PARTITIONS
   SET G = G + W *
      (case when P = V.P then
         2
      else
         -2
      end)
FROM (
   SELECT
      case when A = V.A then
         B
      else
         A
      end,
      W
   FROM GRAPH
   WHERE B = V.A OR A = V.A)
   as neighbors
WHERE neighbors.A = PARTITIONS.A;
UPDATE PARTITIONS
   SET G = -G, P = 1-P
WHERE A=V.A;
```

(b) Инвертирование цвета вершины

Рис. 3. Реализация шага улучшения разбиения в алгоритме *dbPargraph*

На шаге *улучшения разбиения* (см. рис. 3) осуществляются поиск вершин с максимальным значением функции выгоды и инвертирование ее цвета (до тех пор, пока такие вершины существуют).

Эксперименты с алгоритмом *dbParGraph* на суперкомпьютере «Торнадо ЮУрГУ» со сверхбольшими социальными графами (граф Ваттса—Строгаца из 10 млн. вершин и 50 млн. ребер, граф R-MAT из 7.7 млн. вершин и 133 млн. ребер и др.) показали, что предложенный алгоритм опережает по быстродействию аналоги с учетом накладных расходов последних на экспорт анализируемых данных из базы данных, конверта-

цию в специализированный формат для систем разбиения графов и импорт результатов обратно в базу данных.

Алгоритм pgFCM (см. алг. 1) выполняет в параллельной СУБД нечеткую кластеризацию данных, объем которых существенно превышает доступную оперативную память. Алгоритм предполагает нахождение матрицы степеней принадлежности $U \subset \mathbb{R}^{n \times k}$, где число $u_{ij} \in \mathbb{R}$ показывает степень принадлежности объекта x_i кластеру j и выполняются следующие свойства:

$$\forall\, i, j \quad u_{ij} \in [0; 1], \quad \forall\, i \quad \sum_{j=1}^{k} u_{ij} = 1 \tag{4}$$

Нечеткое разбиение исходного множества объектов X по k кластерам достигается посредством минимизации целевой функции

$$J(X, k, m) := \sum_{i=1}^{n} \sum_{j=1}^{k} u_{ij}^{m} \rho^2(x_i, c_j), \tag{5}$$

где $m > 1$ — степень нечеткости целевой функции (параметр алгоритма).

Алг. 1 pgFCM(in таблица SH, m, ε, k; out таблица U)

1: Создать и инициализировать таблицы U, P, SV и др.
2: **repeat**
3: Вычислить координаты центроидов c_j. Обновить таблицу C
4: Вычислить расстояния от точек x_i до центроидов c_j. Обновить таблицу SD
5: Вычислить степени принадлежности ut_{ij}. Обновить таблицу UT
6: Обновить таблицы U, P
7: **until** $P.delta > \varepsilon$
8: **return** U

Алгоритм итеративно обновляет таблицу с данными о координатах центров кластеров C и таблицу с данными о степени принадлежности объектов кластерам U, используя следующие формулы:

$$\forall j, \ell \quad c_{j\ell} = \frac{\sum\limits_{i=1}^{n} u_{ij}^{m} \cdot x_{i\ell}}{\sum\limits_{i=1}^{n} u_{ij}^{m}}, \qquad u_{ij} = \sum_{t=1}^{k} \left(\frac{\rho(x_i, c_j)}{\rho(x_i, c_t)} \right)^{\frac{2}{1-m}} \tag{6}$$

Критерий останова вычислений имеет следующий вид:

$$\max_{ij} \{|u_{ij}^{(s+1)} - u_{ij}^{(s)}|\} \leqslant \varepsilon, \tag{7}$$

где $\varepsilon \in \mathbb{R}$ — параметр алгоритма.

Табл. 2. Схема базы данных алгоритма *pgFCM*

№ п/п	Таблица	Поля таблицы	Кол-во записей	Семантика
1	SH	$\underline{i}, x1, x2, \ldots, xd$	n	Исходное множество объектов
2	SV	$\underline{i, \ell}, val$	$n \cdot d$	«Вертикальное» представление исходного множества
3	C	$\underline{j, \ell}, val$	$k{\cdot}d$	Координаты центроидов
4	SD	$\underline{i, j}, dist$	$n{\cdot}k$	Расстояния между объектами x_i и центроидами c_j
5	U	$\underline{i, j}, val$	$n{\cdot}k$	Степени принадлежности объектов x_i кластерам j на шаге s
6	UT	$\underline{i, j}, val$	$n{\cdot}k$	Степени принадлежности объектов x_i кластерам j на шаге $s+1$
7	P	$d, k, n, \underline{s}, delta$	s	Значение критерия останова на текущей итерации

Разработана схема базы данных, в которой исходные, промежуточные и результирующие данные алгоритма представляются в виде реляционных таблиц (см. табл. 2). Каждая таблица базы данных подвергается горизонтальной фрагментации. Фрагменты базы данных обрабатываются параллельно и независимо на каждом узле вычислительного кластера соответствующим экземпляром параллельной СУБД.

Предложенное индексное представление матриц исходных данных в виде реляционных таблиц позволяет применить конструкции SQL, выполняющие построчные агрегатные вычисления (см. рис. 4).

Эксперименты с алгоритмом *pgFCM* на суперкомпьютере «Торнадо ЮУрГУ» со сверхбольшими наборами данных (набор HIGGS из 11 млн. элементов с 28 вещественными атрибутами, набор KDD99 из 4.9 млн. элементов с 41 вещественным атрибутом) показали, что предложенный алгоритм опережает по быстродействию аналоги с учетом накладных расходов последних на экспорт анализируемых данных из базы данных и импорт результатов обратно в базу данных.

Параллельный *алгоритм PDIC* поиска частых наборов (см. алг. 2) использует *битовое представление данных*. Вводится функция битовой

```
-- Вычисление центров кластеров
INSERT INTO C
  SELECT R.j, SV.l, sum(R.s * SV.val) / sum(R.s) AS val
  FROM (
    SELECT i, j, U.val^m AS s
    FROM U) AS R, SV
  WHERE R.i = SV.i
  GROUP BY j, l;

-- Вычисление расстояний между объектами и центрами кластеров
INSERT INTO SD
  SELECT i, j, sqrt(sum((SV.val - C.val)^2)) as dist
  FROM SV, C
  WHERE SV.l = C.l;
  GROUP BY i, j;

-- Вычисление степени принадлежности объектов кластерам
INSERT INTO UT
  SELECT i, j, SD.dist^(2.0^(1.0 - m)) * SD1.den AS val
  FROM (
    SELECT i, 1.0 / sum(dist^(2.0^(m - 1.0))) AS den
    FROM SD
    GROUP BY i) AS SD1, SD
  WHERE SD.i = SD1.i;
```

Рис. 4. Реализация вычислений в алгоритме *pgFCM*

маски $BitMask : \mathcal{I} \to \mathbb{N}$, которая для транзакции $T \subseteq \mathcal{D}$ (набора $I \subseteq \mathcal{I}$, соответственно) возвращает слово, где 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. Использование битовых масок наборов и битовой карты базы транзакций упрощает подсчет поддержки и обеспечивает векторизацию этой операции: проверка вхождения набора I в транзакцию T выполняется с помощью одной логической побитовой операции $BitMask(I)$ `AND` $BitMask(T)$=$BitMask(I)$, а ее циклическое выполнение векторизуется компилятором.

Поиск частых наборов осуществляется в $\mathcal{B}$, битовой карте множества транзакций $\mathcal{D}$, определяемой как $\mathcal{B} := \{BitMask(T_1), \ldots, BitMask(T_n)\}$. Обработка битовой карты выполняется блоками по M транзакций, где M — параметр алгоритма. Вводится функция-счетчик просмотренных транзакций для заданного набора $Count : \mathcal{I} \to \mathbb{N} \cup \{0\}$ и следующие множества наборов:

$$\begin{gathered} DASHED := Circle_{Dashed} \cup Box_{Dashed}, \\ Circle_{Dashed} := \{I \mid I \subseteq \mathcal{I} \wedge Count(I) < n \wedge supp(I) < minsup\}, \\ Box_{Dashed} := \{I \mid I \subseteq \mathcal{I} \wedge Count(I) < n \wedge supp(I) \geqslant minsup\}. \end{gathered} \tag{8}$$

Алг. 2 PDIC(IN $\mathcal{B}$, $minsup$, M; OUT $\mathcal{L}$)

1: $SOLID.init()$; $DASHED.init()$
2: **for** $i \in 0..m-1$ **do**
3: $\quad I.shape \leftarrow$ NIL; $I.mask \leftarrow SetBit(I.mask, i)$
4: $\quad I.stop \leftarrow 0$; $I.supp \leftarrow 0$; $I.k \leftarrow 1$
5: $\quad SOLID.push\_back(I)$
6: $k \leftarrow 1$; $stop \leftarrow 0$; $stop_{max} \leftarrow \lceil \frac{n}{M} \rceil$
7: **while not** $DASHED.empty()$ **do**
8: $\quad k \leftarrow k+1$; $stop \leftarrow stop+1$
9: $\quad$ **if** $stop > stop_{max}$ **then**
10: $\quad\quad stop \leftarrow 1$
11: $\quad first \leftarrow (stop-1) \cdot M$; $last \leftarrow stop \cdot M - 1$
12: $\quad$ CountSupp($DASHED$)
13: $\quad$ Prune($DASHED$)
14: $\quad$ MakeCand($DASHED$)
15: $\quad$ CheckFullPass($DASHED$)
16: $\mathcal{L} \leftarrow \{I \mid I \in SOLID \wedge I.shape =$ BOX$\}$
17: **return** $\mathcal{L}$

$$
\begin{gathered}
SOLID := Circle_{Solid} \cup Box_{Solid}, \\
Circle_{Solid} := \{I \mid I \subseteq \mathcal{I} \wedge Count(I) = n \wedge supp(I) < minsup\}, \\
Box_{Solid} := \{I \mid I \subseteq \mathcal{I} \wedge Count(I) = n \wedge supp(I) \geqslant minsup\}.
\end{gathered} \tag{9}
$$

После обработки блока наборы перераспределяются по указанным множествам и генерируются новые наборы для подсчета их поддержки.

