УДК 004.272.25; 004.421; 004.032.24 doi 10.26089/NumMet.v20r320

ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ ПОИСКА ДИССОНАНСОВ ВРЕМЕННОГО РЯДА ДЛЯ МНОГОЯДЕРНЫХ УСКОРИТЕЛЕЙ

М. Л. Цымблер[1]

Диссонанс является уточнением понятия аномальной подпоследовательности (существенно непохожей на остальные подпоследовательности) временного ряда. Задача поиска диссонанса встречается в широком спектре предметных областей, связанных с временными рядами: медицина, экономика, моделирование климата и др. В работе предложен новый параллельный алгоритм поиска диссонанса во временном ряде на платформе многоядерного ускорителя для случая, когда входные данные могут быть размещены в оперативной памяти. Алгоритм использует возможность независимого вычисления евклидовых расстояний между подпоследовательностями ряда. Алгоритм состоит из двух этапов: подготовка данных и поиск. На этапе подготовки выполняется построение вспомогательных матричных структур данных, обеспечивающих распараллеливание и векторизацию вычислений. На стадии поиска осуществляется нахождение диссонанса с помощью построенных структур данных. Выполнена реализация алгоритма для ускорителей архитектур Intel MIC (Many Integrated Core) и NVIDIA GPU, распараллеливание выполнено с помощью технологий программирования OpenMP и OpenAcc соответственно. Представлены результаты вычислительных экспериментов, подтверждающих масштабируемость разработанного алгоритма.

Ключевые слова: временной ряд, поиск диссонансов, параллельный алгоритм, векторизация вычислений, OpenMP, OpenAcc, Intel Xeon Phi, NVIDIA GPU.

1. Введение. Поиск аномальных подпоследовательностей (существенно непохожих на все остальные подпоследовательности) временного ряда является одной из наиболее востребованных задач интеллектуального анализа временных рядов и в настоящее время применяется в широком спектре практических приложений: моделирование климата, финансовые прогнозы, медицинские исследования и др.

В работе [12] Кеогом (Keogh) и др. предложен термин диссонанс, уточняющий понятие аномальной подпоследовательности временного ряда. *Диссонанс* определяется как подпоследовательность ряда, которая имеет максимальное расстояние (минимальную схожесть) до своего ближайшего соседа. Ближайшим соседом подпоследовательности является та подпоследовательность ряда, которая не пересекается с данной и имеет минимальное расстояние до нее (максимальную схожесть) в смысле выбранной меры схожести.

Понятие диссонанса привлекательно для интеллектуального анализа временных рядов в качестве подхода к нахождению аномалий, поскольку требует только одного интуитивно понятного параметра — длина подпоследовательности, — в отличие от большинства других алгоритмов поиска аномалий, требующих от трех до семи параметров, не все из которых являются интуитивно понятными [13].

Последовательный алгоритм *HOTSAX* [12] поиска диссонансов использует кодирование временного ряда с помощью техники символьной агрегатной аппроксимации (Symbolic Aggregate ApproXimation, SAX) [16] и евклидово расстояние в качестве меры схожести. Алгоритм осуществляет перебор всех пар подпоследовательностей ряда, вычисляя евклидово расстояние между ними, и находит максимум среди расстояний до ближайшего соседа. Подпоследовательность с максимальным расстоянием до ближайшего соседа является диссонансом. При переборе бесперспективные подпоследовательности (заведомо не являющиеся диссонансами) отбрасываются без вычисления расстояний. Бесперспективными являются подпоследовательности, которые имеют соседа, расположенного ближе текущего максимума расстояний до всех ближайших соседей. HOTSAX осуществляет перебор в соответствии с определенной эвристикой, которая позволяет отбрасывать больше бесперспективных кандидатов.