Распараллеливанию с помощью технологии OpenMP подвергаются следующие стадии алгоритма: подсчет поддержки, отбрасывание заведомо редких наборов и проверка завершения просмотра наборов по всей базе транзакций. Подсчет поддержки (см. алг. 3) 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. При подсчете поддержки осуществляется балансировка нагрузки нитей в зависимости от текущей мощности множества наборов-кандидатов. В случае, если наборов больше, чем доступных алгоритму нитей, внешний цикл распараллеливается на все нити. Иначе алгоритм активирует режим вложенного параллелизма, и внешний цикл распараллеливается на количество нитей, равное количеству наборов-кандидатов. Предложенная техника обеспечивает баланс загрузки нитей в начальной и конечной стадиях работы алгоритма (когда кандидатов становится соответственно больше либо меньше, чем нитей).

Алг. 3 CountSupp(in out $DASHED$)

```
1: if DASHED.size() ⩾ num_of_threads then
2:     #pragma omp parallel for
3:     for I ∈ DASHED do
4:         I.stop ← I.stop + 1
5:         for 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 I ∈ DASHED do
12:        I.stop ← I.stop + 1
13:        #pragma omp parallel for reduction(+:I.supp)
14:         num_threads(⌈num_of_threads / DASHED.size()⌉)
15:        for T ∈ B_first .. B_last do
16:            if I.mask AND T = I.mask then
17:                I.supp ← I.supp + 1
18: return DASHED
```

Алг. 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})
13: return DASHED
```

Отбрасывание заведомо редких наборов позволяет существенно сократить вычисления и выполняется следующим образом (см. алг. 4). Максимально возможная поддержка набора вычисляется путем сложения текущего значения поддержки данного набора с количеством транзакций, которые еще не были обработаны. Если вычисленная максималь-

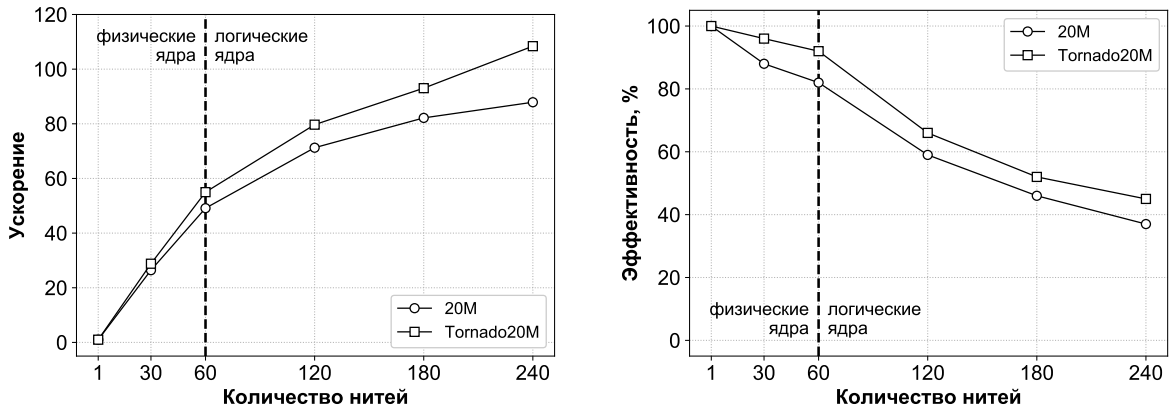
но возможная поддержка меньше, чем порог $minsup$, то данный набор заведомо является редким, и может быть исключен из дальнейшего рассмотрения. Дополнительно отбрасывается каждый набор-кандидат, который является надмножеством заведомо редкого набора.

По завершении отбрасывания заведомо редких наборов выполняется генерация новых наборов-кандидатов для обработки на следующей итерации алгоритма. Генерация выполняется применением логической побитовой операции `OR` к каждой паре имеющихся кандидатов.

Алг. 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})
9: return DASHED
```

Финальным шагом обработки наборов-кандидатов является проверка завершения просмотра этих наборов по всей базе данных транзакций (см. алг. 5), выполняемая параллельно. Если просмотр завершен и набор имеет поддержку не ниже порога $minsup$, то набор является частым и перемещается в искомое множество Box_{Solid}.



(a) ускорение

(b) параллельная эффективность

Рис. 5. Масштабируемость алгоритма *PDIC*

Эксперименты с алгоритмом *PDIC* на суперкомпьютере «Торнадо ЮУрГУ» показали (см. рис. 5; реальные и синтетические данные, $|\mathcal{D}| = 2{\cdot}10^7$), что предложенный алгоритм хорошо масштабируется и опережает алгоритмы-аналоги.

Алг. 6 DDCapriori

Master

(in $\mathcal{D}, minsup$; out $\mathcal{L}$)

▷ Отправка заданий рабочим
1: **for all** S_j **do**
2: $task_j \leftarrow \mathsf{MakeTask}(D_j)$
3: $\mathsf{Send}(S_j, task_j)$
4: $\mathsf{Send}(S_j, \mathcal{D})$
5: $\mathsf{Send}(S_j, I)$
▷ Вычисление L_1
6: **for all** S_j **do**
7: $\mathsf{Recv}(S_j, supp_j)$
8: **for all** $i \in \mathcal{I}$ **do**
9: $supp(i) \leftarrow \sum_{j=1}^{n} supp_j(i)$
10: $\mathcal{L}_1 \leftarrow \mathcal{I} \setminus \{i \mid supp(i) < minsup\}$
▷ Обработка k-наборов
11: $k \leftarrow 2$
12: **while not** $Stop$ **do**
▷ Генерация кандидатов
13: $C_k \leftarrow \textsc{MakeCand}(\mathcal{L}_{k-1})$
▷ Вычисление результата
14: **if** $C_k = \varnothing$ **then**
15: $\mathcal{L} \leftarrow \cup_{j=1}^{k-1} \mathcal{L}_j$
16: $Stop \leftarrow \texttt{TRUE}$
17: **break**
18: **for all** S_j **do**
19: $msg_j \leftarrow \textsc{MakeTask}(C_k^j)$
20: $\mathsf{Send}(S_j, msg_j)$
▷ Агрегация результатов
21: $\mathsf{Recv}(S_j, \mathcal{L}_k^j)$
22: $\mathcal{L}_k \leftarrow \cup_{j=1}^{num_{slaves}} \mathcal{L}_k^j$
23: $k \leftarrow k + 1$
24: **return** $\mathcal{L}$

Slave

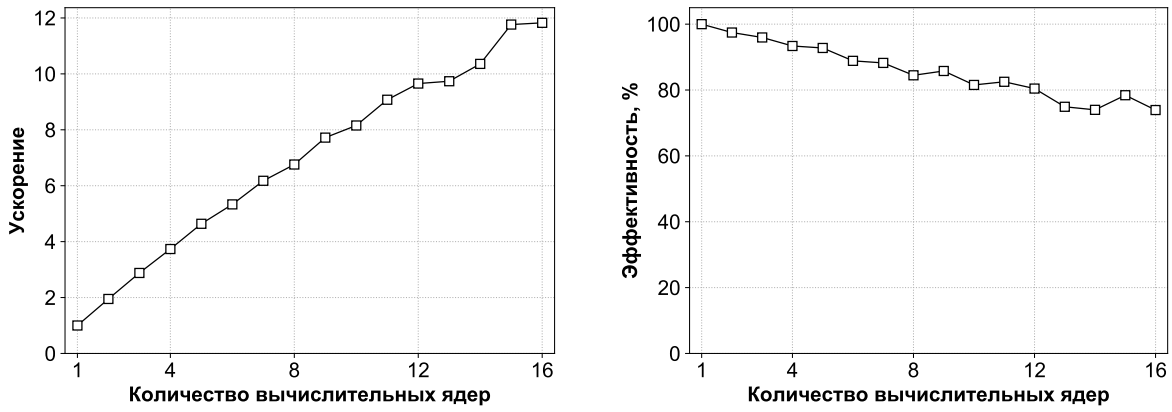
(in $task$, $minsup$; out $\mathcal{L}_k$)

▷ Прием задания от мастера
1: $k \leftarrow 1$
2: $\mathsf{Recv}(M, task)$
3: $\mathsf{Recv}(M, \mathcal{D})$
4: $\mathsf{Recv}(M, C)$
▷ Обработка 1-наборов
5: **for all** $c \in C$ **do**
6: $supp(c) \leftarrow \textsc{CountSupp}(c, D)$
7: $\mathsf{Send}(M, supp)$
▷ Обработка k-наборов
8: **while not** $Stop$ **do**
9: $k \leftarrow k + 1$
10: **if** $C = \varnothing$ **then**
11: $Stop \leftarrow \texttt{TRUE}$
12: **break**
▷ Вычисление поддержки
13: **for all** $c \in C$ **do**
14: $supp(c) \leftarrow \textsc{CountSupp}(c, \mathcal{D})$
▷ Отбрасывание
15: $\mathcal{L}_k \leftarrow C \setminus \{c \mid supp(c) < minsup\}$
▷ Отправка мастеру
16: $\mathsf{Send}(M, \mathcal{L}_k)$

Параллельный *алгоритм DDCapriori* (см. алг. 6) использует вычислительную модель «мастер-рабочие». *Нить-мастер* запускается на управляющем ядре PPE ускорителя IBM Cell и формирует множество частых наборов $\mathcal{L}_k$ и множества наборов-кандидатов C_k, распределяя его между рабочими. *Нити-рабочие* запускаются на вычислительных ядрах SPE и выполняют вычисление поддержки для наборов-кандидатов из C_k, получаемых от мастера.

Мастер отправляет каждому рабочему первое задание и множество транзакций $\mathcal{D}$, после чего переходит в состояние ожидания. По получении результатов от рабочих мастер выполняет их агрегацию и отбрасывание редких наборов, формируя таким образом множество $\mathcal{L}_1$. Далее из частых k-наборов множества $\mathcal{L}_k$ мастер формирует множество наборов-кандидатов C_{k+1}. Затем мастер отправляет рабочим задания на обработку полученного множества кандидатов и ожидает от них результаты вычислений (значения поддержки). Если не удается сформировать множество C_{k+1}, то мастер прерывает обработку, уничтожает рабочих и вычисляет результирующее множество частых наборов $\mathcal{L}$. Рабочий, получив задание от мастера, формирует множество наборов-кандидатов C_1 из своего подмножества транзакций и отправляет результаты мастеру. Далее рабочий циклически выполняет следующую последовательность действий: ожидание от мастера подмножества наборов, вычисление поддержки и отправка результата вычислений мастеру. Цикл завершается, если получено задание на обработку пустого множества наборов.

Подсчет поддержки наборов организован с использованием векторных функций Cell, обеспечивающих эффективную реализацию проверки вхождения набора в транзакцию.



(a) ускорение

(b) параллельная эффективность

Рис. 6. Масштабируемость алгоритма *DDCapriori*

Эксперименты с алгоритмом *DDCapriori* показали (см. рис. 6; реальные данные, $|\mathcal{D}| = 10^6$) ускорение, близкое к линейному.

Третья глава, «Анализ временных рядов», посвящена методам интеллектуального анализа временных рядов на платформе современных многопроцессорных многоядерных вычислительных систем. Предложены следующие новые параллельные алгоритмы: алгоритм *PBM* поиска похожих подпоследовательностей во временном ряде для кластерных систем с узлами на базе многоядерных ускорителей и алгоритм поиска диссонансов во временном ряде *MDD* для многоядерных ускорителей. Представлены результаты вычислительных экспериментов, исследующих эффективность предложенных алгоритмов.

Алгоритм PBM предполагает двухуровневое распараллеливание вычислений: на уровне всех узлов кластерной системы и внутри одного узла кластера. Параллелизм на уровне узлов достигается посредством разбиения временного ряда на фрагменты, каждый из которых размещается на диске отдельного вычислительного узла кластерной системы. Для обменов данными между узлами кластера используется технология MPI, для распараллеливания вычислений в рамках одного узла кластера используется технология OpenMP.

Предложенная техника фрагментации предотвращает потери результирующих подпоследовательностей на стыке фрагментов при поиске. Пусть P ($P{>}1$) — количество фрагментов, на которые следует разбить временной ряд T, $T^{(k)}$ — k-й фрагмент ряда ($0 \leqslant k \leqslant P-1$). Тогда *фрагмент* $T^{(k)}$ определяется как подпоследовательность $T_{start,\,len}$, где

$$start := k \cdot \lfloor \tfrac{N}{P} \rfloor + 1,$$
$$len := \begin{cases} \lfloor \frac{N}{P} \rfloor + (N \bmod P) + n - 1, & k = P-1 \\ \lfloor \frac{N}{P} \rfloor + n - 1, & otherwise. \end{cases} \qquad (10)$$

Предложена оригинальная компоновка данных в оперативной памяти вычислительного узла кластера, обеспечивающая представление фрагмента временного ряда и вспомогательных данных алгоритма в виде выровненных в памяти матриц, циклы обработки которых векторизуются компилятором. Пусть обработка подпоследовательности $T_{i,\,n}$ осуществляется с использованием векторного регистра ускорителя, вмещающего w вещественных чисел. Если длина подпоследовательности не кратна w, то $T_{i,\,n}$ дополняется фиктивными нулевыми элементами. Обозначим количество фиктивных элементов как $pad := w - (n \bmod w)$, тогда выровненная подпоследовательность $\tilde{T}_{i,\,n}$ определяется следующим образом:

$$\tilde{T}_{i,n} := \begin{cases} (t_i, t_{i+1}, \ldots, t_{i+n-1}, \overbrace{0, 0, \ldots, 0}^{pad}), & if\ n\ mod\ w > 0 \\ (t_i, t_{i+1}, \ldots, t_{i+n-1}), & otherwise. \end{cases} \quad (11)$$

Для обеспечения векторизации вычислений все выровненные подпоследовательности временного ряда хранятся в виде *матрицы подпоследовательностей* $S_T^n \in \mathbb{R}^{N \times (n+pad)}$, определяемой следующим образом:

$$S_T^n(i, j) := \tilde{t}_{i+j-1}. \quad (12)$$

Каждая подпоследовательность C исходного временного ряда *нормализуется* следующим образом: $\hat{C} := (\hat{c}_1, \ldots, \hat{c}_n)$, где

$$\hat{c}_i := \frac{c_i - \mu}{\sigma},$$
$$\mu := \frac{1}{n}\sum_{i=1}^{n} c_i, \quad \sigma := \sqrt{\frac{1}{n}\sum_{i=1}^{n} c_i^2 - \mu^2}. \quad (13)$$

Нижняя граница схожести представляет собой функцию $LB : S_T^n \times S_T^n \to \mathbb{R}_+ \cup \{0\}$, вычислительная сложность которой меньше $O(nr)$ (вычислительной сложности меры DTW). Нижняя граница используется для отбрасывания подпоследовательностей, заведомо непохожих на поисковый запрос, без вычисления меры DTW, что позволяет существенно сократить объем вычислений. Алгоритм выполняет сканирование подпоследовательностей исходного временного ряда, с $T_{1,n}$ до $T_{N,n}$, с шагом 1. Лучшее среди всех просмотренных на текущий момент подпоследовательностей значение схожести с поисковым запросом Q обозначается как *bsf (best-so-far)*. Если нижняя граница для текущей подпоследовательности $T_{i,n}$ превышает bsf, то значение меры DTW для данной подпоследовательности также превысит bsf, и $T_{i,n}$ заведомо непохожа на запрос.

Алгоритм инициализирует bsf значением $+\infty$ и на i-м шаге сканирования ряда пытается улучшить (уменьшить) значение bsf, вычисляя его следующим образом:

$$bsf_{(i)} = \min\Big(bsf_{(i-1)}, \begin{cases} +\infty & , LB(Q, T_{i,n}) > bsf_{(i-1)} \\ DTW(Q, T_{i,n}) & , otherwise \end{cases}\Big) \quad (14)$$

Для повышения эффективности отбрасывания заведомо непохожих подпоследовательностей применяется каскад нижних границ $LB_{Kim}FL$, $LB_{Keogh}EQ$, $LB_{Keogh}EC$, определяемых следующим образом:

$$LB_{Kim}FL(Q,C) := \mathrm{ED}(q_1,c_1) + \mathrm{ED}(q_n,c_n); \tag{15}$$

$$LB_{Keogh}EC(Q,C) := \sum_{i=1}^{n} \begin{cases} (c_i - u_i)^2, & if\ c_i > u_i \\ (c_i - \ell_i)^2, & if\ c_i < \ell_i \\ 0, & otherwise \end{cases}, \tag{16}$$

где последовательности $U = (u_1, \ldots, u_n)$ и $L = (\ell_1, \ldots, \ell_n)$ вычисляются следующим образом:

$$\begin{aligned} u_i &:= \max(q_{i-r}, \ldots, q_{i+r}), \\ \ell_i &:= \min(q_{i-r}, \ldots, q_{i+r}); \end{aligned} \tag{17}$$

$$LB_{Keogh}EQ(Q,C) := LB_{Keogh}EC(C,Q). \tag{18}$$

Алгоритм *PBM* выполняет предварительное вычисление всех нижних границ схожести и их хранение в виде матрицы, определяемой следующим образом. Пусть в алгоритме поиска используется lb_{max} ($lb_{max} \geqslant 1$) нижних границ схожести, и за LB_1, …, $LB_{lb_{max}}$ обозначим эти границы, перечисленные в порядке их вычисления в каскаде применения нижних оценок. Тогда $L_T^n \in \mathbb{R}^{N \times lb_{max}}$, *матрица нижних границ схожести* всех подпоследовательностей длины n временного ряда T с поисковым запросом Q, имеет следующий вид:

$$L_T^n(i,j) := LB_j(T_{i,n}, Q). \tag{19}$$

Карта схожести представляет собой вектор-столбец $B_T^n \in \mathbb{B}^N$, который для каждой подпоследовательности длины n ряда T хранит конъюнкцию применения всех нижних границ схожести этой подпоследовательности с текущим значением порога *bsf*:

$$B_T^n(i) := \wedge_{j=1}^{lb_{max}}(L_T^n(i,j) < bsf). \tag{20}$$

Строки матрицы S_T^n, не отброшенные при применении нижних границы схожести, помещаются в матрицу кандидатов для последующей параллельной обработки. Вычисляются значения меры DTW между кандидатами и поисковым запросом и их минимум используется в качестве порога *bsf*. Пусть параллельный алгоритм запускается на p ($p \geqslant 1$) нитях

и каждая нить обрабатывает сегмент из s ($s \leqslant \lceil \frac{N}{p} \rceil$) кандидатов. Тогда *матрица кандидатов* $C_T^n \in \mathbb{R}^{(s \cdot p) \times (n+pad)}$ имеет следующий вид:

$$C_T^n(i, \cdot) := S_T^n(k, \cdot) : B_T^n(i) = \texttt{TRUE}. \tag{21}$$

Для хранения номера последнего обработанного кандидата в сегменте алгоритм использует индекс $Pos \in \mathbb{N}^p$:

$$\begin{gathered} pos_i = k : p \cdot (i-1) + 1 \leqslant k \leqslant \lceil \tfrac{N}{i \cdot p} \rceil \wedge \\ \forall j \leqslant lb_{max}, LB_T^n(k, j) < bsf. \end{gathered} \tag{22}$$

Для хранения позиции подпоследовательности во временном ряде алгоритм использует индекс $Idx \in \mathbb{N}^{s \cdot p}$:

$$idx_i = k : 1 \leqslant k \leqslant N \wedge \exists S_T^n(i, \cdot) \Leftrightarrow \exists T_{i,n} \Leftrightarrow k = (i-1) \cdot n + 1. \tag{23}$$

Алг. 7 PBM(IN T, Q, r; OUT $bsf, bestmatch$)

▷ Инициализация

1: $myrank \leftarrow$ MPI_Comm_rank(); $N \leftarrow |T^{(myrank)}| - n + 1$
2: $num_{cand} \leftarrow N$; $subseq_{rnd} \leftarrow T^{(myrank)}_{random(1..N),\, n}$
3: $bsf \leftarrow \mathrm{DTW}(subseq_{rnd}, Q, r, \infty)$

▷ Стадия подготовки данных

4: CalcEnvelope(Q, r, U, L)
5: $L^n_{T^{(myrank)}} \leftarrow$ CalcLowerBounds$(S^n_{T^{(myrank)}}, Q, r)$
6: $myFragDone \leftarrow$ FALSE

▷ Стадия улучшения порога bsf

7: **repeat**
8: $\quad B^n_{T^{(myrank)}} \leftarrow$ LowerBounding$(L^n_{T^{(myrank)}}, bsf)$
9: $\quad \{C^n_{T^{(myrank)}}, num_{cand}\} \leftarrow$ FillCandMatr$(S^n_{T^{(myrank)}}, B^n_{T^{(myrank)}})$
10: $\quad$ **if** $num_{cand} > 0$ **then**
11: $\quad\quad \{bsf, bestmatch\} \leftarrow$ CalcCandMatr$(C^n_{T^{(myrank)}}, num_{cand}, r)$
12: $\quad$ **else**
13: $\quad\quad myFragDone \leftarrow$ TRUE
14: $\quad \{bsf, bestmatch\} \leftarrow$ MPI_Allreduce($\{bsf, bestmatch\}$, MPI_FLOAT_LONG, MPI_MIN)
15: $\quad Stop \leftarrow$ MPI_Allreduce($myFragDone$, MPI_BOOL, MPI_AND)
16: **until** $Stop$
17: **return** $\{bsf, bestmatch\}$

Алгоритм *PBM* (см. алг. 7) выполняется следующим образом. При *инициализации* с помощью MPI-функции алгоритм определяет номер текущего узла кластера $myrank$, и далее каждый узел обрабатывает матрицу подпоследовательностей $S^n_{T(myrank)}$. Порог bsf инициализируется значением меры схожести DTW между поисковым запросом и случайной подпоследовательностью текущего фрагмента.