Целью данного исследования является распараллеливание поиска диссонансов для многоядерных ускорителей архитектур Intel MIC (Many Integrated Core) [7] и GPU [18]. Увеличение количества вычислительных ядер вместо тактовой частоты является одной из современных тенденций развития процессорной техники. Ускорители обеспечивают от десятков до сотен процессорных ядер и значительно опережают

[1] Южно-Уральский государственный университет, Высшая школа электроники и компьютерных наук, просп. Ленина, 76, 454080, Челябинск; начальник отдела, e-mail: mzym@susu.ru

© Научно-исследовательский вычислительный центр МГУ им. М. В. Ломоносова

традиционные процессоры по производительности. Одной из ключевых особенностей ускорителей является поддержка векторизации циклов. Под векторизацией понимается способность компилятора заменить несколько скалярных операций в теле цикла с фиксированным количеством повторений в одну векторную операцию [4].

В настоящей статье предлагается новый параллельный алгоритм *MDD* (Many-core Discord Discovery) для поиска диссонансов временного ряда на платформе многоядерных ускорителей архитектур Intel MIC и NVIDIA GPU для случая, когда входные данные могут быть размещены в оперативной памяти. Статья организована следующим образом. В разделе 2 приводится формальная постановка задачи и кратко описан последовательный алгоритм HOTSAX, используемый в качестве базиса нового параллельного алгоритма. Раздел 3 представляет собой обзор связанных работ. Раздел 4 содержит описание предлагаемого алгоритма MDD. В разделе 5 описаны вычислительные эксперименты, исследующие эффективность предложенного алгоритма. Заключение резюмирует результаты, полученные в рамках исследования.

2. Постановка задачи.

2.1. Формальные определения и нотация. Дадим определения используемых терминов в соответствии с работой [12].

Временной ряд (time series) T представляет собой последовательность хронологически упорядоченных вещественных значений: $T = (t_1, t_2, \ldots, t_m)$, $t_i \in \mathbb{R}$. Число m обозначается как $|T|$ и называется длиной ряда.

Подпоследовательность (subsequence) $T_{i,n}$ временного ряда T представляет собой непрерывное подмножество T из n элементов начиная с позиции i: $T_{i,n} = (t_i, t_{i+1}, \ldots, t_{i+n-1})$, $1 \leqslant n \ll m$, $1 \leqslant i \leqslant m-n+1$.

Множество всех подпоследовательностей ряда T, имеющих длину n, обозначим как S_T^n. Обозначим через N мощность указанного множества, т.е. $N := \left|S_T^n\right| = m - n + 1$.

Подпоследовательности $T_{i,n}$ и $T_{j,n}$ ряда T называются *непересекающимися (non-self match)*, если $|i-j| \geqslant n$. Подпоследовательность, которая является непересекающейся к данной подпоследовательности C, обозначается как M_C.

Подпоследовательность D ряда T является *диссонансом (discord)*, если

$$\forall C,\ M_C \in T \qquad \min\bigl(\mathrm{ED}(D, M_D)\bigr) > \min\bigl(\mathrm{ED}(C, M_C)\bigr).$$

Другими словами, некая подпоследовательность ряда является диссонансом, если она имеет максимальное расстояние до своего ближайшего соседа — наиболее близкой непересекающейся подпоследовательности. Для поиска диссонансов используется *евклидово расстояние*, определяемое следующим образом. Пусть имеются подпоследовательности X и Y длины n временного ряда T, тогда расстояние ED между X и Y вычисляется как

$$ED(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}\,. \qquad (1)$$

2.2. Последовательный алгоритм. Алгоритм HOTSAX [12] состоит из двух этапов: подготовка и поиск. На этапе подготовки выполняется нормализация ряда, сжатие и кодирование подпоследовательностей ряда. На этапе поиска алгоритм строит префиксное дерево из подготовленных данных и выполняет его обход для перебора подпоследовательностей и нахождения диссонанса.

На этапе подготовки алгоритм HOTSAX выполняет следующие преобразования исходных данных: z-нормализация ряда, сжатие нормализованных подпоследовательностей с помощью кусочной агрегатной аппроксимации и кодирование сжатых подпоследовательностей с помощью символьной агрегатной аппроксимации.