Алг. 8 CalcLowerBounds(in S^n_T, Q, r, p; out L^n_T)

1: #pragma omp parallel for num_threads(p)
2: **for** i **from** 1 **to** N **do**
3: ZNormalize($S^n_T(i,\cdot)$)
4: $L^n_T(i,1) \leftarrow LB_{Kim}FL(Q, S^n_T(i,\cdot))$
5: $L^n_T(i,2) \leftarrow LB_{Keogh}EC(Q, S^n_T(i,\cdot))$
6: CalcEnvelope($S^n_T(i,\cdot), r, U, L$)
7: $L^n_T(i,3) \leftarrow LB_{Keogh}EQ(S^n_T(i,\cdot), Q, U, L)$
8: **return** L^n_T

На стадии *подготовки данных* алгоритм выполняет нормализацию строк матрицы выровненных подпоследовательностей и вычисляет матрицу нижних границ схожести (см. алг. 8). Указанные вычисления реализованы с помощью векторизуемых компилятором циклов и распараллеливаются с помощью директив OpenMP.

Алг. 9 LowerBounding(in L^n_T, bsf, p; out B^n_T)

1: #pragma omp parallel for num_threads(p)
2: $whoami \leftarrow$ omp_get_thread_num()
3: **for** $i \in pos_{whoami}..\lceil \frac{N}{whoami \cdot p} \rceil$ **do**
4: $B^n_T(i) \leftarrow$ TRUE
5: **for** $j \in 1..lb_{max}$ **do**
6: $B^n_T(i) \leftarrow B^n_T(i)$ AND $(L^n_T(i,j) < bsf)$
7: **return** B^n_T

Далее на стадии *улучшения порога схожести* алгоритм выполняет следующий цикл действий, пока каждый узел кластера не завершит обработку своего фрагмента. На основе матрицы нижних границ схожести вычисляется карта схожести (см. алг. 9). Кандидат со значением FALSE в карте схожести заведомо непохож на образец поиска и отбрасывается без вычисления меры DTW, иначе он добавляется в матрицу кандидатов.

Заполнение матрицы кандидатов выполняется следующим образом (см. алг. 10). Карта схожести разбивается на сегменты равного размера

Алг. 10 FillCandMatr(in S_T^n, B_T^n, p, s; out C_T^n, num_{cand})

```
1: num_cand ← 0
2: for i ∈ 1..p do
3:     for k ∈ 1..s do
4:         if B_T^n(pos_i + k) =TRUE then
5:             if num_cand < s · p then
6:                 num_cand ← num_cand + 1
7:                 C_T^n(num_cand, ·) ← S_T^n(pos_i, ·)
8:                 idx_{num_cand} ← (pos_i − 1) · n + 1
9:             else
10:                break
11:    if num_cand = s · p then
12:        break
13: return {C_T^n, num_cand}
```

по числу используемых нитей ускорителя и выполняется сканирование каждого сегмента. Если элемент карты схожести имеет значение TRUE, то соответствующая подпоследовательность не является заведомо непохожей на образец поиска и добавляется в матрицу кандидатов. Сканирование сегмента начинается с соответствующего элемента индексного массива Pos.

Алг. 11 CalcCandMatr(in $C_T^n, num_{cand}, Q, r, p$; out bsf, $bestmatch$)

```
1: #pragma omp parallel for num_threads(p) shared (bsf, Idx) private
   (distance)
2: for i ∈ 1..num_cand do
3:     distance ← DTW(C_T^n(i, ·), Q, r, bsf)
4:     #pragma omp critical
5:     if bsf > distance then
6:         bsf ← distance
7:         bestmatch ← idx_i
8: return {bsf, bestmatch}
```

По заполнении матрицы кандидатов выполняется цикл вычисления меры схожести DTW каждой ее строки с поисковым запросом и поиск минимума этих значений, распараллеливаемые с помощью директив OpenMP (см. алг. 11).

Реализация подсчета меры схожести DTW (см. алг. 12) предполагает разбиение одного из циклов на два, чтобы избежать зависимости по

Алг. 12 DTW(IN $X = (x_1, \dots, x_n)$, $Y = (x_1, \dots, x_n)$, r, bsf)

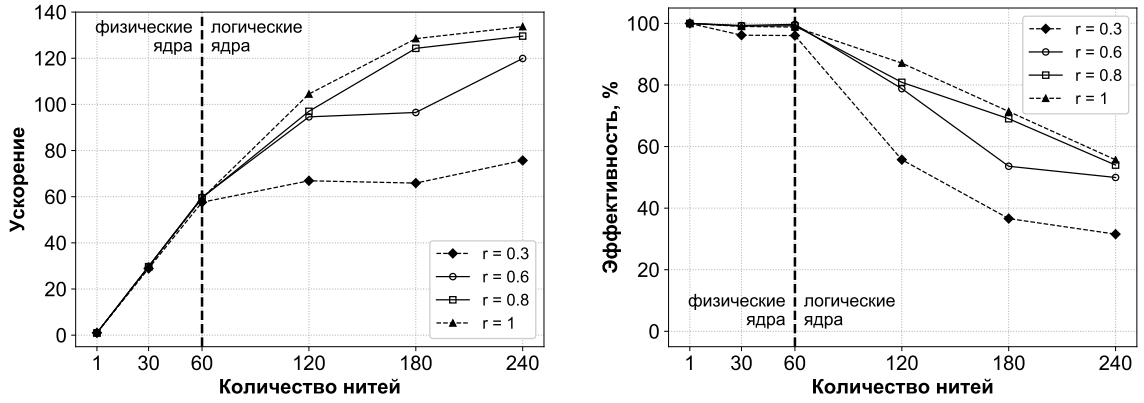
$cost(1, \dots, n) \leftarrow \overline{bsf}$; $cost_{prev}(1, \dots, n) \leftarrow \overline{bsf}$; $cost_{prev}(1) \leftarrow d(x_1, y_1)$
for $j \in \max(2, i - r)..\min(n, i + r)$ **do**
    $cost_{prev}(j) \leftarrow cost_{prev}(j) + d(x_1, y_j)$
for $i \in 2..n$ **do**
    ▷ Цикл без зависимостей по данным (*векторизуемый*)
    for $j \in \max(1, i - r)..\min(n, i + r)$ **do**
        $cost(j) \leftarrow \min(cost_{prev}(j), cost_{prev}(j - 1)$
    ▷ Цикл с зависимостями по данным (не векторизуемый)
    for $j \in \max(1, i - r)..\min(n, i + r)$ **do**
        $cost(j) \leftarrow d(x_i, y_j) + \min(cost_{prev}(j), cost_{prev}(j - 1))$
    SWAP($cost$,$cost_{prev}$)
return $cost_{prev}(n)$

данным и векторизовать часть вычислений, что повышает общую производительность алгоритма.

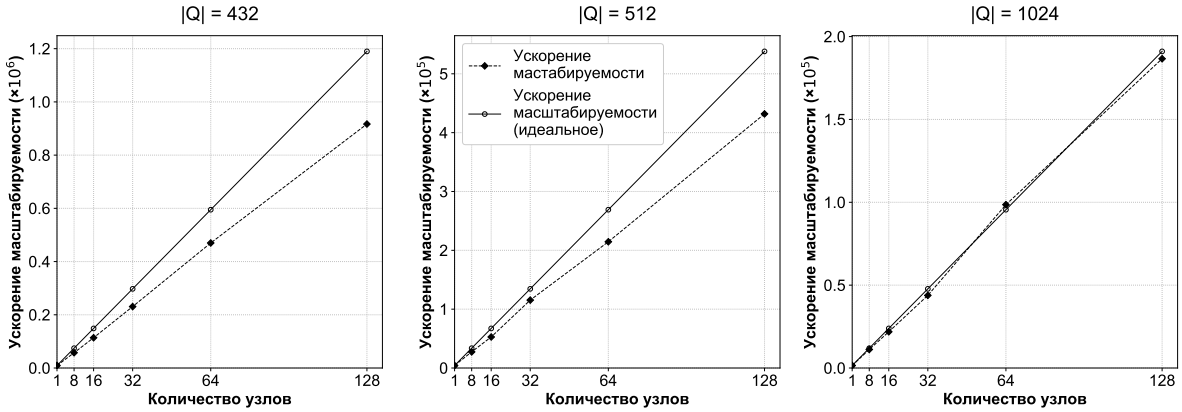
Далее алгоритм выполняет выбор наименьшего среди всех фрагментов ряда значения порога схожести и соответствующей подпоследовательности с помощью операции глобальной редукции стандарта MPI. Момент полного завершения обработки ряда определяется аналогичным образом путем вычисления глобальной конъюнкции флагов завершения обработки каждого фрагмента.

Для исследования эффективности алгоритма *PBM* проведены вычислительные эксперименты на суперкомпьютере «Торнадо ЮУрГУ». На одном узле кластера алгоритм демонстрирует (см. рис. 7a; реальный ряд, $|T| = 2.5 \cdot 10^5, |Q| = 360$) ускорение, близкое к линейному, и параллельную эффективность, близкую к 100%, если количество нитей, на которых запущен алгоритм, совпадает с количеством физических ядер ускорителя. Бóльшая точность при определении схожести (параметр r) обеспечивает больший объем вычислений и большее ускорение. На кластерной системе в целом алгоритм демонстрирует (см. рис. 7b; реальный ряд, $|T| = 12.8 \cdot 10^7$) ускорение масштабируемости, близкое к линейному. При сравнении с лучшими аналогами алгоритм *PBM* показал быстродействие, опережающее конкурентов.

Алгоритм MDD (см. алг. 13) при нахождении диссонансов использует возможность независимого вычисления евклидовых расстояний между подпоследовательностями ряда. Алгоритм состоит из двух стадий: подготовка и поиск. На стадии подготовки выполняется построение вспо-



(a) на одном узле кластера



(b) на кластерной системе в целом

Рис. 7. Масштабируемость алгоритма *PBM*

Алг. 13 MDD(IN T, n, w; OUT $\{pos_{bsf}, dist_{bsf}\}$)

▷ Стадия подготовки данных

1: $S_T^n \leftarrow$ ZNORMALIZE(S_T^n)
2: $W_{\mathcal{A}} \leftarrow$ MAKEWORDMATRIX($\mathcal{A}, w$)
3: $PAA_T^{n,w} \leftarrow$ PAA(S_T^n, w)
4: $SAX_T^{n,\mathcal{A}} \leftarrow$ SAX($PAA_T^{n,w}, \mathcal{A}$)
5: $Cand \leftarrow$ MAKECANDIDATES($SAX_T^{n,\mathcal{A}}$)
6: $I_W \leftarrow$ MAKEINDEXWORD($SAX_T^{n,\mathcal{A}}$)

▷ Стадия поиска диссонанса

7: $pos_{bsf} \leftarrow 0$; $dist_{bsf} \leftarrow 0$
8: $\{pos_{bsf}, dist_{bsf}\} \leftarrow$ FIND($I_W, Cand, pos_{bsf}, dist_{bsf}$)
9: $\{pos_{bsf}, dist_{bsf}\} \leftarrow$ REFINE($I_W, Cand, pos_{bsf}, dist_{bsf}$)
10: **return** $\{pos_{bsf}, dist_{bsf}\}$

могательных матричных структур данных, обеспечивающих эффектив-

ную векторизацию вычислений на ускорителе. На стадии поиска алгоритм находит диссонанс с помощью построенных структур.

На стадии подготовки в соответствии с (11) и (12) формируется $S_T^n \in \mathbb{R}^{N\times(n+pad)}$, матрица выровненных нормализованных подпоследовательностей. Подпоследовательности ряда кодируются с помощью метода символьной агрегатной аппроксимации (SAX, Symbolic Aggregate ApproXimation) и формируется *матрица SAX-кодов* $SAX_T^{n,\mathcal{A}} \in \mathbb{N}^{N\times w}$, где $\mathcal{A}$ и w — алфавит и степень агрегации соответственно (параметры алгоритма). Далее создается *словарный индекс* — матрица $I_W \in \mathbb{N}^{dict_{size}\times N}$, в которой хранятся номера закодированных подпоследовательностей в матрице SAX-кодов:

$$I_W(i,j) = k \Leftrightarrow W_{\mathcal{A}}(i,\cdot) = SAX_T^{n,\mathcal{A}}(k,\cdot). \tag{24}$$

Матрица $W_{\mathcal{A}} \in \mathbb{N}^{dict_{size}\times w}$ — это *словарь*, который хранит все возможные слова длины w в алфавите $\mathcal{A}$. Мощность словаря $dict_{size}$ вычисляется как число размещений символов алфавита $\mathcal{A}$ по w символов с повторениями:

$$dict_{size} := \bar{A}^w_{|\mathcal{A}|} = |\mathcal{A}|^w. \tag{25}$$

С помощью хэш-функции $h : \mathbb{N}^w \to \{1, 2, \ldots, dict_{size}\}$ словарь организуется таким образом, чтобы слова в нем были упорядочены по возрастанию:

$$h(a_1, a_2, \ldots, a_w) := \sum_{j=1}^{w+1} a_j \cdot w^{w-j-1}. \tag{26}$$

Затем формируется *индекс потенциальных диссонансов* $Cand \in \mathbb{N}^N$ — упорядоченный по возрастанию массив с номерами тех подпоследовательностей в матрице S_T^n, которые наиболее редко встречаются в матрице SAX-кодов:

$$\begin{gathered} Cand(i) = k \Leftrightarrow F_{SAX}(k) = \min_{1\leqslant j\leqslant N} F_{SAX}(j) \wedge \\ \forall i < j \; Cand(i) < Cand(j). \end{gathered} \tag{27}$$

Частотный индекс SAX-кодов $F_{SAX} \in \mathbb{N}^N$ хранит частоты слов из матрицы SAX-кодов:

$$F_{SAX}(i) = k \Leftrightarrow |\{j \mid SAX_T^{n,\mathcal{A}}(j,\cdot) = SAX_T^{n,\mathcal{A}}(i,\cdot)\}| = k. \tag{28}$$

Частотный индекс словаря $F_W \in \mathbb{N}^{|\mathcal{A}|^w}$ для каждого слова в словаре хранит частоту появления этого слова в матрице SAX-кодов:

$$F_W(i) = k \Leftrightarrow k = |\{j \mid W_{\mathcal{A}}(i, \cdot) = SAX_T^{n,\mathcal{A}}(j, \cdot)\}|. \tag{29}$$

Алг. 14 Реализация стадии поиска алгоритма *MDD*

FIND (IN $I_W, Cand$; OUT $\{pos_{bsf}, dist_{bsf}\}$)

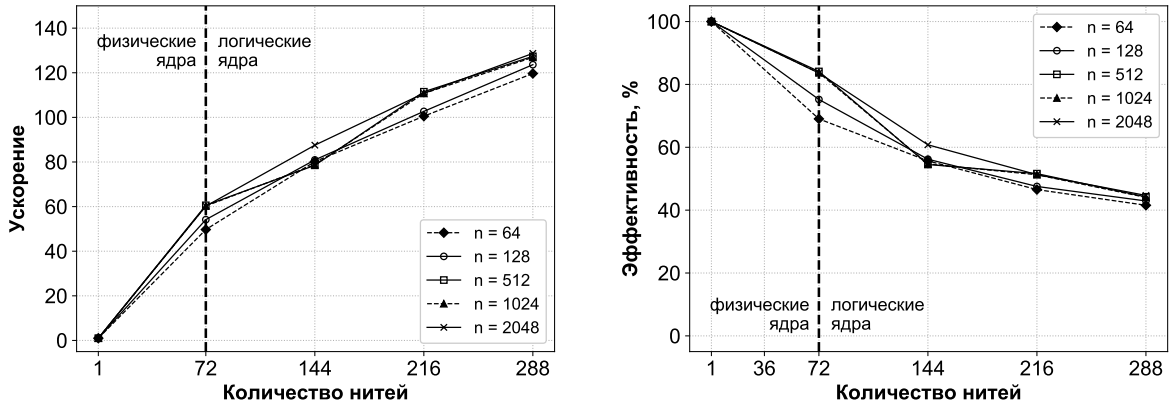
```
for C_i ∈ Cand do
    dist_min ← +∞
    #pragma omp parallel for
    for C_j ∈ I_W(SAX_T^{n,A}(C_i)) do
        if |i − j| ⩾ n then
            continue
        d ← ED²(C_i, C_j)
        if d < dist_bsf then
            break
        dist_min ← min(d, dist_min)
    #pragma omp parallel for
    for C_j ∉ I_W(SAX_T^{n,A}(C_i)) do
        if |i − j| ⩾ n then
            continue
        d ← ED²(C_i, C_j)
        if d < dist_bsf then
            break
    dist_min ← min(d, dist_min)
    if dist_min > dist_bsf then
        dist_bsf ← dist_min
        pos_bsf ← i
return {pos_bsf, dist_bsf}
```

REFINE (IN $I_W, Cand$; OUT $\{pos_{bsf}, dist_{bsf}\}$)

```
#pragma omp parallel for
for C_i ∉ Cand do
    dist_min ← +∞
    for C_j ∈ I_W(SAX_T^{n,A}(C_i)) do
        if |i − j| ⩾ n then
            continue
        d ← ED²(C_i, C_j)
        if d < dist_bsf then
            break
        dist_min ← min(d, dist_min)
    for C_j ∉ I_W(SAX_T^{n,A}(C_i)) do
        if |i − j| ⩾ n then
            continue
        d ← ED²(C_i, C_j)
        if d < dist_bsf then
            break
        dist_min ← min(d, dist_min)
    if dist_min > dist_bsf then
        dist_bsf ← dist_min
        pos_bsf ← i
return {pos_bsf, √dist_bsf}
```

Стадия поиска состоит из двух следующих шагов (см. алг. 14). На первом шаге выполняется поиск диссонансов среди подпоследовательностей, входящих в индекс потенциальных диссонансов, построенный на предыдущей стадии, в порядке, задаваемом указанным индексом. Результатом данного шага позиция предполагаемого диссонанса в исходном временном ряде и расстояние до его ближайшего соседа. На втором шаге выполняется уточнение найденной ранее позиции предполагаемого диссонанса среди тех подпоследовательностей, которые не входят в

индекс потенциальных диссонансов. На каждой итерации цикла перебора подпоследовательностей находится расстояние до ближайшего соседа и обновляется значение пары (pos_{bsf}, $dist_{bsf}$), определяющей диссонанс: максимальное расстояние до ближайшего соседа и индекс соответствующей подпоследовательности исходного ряда. Если вычисленное расстояние меньше, чем $dist_{bsf}$, то соседи текущей подпоследовательности отбрасываются без вычисления расстояний. В качестве метрики в алгоритме используется квадрат евклидова расстояния, чтобы операция извлечения квадратного корня не замедляла вычисления. Распараллеливание циклов перебора осуществляется с помощью директив OpenMP.



(a) ускорение

(b) параллельная эффективность

Рис. 8. Масштабируемость алгоритма *MDD*

Эксперименты с алгоритмом *MDD* на суперкомпьютере ИВМиМГ СО РАН показали (см. рис. 8; реальный ряд, $|T| = 10^7$) ускорение, близкое к линейному, и параллельную эффективность 70–85%, если количество нитей, на которых запущен алгоритм, совпадает с количеством физических ядер ускорителя архитектуры Intel MIC. При сравнении с лучшими аналогами алгоритм *MDD* показал быстродействие, опережающее конкурентов в 2–3 раза.

В четвертой главе, «Интеграция в СУБД параллельных алгоритмов анализа данных», представлен подход к интеграции интеллектуального анализа данных в реляционную СУБД. Предложенный подход предполагает внедрение параллельных алгоритмов интеллектуального анализа данных в реляционную СУБД с открытым кодом на основе определяемых пользователем функций. В качестве целевой площадки реализации подхода рассмотрена СУБД PostgreSQL. Описана системная архитектура и методы реализации подхода. Предложены параллельные алгоритмы вычисления матрицы евклидовых расстояний *PBlockwise*

и кластеризации данных *PPAM* для ускорителей Intel MIC, внедряемые в СУБД. Представлены результаты вычислительных экспериментов, исследующих эффективность предложенного подхода.