Z-нормализация временного ряда T представляет собой временной ряд $\hat{T} = \bigl(\hat{t}_1, \ldots, \hat{t}_m\bigr)$, элементы которого вычисляются следующим образом:

$$\hat{t}_i = \frac{t_i - \mu}{\sigma}\,, \qquad 1 \leqslant i \leqslant m; \qquad \mu = \frac{1}{m}\sum_{i=1}^{m} t_i; \qquad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m} t_i^2 - \mu^2. \qquad (2)$$

После нормализации среднее арифметическое элементов временного ряда приблизительно равно 0, а среднеквадратичное отклонение близко к 1.

Подпоследовательности ряда подвергаются сжатию с помощью *кусочной агрегатной аппроксимации (PAA, Piecewise Aggregate Approximation)* [16]. *PAA-кодом* подпоследовательности $C = (c_1, c_2, \ldots, c_n)$ является вектор $\overline{C} = \bigl(\overline{c}_1, \overline{c}_2, \ldots, \overline{c}_w\bigr)$, где *степень сжатия* $w \leqslant n$ — параметр, а координаты вектора

вычисляются следующим образом:

$$\overline{c}_i = \frac{w}{n} \sum_{j=n\cdot(j-1)/w+1}^{n\cdot j/w} c_j. \tag{3}$$

Далее сжатая подпоследовательность подвергается кодированию с помощью *символьной агрегатной аппроксимации (SAX, Symbolic Aggregate ApproXimation)* [16]. *SAX-кодом* подпоследовательности $C = (c_1, c_2, \ldots, c_n)$ является *слово* $\hat{C} = \big(\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_w\big)$, получаемое следующим образом. Пусть имеется символьный *алфавит кодирования* $\mathcal{A} = (\alpha_1, \alpha_2, \ldots, \alpha_{|\mathcal{A}|})$, где запись $|\mathcal{A}|$ означает мощность алфавита и $\alpha_1 = \ 'a'$, $\alpha_2 = \ 'b'$, $\alpha_3 = \ 'c'$ и т.д. Тогда

$$\hat{c}_i = \alpha_i \Leftrightarrow \beta_{j-1} \leqslant \hat{c}_i < \beta_j. \tag{4}$$

В формуле (4) числа β_i представляют собой *точки разделения (breakpoints)* [16], определяемые как упорядоченный по возрастанию список $\mathcal{B} := (\beta_0, \beta_1, \ldots, \ \beta_{|\mathcal{A}|-1}, \beta_{|\mathcal{A}|})$, где $\beta_0 := -\infty$ и $\beta_{|\mathcal{A}|} := +\infty$, а площадь под кривой нормального распределения $N(0,1)$ между β_i и β_{i+1} равна $\dfrac{1}{|\mathcal{A}|}$. Точки разделения для различных значений мощности алфавита кодирования могут быть получены из статистических таблиц [12].

Значения параметров $w = 4$ (степень сжатия) и $|\mathcal{A}| = 4$ (мощность алфавита кодирования), как показывают эксперименты [12], хорошо подходят при поиске диссонансов во временных рядах из различных предметных областей.

На этапе поиска выполняется подсчет частоты возникновения каждого из полученных SAX-кодов подпоследовательностей и на основе полученных частот строится префиксное дерево, используемое для перебора подпоследовательностей и нахождения диссонанса.

Используемое алгоритмом *префиксное дерево (trie)* [8] представляет собой дерево, каждое ребро которого помечено символом алфавита $\mathcal{A}$ таким образом, что для любого узла все ребра, соединяющие этот узел с его сыновьями, помечены разными символами. Строка SAX-кода, соответствующая листу префиксного дерева, получается выписыванием подряд символов, помечающих ребра на пути от корня до листа. В листе префиксного дерева хранится упорядоченный по возрастанию список индексов подпоследовательностей исходного ряда, имеющих соответствующий SAX-код.