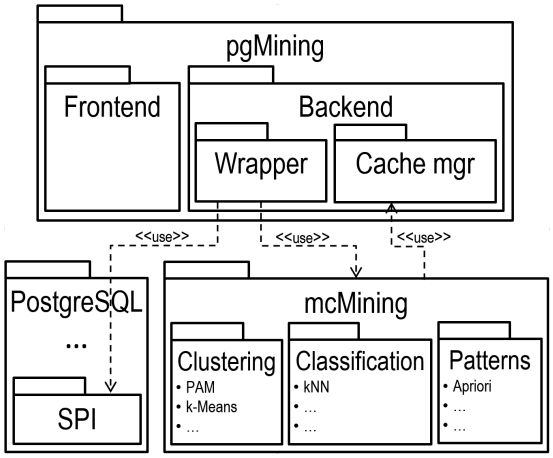


Рис. 9. Архитектура интеграции интеллектуального анализа данных в СУБД

Системная архитектура *подхода к интеграции интеллектуального анализа данных в СУБД* (см. рис. 9) предполагает разработку двух следующих компонентов, работающих в связке с СУБД. Компонент *pgMining* обеспечивает библиотечные функции, выполняющие интеллектуальный анализ данных в рамках СУБД. Библиотека функций, реализующих параллельные алгоритмы анализа данных для многоядерных ускорителей, представлена компонентом *mcMining*. Компонент СУБД PostgreSQL *SPI (Server Programming Interface)* обеспечивает низкоуровневые функции доступа к данным.

Компонент *pgMining* состоит из двух основных подсистем: *Backend* и *Frontend*. Подсистема *Backend* реализует внутренний интерфейс и состоит из следующих модулей: *Wrapper* и *Cache manager*. Модуль *Wrapper* обеспечивает обертки для функций библиотеки *mcMining*, позволяющие осуществлять вызов параллельного аналитического алгоритма и сохранять результат анализа в базе данных в виде промежуточного описания в формате JSON. JSON-представление включает в себя описание не только результатов анализа, но также и его метаданных (значения целевой функции в процессе вычислений, время выполнения алгоритма в различных фазах его работы и др.), что может быть использовано для интеллектуального анализа научных данных.

Модуль *Cache manager* обеспечивает буферный пул для долговременного хранения в оперативной памяти предварительно вычисляемых структур данных, которые далее могут быть многократно использованы для интеллектуального анализа. Типичным примером такой структуры

данных может служить матрица расстояний, в которой хранятся значения схожести между каждой парой элементов исходного множества. Матрица расстояний может быть многократно использована для выполнения кластеризации данных или классификации по принципу ближайших соседей с различными параметрами (количество кластеров, количество «соседей» и др.).

Подсистема *Frontend* представляет собой интерфейс прикладного программиста баз данных и обеспечивает функции, позволяющие выполнять анализ данных внутри СУБД и извлекать результаты анализа, сохраненные в виде промежуточного описания в формате JSON.

Инкапсуляция в библиотеке *mcMining* деталей параллельного исполнения аналитических функций от СУБД позволяет применить данный подход в других свободных СУБД (например, MySQL). Разработка соответствующих версий модулей *Frontend* и *Backend* потребует относительно механических переработок в исходных текстах библиотеки *pgMining*.

Алг. 15 PBlockwise(in A, B, *block*; out M)

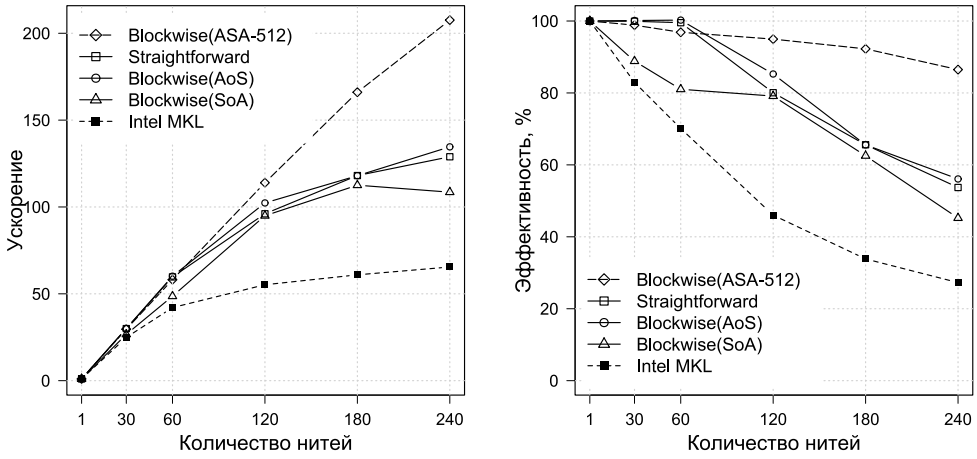
```
1: #pragma omp parallel for
2: for i from 1 to n do
3:     for j from 1 to ⌈m/block⌉ do
4:         sum ← 0̄
5:         for k from 1 to d do
6:             for ℓ from 1 to block do
7:                 sum(ℓ) ← sum(ℓ) + (A(i,k) − B(j+k,ℓ))²
8:         for k from 1 to block do
9:             M(i, j·block + k) ← sum(k)
10: return M
```

Параллельный *алгоритм PBlockwise* (см. алг. 15) вычисляет матрицу расстояний между точками евклидова пространства из двух непустых конечных (возможно, совпадающих) множеств. Строки матриц $A \in \mathbb{R}^{n\times d}$ и $B \in \mathbb{R}^{m\times d}$ представляют собой точки первого и второго множества соответственно. *Матрица евклидовых расстояний* $M \in \mathbb{R}^{n\times m}$ состоит из строк $M(i,\cdot) \in \mathbb{R}^m$, где $M(i,j) = \|A(i,\cdot) - B(j,\cdot)\|^2$ и $\|\cdot\|$ означает евклидову норму.

Схема вычисления евклидовых расстояний между точками множеств изменяется таким образом, чтобы было векторизовано как можно большее количество операций. *PBlockwise* итеративно вычисляет несколько расстояний от одной точки из первого множества до *block* точек из второго множества, где *block* — параметр алгоритма, подбираемый эмпири-

чески. Вычисления организуются с помощью двух вложенных циклов. Внешний цикл перебирает первое множество точек и распараллеливается с помощью директивы OpenMP. Внутренний цикл перебирает блоки второго множества точек и выполняет цикл вычислений евклидова расстояния по координатам точек блока, компилируемый в две векторные операции. Сохранение вычисленных расстояний осуществляется с помощью цикла, который компилируется в одну векторную операцию.

Для эффективной векторизации вычислений алгоритм выравнивает данные в памяти: мощность второго множества m должна быть кратна количеству блоков *block*, иначе матрица B дополняется соответствующим количеством фиктивных нулевых строк; параметр *block* должен быть кратен ширине векторного регистра ускорителя Intel MIC. Для уменьшения количества кэш-промахов *PBlockwise* использует усложненную компоновку данных в памяти. Для представления матрицы B в памяти используется стандартная схема ASA (Array of Structures of Arrays), в соответствии с которой B хранится в виде массива из $\lceil \frac{m}{block} \rceil$ элементов, где элемент состоит из d массивов, каждый из которых содержит *block* вещественных чисел.



(a) ускорение (b) параллельная эффективность

Рис. 10. Масштабируемость алгоритма *PBlockwise*

Эксперименты с алгоритмом *PBlockwise* на суперкомпьютере ИВМиМГ СО РАН показали (см. рис. 10; стандартный набор данных MixSim, $n = m = 35 \cdot 2^{10}$, d=5, *block*=512), что предложенный алгоритм хорошо масштабируется. При сравнении с лучшими аналогами алгоритм *PBlockwise* показал быстродействие, опережающее конкурентов.

Параллельный *алгоритм PPAM* (см. алг. 16) выполняет кластеризацию точек евклидова пространства на основе техники медоидов. Медо-

Алг. 16 PPAM(in X, k, *block*, L; out C)

1: $X_{ASA} \leftarrow \text{PERMUTE}(X, block)$
2: $M \leftarrow \text{PBLOCKWISE}(X, X_{ASA}, block)$
3: $C \leftarrow \text{PBUILD}(M, k, L)$
4: **repeat**
5: $\quad \{T_{min}, c_{min}, x_{min}\} \leftarrow \text{PSWAP}(C, M, L)$
6: $\quad C_{c_{min}} \leftarrow x_{min}$
7: **until** $T_{min} < 0$
8: **return** C

ид представляет собой объект исходного множества, который находится ближе остальных к центру кластера. Техника медоидов обеспечивает повышение робастности алгоритма (устойчивости к выбросам и шумам в данных). В отличие от последовательного алгоритма *PAM*, используется предвычисление матрицы расстояний с помощью описанного выше алгоритма *PBlockwise*, для чего объекты X переставляются в соответствии с способом компоновки данных ASA.

Далее выполняется стадия *Build* алгоритма. Вектор расстояний от точек до ближайшего медоида инициализируется значением $+\infty$. Далее выполняется цикл по всем кластерам, в котором осуществляется поиск медоида, минимизирующего значение целевой функции. Тело указанного цикла организовано следующим образом. Выполняется сканирование всех кластеризуемых объектов, и для каждого объекта вычисляет значение целевой функции относительно данного объекта и ранее найденных медоидов. Объект, на котором достигнут минимум целевой функции, добавляется в множество медоидов. Итогом данной стадии является начальная расстановка медоидов, используемая на стадии *Swap*.

Затем на стадии *Swap* циклически выполняются следующие действия. Находится один из кластеризуемых объектов и один из медоидов, перестановка которых минимизирует значение целевой функции, и указанные объекты меняются местами. Цикл выполняется до тех пор, пока возможно улучшение значения целевой функции (найдены все медоиды, такие, что любая перестановка любого из них с кластеризуемым объектом ухудшит значение целевой функции). Алгоритм возвращает индексы медоидов в исходном множестве кластеризуемых объектов. Объект исходного множества принадлежит тому кластеру, медоид которого является ближайшим к данному объекту.