Алгоритм 1. HOTSAX(IN $T,\ n$; OUT $pos_{bsf}, dist_{bsf}$)

1: $dist_{bsf} \leftarrow 0$; $dist_{\min} \leftarrow \infty$
2: **for** $C_i \in (NearDiscords \cdot Others)$ **do**
3:     **for** $C_j \in \big(Neighbors(C_i) \cdot Strangers(C_i)\big)$ **do**
4:         $dist \leftarrow \mathrm{ED}(C_i, C_j)$
5:         **if** $dist < dist_{bsf}$ **then**
6:             **break**
7:         $dist_{\min} \leftarrow \min(dist, dist_{\min})$
8:     $dist_{bsf} \leftarrow \max(dist_{\min}, dist_{bsf})$; $pos_{bsf} \leftarrow i$
9: **return** $\{pos_{bsf}, dist_{bsf}\}$

Последовательная реализация поиска диссонансов, предложенная Кеогом и др. [12], представлена в алгоритме 1. Алгоритм получает на входе временной ряд и длину искомого диссонанса и возвращает индекс найденного диссонанса и его расстояние до ближайшего соседа.

Алгоритм осуществляет перебор всех пар подпоследовательностей ряда, вычисляя евклидово расстояние между ними, и находит максимум среди расстояний до ближайшего соседа. Подпоследовательность с максимальным расстоянием до ближайшего соседа является диссонансом. Подпоследовательности ряда перебираются посредством двух вложенных циклов. При переборе бесперспективные подпоследовательности (заведомо не являющиеся диссонансами) отбрасываются без вычисления расстояний. Бесперспективной является подпоследовательность, которая имеет соседа, расположенного ближе текущего максимума расстояний до всех ближайших соседей.

HOTSAX осуществляет перебор в соответствии с эвристикой, которая позволят отбрасывать больше бесперспективных подпоследовательностей. Для этого определяются следующие четыре множества подпоследовательностей исходного ряда. Множество *потенциальных диссонансов* $NearDiscords$ содержит

подпоследовательности, SAX-коды которых встречаются наиболее редко. Множество $Others$ содержит *остальные* подпоследовательности.

Для данной подпоследовательности C множества *соседей* $Neighbors(C)$ и *чужаков* $Strangers(C)$ содержат подпоследовательности с SAX-кодом, совпадающим или отличным от SAX-кода C соответственно. Эвристика предписывает во внешнем цикле перебирать сначала потенциальные диссонансы, затем — остальные подпоследовательности. Во внутреннем цикле сначала перебираются соседи, затем — чужаки.

3. Обзор работ. Понятие диссонанса и алгоритм HOTSAX, предложенные в работе [12], инициировали исследования, направленные на применение диссонансов для поиска аномалий в различных приложениях временных рядов: ЭКГ [5], энергопотребление [3] и др. Улучшению алгоритма HOTSAX посвящены следующие работы. Алгоритм WAT [9] применяет вейвлеты Хаара вместо трансформации SAX. Алгоритм Hash_DD [20] использует хэш-таблицу вместо префиксного дерева. В алгоритме BitClusterDiscord [15] применяется кластеризация битовых представлений подпоследовательностей.

В работе [22] Янков, Кеог и др. представили алгоритм поиска диссонансов для временного ряда, хранящегося на диске, а не в оперативной памяти (далее для краткости данный алгоритм обозначается как DADD, Disk Aware Discord Discovery). В алгоритме DADD вводится понятие *диапазонного диссонанса (range discord)*, означающее диссонанс, который имеет расстояние по меньшей мере r до своего ближайшего соседа, где r — наперед заданный параметр. Алгоритм DADD состоит из двух фаз – поиск и отсеивание кандидатов в диссонансы. Каждая из этих фаз требует одну операцию полного сканирования данных на диске. Для нахождения значения параметра r алгоритм HOTSAX может быть применен следующим образом. С помощью равномерной выборки получают подпоследовательность исходного временного ряда, допускающую размещение в оперативной памяти. Далее алгоритм HOTSAX находит диссонанс в указанной подпоследовательности, а значение r полагается равным расстоянию диссонанса до его ближайшего соседа.

Исследования, связанные с параллельным поиском диссонансов временного ряда, представлены следующими работами. Алгоритмы PDD (Parallel Discord Discovery) [11] и DDD (Distributed Discord Discovery) [21] используют платформу кластера Spark [24] для параллельного поиска диссонансов. В этих алгоритмах временной ряд разбивается на фрагменты, каждый из которых обрабатывается отдельным узлом Spark-кластера.

Алгоритм PDD использует парадигму мастер–рабочие. На первой фазе алгоритма выполняется вычисление оценки расстояния подпоследовательностей ряда до ближайшего соседа (которая, как и в алгоритме HOTSAX, используется для отбрасывания бесперспективных кандидатов). Узел-мастер получает от узлов-рабочих локальные оценки, выбирает из них наибольшую оценку и рассылает ее рабочим. На второй фазе алгоритм сканирует все подпоследовательности ряда для нахождения диссонанса. При этом подпоследовательности, которые отсутствуют в текущем фрагменте и пересекаются друг с другом, объединяются в пакет и пересылаются на текущий узел.

Алгоритм DDD реализуется как распараллеливание алгоритма DADD, выполняемое следующим образом. Каждый узел выполняет поиск кандидатов в диапазонные диссонансы в своем фрагменте. Далее полученные кандидаты объединяются и направляются на каждый узел, где проходят отсеивание. Объединение отброшенных кандидатов исключается из объединения кандидатов в диапазонные диссонансы, давая результирующее множество диссонансов.

Необходимо отметить, что авторы алгоритмов PDD и DDD в своих работах не рассмотрели предшествующий им алгоритм, представляющий собой параллельную версию алгоритма DADD на базе парадигмы MapReduce [6], который Янков, Кеог и др. описали в своей более поздней работе [23] (далее для краткости этот алгоритм обозначается как MR-DADD). В экспериментах авторы алгоритмов PDD и DDD сравнивают свои разработки с последовательными алгоритмами HOTSAX и DADD и ожидаемо опережают их. Фактически, алгоритм DDD по своей структуре и реализации идеологически близок более раннему алгоритму MR-DADD. Алгоритм MR-DADD, однако, более эффективно реализует фазу отсеивания, используя в ней технику подсчета расстояния с ранним окончанием вычислений, которая заключается в следующем. Пусть S_i и C_j — подпоследовательность фрагмента ряда и кандидат в диссонансы соответственно. Тогда вычисление евклидова расстояния $ED(S_i, C_j) = \sum\limits_{k=1}^{n} \sqrt{(s_{i_k} - c_{j_k})^2}$ следует прекратить на позиции $k = p$ $(1 \leqslant p \leqslant n)$, если $\sum\limits_{k=1}^{p} (s_{i_k} - c_{j_k})^2 \geqslant dist^2_{C_j}$, где $dist_{C_j}$ — текущее расстояние до ближайшего соседа C_j.

Проведенный обзор литературы показывает, что пока отсутствуют исследования, связанные с разра-

боткой параллельного алгоритма поиска диссонансов на платформе многоядерных ускорителей. В то же время такой алгоритм может быть востребован как случае, когда временной ряд может быть размещен в оперативной памяти, так и в случае, когда размер временного ряда допускает только хранение на диске. Первому случаю соответствует широкий спектр практических задач, где необходим поиск диссонансов в рядах из десятков миллионов элементов (например, при мониторинге суточной ЭКГ). Во втором случае указанный алгоритм может использоваться в качестве вспомогательного, обеспечивая эффективное вычисление параметра r для параллельного алгоритма поиска диссонансов, использующего хранение данных на диске, а не в оперативной памяти.

4. Параллельный алгоритм поиска диссонансов *MDD*. Распараллеливание алгоритма HOTSAX для ускорителей архитектур Intel MIC и NVIDIA GPU выполнено с помощью технологий OpenMP [17] и OpenAcc [19] соответственно. Указанные технологии идеологически близки и предполагают параллелизм уровня нитей, внедряемый программистом в исходный текст последовательной программы с помощью директив компилятора (например, для циклов с фиксированным количеством повторений — это директивы `#pragma omp parallel for` и `#pragma acc parallel loop` соответственно).

Распараллеливание основано на следующих основных идеях: раздельное распараллеливание циклов перебора подпоследовательностей, использование квадрата евклидова расстояния и матричное представление данных.