В реализации стадий алгоритма *Build* и *Swap* (см. алг. 17) используется техника *тайлинга*, предполагающая разбиение множества итераций

Алг. 17 Реализация стадий алгоритма *PPAM*

PBuild (in M, k, L; out C)

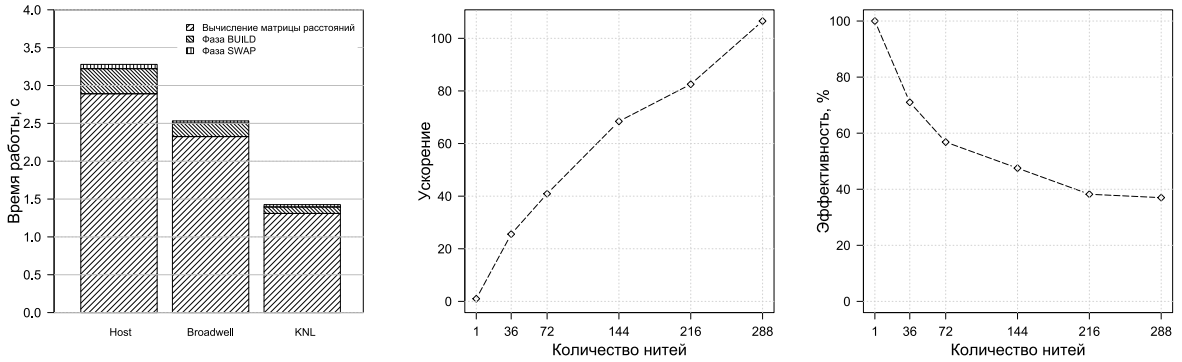
```
1:  D ← overline(+∞)
2:  for ℓ ∈ 1..k do
3:      min ← +∞
4:      #pragma omp parallel for
5:      reduction(min(sum) : {min, C_ℓ})
6:      for i ∈ 1..n tile L do
7:          sum ← 0
8:          for j ∈ 1..n tile L do
9:              if D_j is +∞ then
10:                 sum ← sum + M(i, j)
11:             else if D_j > M(i, j) then
12:                 sum ← sum+
                    M(i, j) − D_j
13:         if sum < min then
14:             min ← sum
15:             C_ℓ ← i
16:     #pragma omp parallel for
17:     for i ∈ 1..n do
18:         D_i ← min(D_i, M(C_ℓ, i))
19: return C
```

PSwap (in C, M, L; out $\{T_{min}, c_{min}, x_{min}\}$)

```
1:  D ← overline(+∞); S ← overline(+∞)
2:  #pragma omp parallel for
3:  for h ∈ 1..n tile L do
4:      for i ∈ 1..k do
5:          if D_h > M(h, C_i) then
6:              S_h ← D_h
7:              D_h ← M(h, C_i)
8:          else if S_h > M(h, C_i) then
9:              S_h ← M(h, C_i)
10: #pragma omp parallel for
11: reduction(min(T) : {T_min, c_min, o_min})
12: for h ∈ 1..n tile L do
13:     for i ∈ 1..k do
14:         T ← 0
15:         for ℓ ∈ 1..n tile L do
16:             T ← T+
                Delta(M, ℓ, C_i, h, D_ℓ, S_ℓ)
17:         if T < T_min then
18:             T_min ← T
19:             c_min ← i; x_min ← h
20: return {T_min, c_min, x_min}
```

счетчика исходного цикла на непересекающиеся подмножества меньшей мощности. Обрабатываемый массив, в свою очередь, разбивается на блоки *(тайлы)*, которые могут быть помещены в кэш-память процессора (размер тайла L — параметр алгоритма). Тайлинг снижает количество кэш-промахов при выполнении вычислений в теле цикла, увеличивая эффективность алгоритма. Циклы распараллеливаются с помощью директив OpenMP.

Эксперименты с алгоритмом *PPAM* на суперкомпьютере ИВМиМГ СО РАН показали (см. рис. 11; стандартный набор данных FCS Human, $n = 18 \cdot 2^{10}$, $d = 423$), что предложенный алгоритм демонстрирует более высокую производительность на ускорителях Intel MIC, чем на системах из двух многоядерных процессоров Intel Xeon, обеспечивая при этом ускорение 40 раз и параллельную эффективность 60%. В экспериментах



(a) производительность (b) ускорение (c) эффективность

Рис. 11. Производительность и масштабируемость алгоритма *PPAM*

по исследованию качества кластеризации данных с шумами и выбросами, обеспечиваемого алгоритмом *PPAM*, использована мера *силуэтного коэффициента*

$$\begin{aligned} \mathcal{S}(X) &:= \frac{1}{n}\sum_{x\in X}\frac{b(x)-a(x)}{\max\{a(x),b(x)\}}, \\ a(x) &:= \frac{\sum_{y\in C_i\wedge x\neq y}\rho(x,y)}{|C_i|-1}, \\ b(x) &:= \min_{C_j:\ 1\leqslant j\leqslant k\wedge j\neq i}\frac{\sum_{y\in C_j}\rho(x,y)}{|C_j|}. \end{aligned} \tag{30}$$

Алгоритм *PPAM* показывает более высокое качество кластеризации при работе с зашумленными данными, чем алгоритмы кластеризации, не использующие технику медоидов.

В пятой главе, «Интеграция в СУБД фрагментного параллелизма», предлагаются методы, позволяющие внедрить фрагментный параллелизм в свободную реляционную СУБД посредством модификации ее открытых исходных кодов. Описаны архитектура и принципы реализации параллельной СУБД *PargreSQL* на платформе вычислительного кластера, получаемой посредством распараллеливания свободной СУБД PostgreSQL. Представлены результаты вычислительных экспериментов, исследующих эффективность предложенных методов.

Оригинальная СУБД PostgreSQL рассматривается *как подсистема*, входящая в СУБД *PargreSQL* (см. рис. 12). Для реализации *PargreSQL* необходимо как внести изменения в исходный код некоторых подсистем PostgreSQL, так и разработать новые подсистемы, не затрагивающие исходный код оригинальной СУБД.

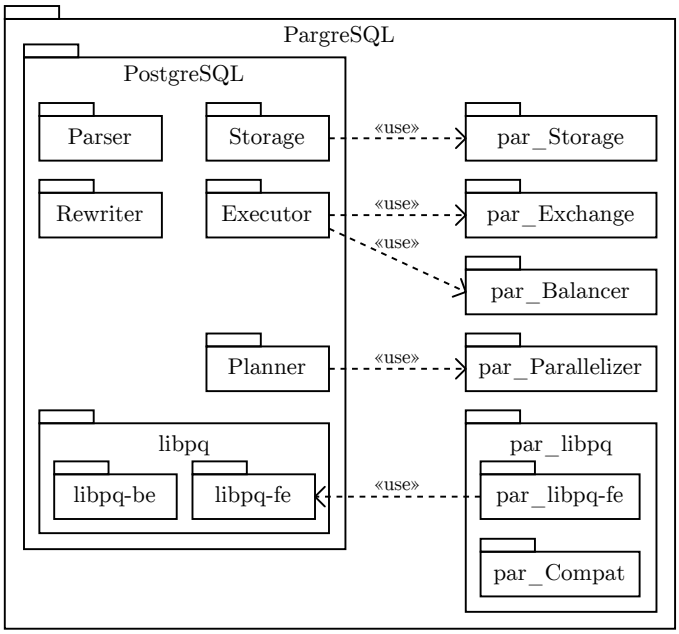


Рис. 12. Архитектура СУБД PargreSQL

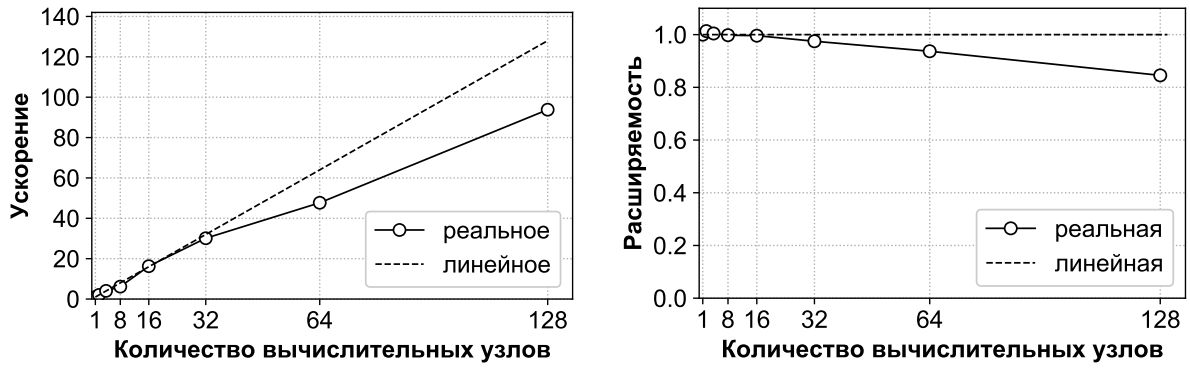
Путем *изменения исходных кодов* подсистем `Storage`, `Executor` и `Planner` внедряются следующие подсистемы. Подсистема *par_Storage* обеспечивает хранение и обработку метаданных о фрагментации отношений. Подсистема *par_Balancer* выполняет динамическую балансировку загрузки серверных процессов СУБД. Подсистема *par_Exchange* реализует оператор *Exchange*, который обеспечивает пересылку кортежей между экземплярами СУБД во время выполнения запроса. Подсистема *par_Parallelizer* добавляет в нужные места последовательного плана запроса операторы *Exchange*.

Оператор *Exchange* с точки зрения реляционной алгебры представляет собой пустой оператор и инкапсулирует передачу кортежей между узлами вычислительного кластера. Оператор *Exchange* является составным и имеет два следующих атрибута. Порт представляет собой порядковый номер оператора в запросе. Функция пересылки для каждого кортежа вычисляет номер узла вычислительного кластера, где этот кортеж должен обрабатываться. Оператор *Split* вычисляет функцию пересылки для входного кортежа. Если кортеж должен быть обработан на текущем узле, он передается оператору *Merge*, иначе кортеж передается оператору *Scatter* для пересылки на другие узлы. Оператор *Gather* принимает кортежи с других узлов и направляет их в оператор *Merge*. Оператор *Merge* поочередно объединяет потоки кортежей, поступающих от операторов *Gather* и *Split*.

Для реализации фрагментации таблиц модифицируется блок синтаксического разбора СУБД PostgreSQL и в команду создания таблицы добавляется конструкция `WITH (FRAG_ATTR=`*attr*`)`, позволяющая задать в качестве атрибута фрагментации одну из целочисленных колонок таблицы. Соответствующим образом модифицируется словарь СУБД и обеспечивается хранение атрибута фрагментации каждой таблицы базы данных. В СУБД *PargreSQL* функция фрагментации таблицы `T` имеет вид $\varphi(t) = t.attr \ mod \ P$, где P — количество вычислительных улов кластерной системы.

PargreSQL включает в себя следующие *новые подсистемы*. Библиотека *par_ libpq-fe* представляет собой надстройку над стандартной библиотекой `libpq-fe` PostgreSQL, и реализует отправку запроса всем серверам, на которых запущены экземпляры *PargreSQL*. При разработке приложения СУБД *PargreSQL* программист подключает библиотеку *par_ libpq* и создает на клиенте конфигурационный файл с параметрами доступа к каждому экземпляру СУБД (IP-адрес, порт, имя пользователя, пароль и др.).

Подсистема *par_ Compat* реализует прозрачное подключение библиотеки *par_ libpq-fe* к приложению пользователя и обеспечивает использование параллельной СУБД *PargreSQL* вместо оригинальной последовательной СУБД PostgreSQL без внесения существенных изменений в исходный код приложения.



(a) ускорение (b) расширяемость

Рис. 13. Масштабируемость СУБД *PargreSQL*

Эксперименты по исследованию ускорения и расширяемости СУБД PargreSQL на суперкомпьютере «Торнадо ЮУрГУ» заключались в выполнении естественного соединения двух отношений. При исследовании ускорения отношения имели размеры 300 млн. и 7.5 млн. кортежей соответственно. При исследовании расширяемости размеры отношений уве-

личивались пропорционально количеству используемых узлов кластера с множителем 12 млн. и 0.3 млн. кортежей соответственно. *PargreSQL* демонстрирует (см. рис. 13) ускорение и расширяемость, близкие к линейным. Эксперименты по исследованию производительности СУБД на стандартных тестах консорциума TPC (Transaction Processing Council) заключались в измерении производительности прототипа СУБД *PargreSQL* на последовательности OLTP-запросов, моделирующих складской учет. В тесте использовалось от одного до 30 параллельно работающих клиентов, выполняющих запросы к СУБД PargreSQL, запущенной на 12 узлах. Производительность, показанная в данном эксперименте, позволила *PargreSQL* попасть в пятерку лидеров рейтинга TPC-C среди параллельных СУБД для кластеров (ноябрь 2013 г.).

В заключении в краткой форме излагаются итоги выполненного диссертационного исследования; представляются отличия данной работы от ранее выполненных родственных работ других авторов; формулируются основные результаты диссертационной работы, выносимые на защиту; приводятся данные о публикациях и апробациях автора по теме диссертации; рассматриваются направления дальнейших исследований в данной области.

Основные результаты диссертационной работы

На защиту выносятся следующие новые научные результаты:

1. Разработан и исследован комплекс параллельных алгоритмов интеллектуального анализа данных средствами СУБД на кластерных вычислительных системах с многоядерными ускорителями:

   1) параллельный алгоритм поиска похожих подпоследовательностей временного ряда для кластеров с многоядерными ускорителями;
   2) параллельный алгоритм поиска диссонансов временного ряда для многоядерных ускорителей;
   3) алгоритм кластеризации графа для параллельной СУБД на основе фрагментного параллелизма;
   4) алгоритм нечеткой кластеризации данных для параллельной СУБД на основе фрагментного параллелизма;
   5) параллельный алгоритм кластеризации данных для многоядерных ускорителей;

6) параллельный алгоритм поиска частых наборов для многоядерных ускорителей.

2. Предложен подход к внедрению параллельных алгоритмов интеллектуального анализа данных в реляционные СУБД.

3. Предложен метод внедрения фрагментного параллелизма в последовательные СУБД с открытым исходным кодом.

Публикации по теме диссертации

Публикации в журналах из перечня ВАК

1. *Пан К.С., Цымблер М.Л.* Внедрение фрагментного параллелизма в СУБД с открытым кодом // Программирование. 2015. Т. 41, № 5. С. 18–32.

2. *Пан К.С., Соколинский Л.Б., Цымблер М.Л.* Интеграция параллелизма в СУБД с открытым кодом // Открытые системы. СУБД. 2013. № 9. С. 56–58.

3. *Пан К.С., Цымблер М.Л.* Разработка параллельной СУБД на основе последовательной СУБД PostgreSQL с открытым исходным кодом // Вестник ЮУрГУ. Серия: Математическое моделирование и программирование. 2012. № 18(277). Вып. 12. С. 112–120.

4. *Мовчан А.В., Цымблер М.Л.* Обнаружение подпоследовательностей во временных рядах // Открытые системы. СУБД. 2015. № 2. С. 42–43.

5. *Миниахметов Р.М., Цымблер М.Л.* Интеграция алгоритма кластеризации Fuzzy c-Means в PostgreSQL // Вычислительные методы и программирование: Новые вычислительные технологии. 2012. Т. 13. С. 46–52.

6. *Пан К.С., Цымблер М.Л.* Параллельный алгоритм решения задачи анализа рыночной корзины на процессорах Cell // Вестник ЮУрГУ. Серия: Математическое моделирование и программирование. 2010. № 16(192). Вып. 5. С. 48–57.

7. *Речкалов Т.В., Цымблер М.Л.* Параллельный алгоритм вычисления матрицы Евклидовых расстояний для многоядерного процессора Intel Xeon Phi Knights Landing // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2018. Т. 7, № 3. С. 65–82. DOI: 10.14529/cmse180305

8. *Краева Я.А., Цымблер М.Л.* Совместное использование технологий MPI и OpenMP для параллельного поиска похожих подпоследовательностей в сверхбольших временных рядах на вычислительном кластере с узлами на базе многоядерных процессоров Intel Xeon Phi Knights Landing // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 1. С. 29–43. DOI: 10.26089/NumMet.v20r104

9. *Цымблер М.Л.* Параллельный алгоритм поиска диссонансов временного ряда для многоядерных ускорителей // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 3. С. 211–223. DOI: 10.26089/NumMet.v20r320

10. *Речкалов Т.В., Цымблер М.Л.* Параллельный алгоритм кластеризации данных для многоядерных ускорителей Intel MIC // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 2. С. 104–115. DOI: 10.26089/NumMet.v20r211

11. *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

12. *Цымблер М.Л.* Обзор методов интеграции интеллектуального анализа данных в СУБД // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 2. С. 32–62. DOI: 10.14529/cmse190203

Статьи в изданиях, индексируемых в Scopus и Web of Science

13. *Zymbler M.* Parallel Algorithm for Frequent Itemset Mining on Intel Many-core Systems // Journal of Computing and Information Technology. 2018. Vol. 26, No. 4. P. 209–221. DOI: 10.20532/cit.2018.1004382

14. *Zymbler M.* Best-match Time Series Subsequence Search on the Intel Many Integrated Core Architecture // Proceedings of the 19th East-European Conference on Advances in Databases and Information Systems, ADBIS 2015 (September 8–11, 2015, Poitiers, France). Lecture Notes in Computer Science. Springer, 2015. Vol. 9282. P. 275–286. DOI: 10.1007/978-3-319-23135-8_19

15. *Kraeva Ya., Zymbler M.* Scalable Algorithm for Subsequence Similarity Search in Very Large Time Series Data on Cluster of Phi KNL // 20th International Conference on Data Analytics and Management in Data

Intensive Domains, DAMDID/RCDL 2018, Moscow, Russia, October 9–12, 2018, Revised Selected Papers. Communications in Computer and Information Science. 2019. Vol. 1003. P. 149–164. DOI: 10.1007/978-3-030-23584-0_9

16. *Movchan A., Zymbler M.* Parallel Algorithm for Local-best-match Time Series Subsequence Similarity Search on the Intel MIC Architecture // Procedia Computer Science. 2015. Vol. 66. P. 63–72. DOI: 110.1016/j.procs.2015.11.009

17. *Rechkalov T., Zymbler M.* Accelerating Medoids-based Clustering with the Intel Many Integrated Core Architecture // Proceedings of the 9th International Conference on Application of Information and Communication Technologies, AICT'2015 (October 14–16, 2015, Rostov-on-Don, Russia). IEEE, 2015. P. 413–417. DOI: 10.1109/ICAICT.2015.7338591

18. *Rechkalov T., Zymbler M.* Integrating DBMS and Parallel Data Mining Algorithms for Modern Many-Core Processors // Revised Selected Papers of the 19th International Conference on Data Analytics and Management in Data Intensive Domains, DAMDID/RCDL 2017 (October 10–13, 2017, Moscow, Russia). Communications in Computer and Information Science. Springer, 2018. Vol. 822. P. 230–245. DOI: 10.1007/978-3-319-96553-6_17

19. *Pan C., Zymbler M.* Taming Elephants, or How to Embed Parallelism into PostgreSQL // Proceedings of the 24th International Conference on Database and Expert Systems Applications, DEXA 2013 (August 26–29, 2013, Prague, Czech Republic). Lecture Notes in Computer Science. Springer, 2013. Vol. 8055. Part I. P. 153–164. DOI: 10.1007/978-3-642-40285-2_15

20. *Pan C., Zymbler M.* Very Large Graph Partitioning by Means of Parallel DBMS // Proceedings of the 17th East-European Conference on Advances in Databases and Information Systems, ADBIS 2013 (September 1–4, 2013, Genoa, Italy). Lecture Notes in Computer Science. Springer, 2013. Vol. 8133. P. 388–399. DOI: 10.1007/978-3-642-40683-6_29

21. *Rechkalov T., Zymbler M.* A Study of Euclidean Distance Matrix Computation on Intel Many-core Processors // Revised Selected Papers of the 12th International Conference, PCT 2018 (April 2–6, 2018, Rostov-on-Don, Russia). Communications in Computer and Information Science. Springer, 2018. Vol. 910. P. 200–215. DOI: 10.1007/978-3-319-99673-8_15

Свидетельства о регистрации программы

22. *Мовчан А.В., Цымблер М.Л.* Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Программный комплекс для поиска похожих подпоследовательностей временного ряда на многоядерном сопроцессоре Intel Xeon Phi» № 2015618537 от 11.08.2015.
23. *Соколинский Л.Б., Цымблер М.Л., Пан К.С., Медведев А.А.* Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Параллельная СУБД PargreSQL» № 2012614599 от 23.05.2012.
24. *Цымблер М.Л., Пан К.С.* Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Программный комплекс для решения задачи анализа рыночной корзины на многоядерных процессорах с поддержкой векторных вычислений» № 2011610732 от 11.01.2011.

Исходные тексты программ, реализующих методы и алгоритмы, разработанные в рамках диссертационного исследования, свободно доступны в сети Интернет по адресу: https://github.com/mzym/thesis/.

Диссертационное исследование выполнено при финансовой поддержке Российского фонда фундаментальных исследований (грант № 17-07-00463) и Министерства науки и высшего образования РФ (государственное задание 2.7905.2017/8.9).

Формат 60 × 84 1/16. Бумага офсетная. Печать офсетная. Авт. л. 2,0. Уч.-изд. л. 2,4.
Подписано в печать 01.11.2019. Тираж 100 экз.
Издательский центр ЮУрГУ. 454080, г. Челябинск, пр. Ленина, 76