На этапе поиска вычисление расстояния между подпоследовательностями ряда может быть выполнено независимо различными нитями параллельного приложения, запущенного на ускорителе. При этом параллельное выполнение перебора подпоследовательностей множеств $NearDiscords$, $Others$, $Neighbors$ и $Strangers$ следует выполнять *раздельно и различным образом для внешнего и внутреннего циклов* с учетом соотношения количества запущенных нитей и мощности указанных множеств. Кроме того, для ускорения вычислений в формуле (1) *вычисление квадратного корня можно опустить*, так как это не изменит относительного ранжирования подпоследовательностей-кандидатов в диссонансы, поскольку функция ED монотонная и вогнутая.

На этапе подготовки *матричное представление* подпоследовательностей ряда позволит выполнить эффективное распараллеливание вычислений предварительной обработки данных по формулам (2), (3) и (4) с помощью OpenMP или OpenAcc. Для эффективной векторизации вычислений предварительной обработки на ускорителях подпоследовательности ряда необходимо выровнять по ширине векторного регистра ускорителя. Под шириной векторного регистра понимается количество вещественных чисел, которые могут быть загружены в векторный регистр за один прием. Выравнивание данных позволяет избежать эффекта разделения цикла (loop peeling) [4], который существенно снижает эффективность векторизации и заключается в следующем. Если начальный адрес массива не выровнен на ширину векторного регистра, то компилятор разбивает цикл на три части, где первая часть итераций, которые обращаются к памяти с начального адреса до первого выровненного адреса, и третья часть итераций с последнего выровненного адреса до конечного адреса, векторизуются отдельно. Применение технологий OpenMP и OpenAcc также обуславливает замену используемого в последовательном алгоритме префиксного дерева на набор матричных структур данных, реализующих хранение соответствующей информации.

Ниже в разделах 4.1 и 4.2 приводится описание реализации указанных идей.

4.1. Реализация алгоритма. Параллельный поиск диссонанса временного ряда *MDD* (см. алгоритм 2) состоит из двух последовательно выполняемых стадий: нахождение потенциальных диссонансов и их уточнение, — каждая из которых является модификацией алгоритма 1.

На стадии нахождения во внешнем цикле перебор подпоследовательностей осуществляется только по множеству $NearDiscords$ и распараллеливается только внутренний цикл перебора, поскольку количество наиболее редких SAX-кодов перебираемых подпоследовательностей (мощность множества $NearDiscords$) потенциально меньше, чем количество нитей, запущенных на ускорителе. *На стадии уточнения* во внешнем цикле перебор подпоследовательностей осуществляется только по множеству $Others$ и распараллеливается внешний цикл, поскольку количество остальных подпоследовательностей (мощность множества $Others$) потенциально больше, чем количество нитей, запущенных на ускорителе. В обеих стадиях перебор подпоследовательностей распараллеливается с помощью вставки директивы компилятора (для унификации обозначенной `#pragma parallel`). Операторы тела цикла, выполняющего вычисление квадратов евклидовых расстояний, векторизуются компилятором.

4.2. Реализация структур данных. Предлагаемый алгоритм использует следующие основные структуры для хранения данных в оперативной памяти (см. рис. 1).

Исходный временной ряд представляется алгоритмом MDD в виде матрицы выровненных подпоследовательностей для обеспечения эффективной векторизации вычислений. Выравнивание подпоследова-

Алгоритм 2. MDD(IN T, n; OUT $pos_{bsf}, dist_{bsf}$)

1. FIND

```
1:  dist_bsf ← 0; dist_min ← ∞
2:  for C_i ∈ (NearDiscords · Others) do
3:      #pragma parallel
4:      for C_j ∈ (Neighbors(C_i) · Strangers(C_i)) do
5:          d ← ED²(C_i, C_j)
6:          if d < dist_bsf then
7:              break
8:          dist_min ← min(dist, dist_min)
9:      dist_bsf ← max(dist_min, dist_bsf)
10:     if dist_bsf < dist_min then
11:         pos_bsf ← i
12: return {pos_bsf, dist_bsf}
```

2. REFINE

```
1:  dist_bsf ← 0; dist_min ← ∞
2:  #pragma parallel
3:  for C_i ∈ Others do
4:      for C_j ∈ (Neighbors(C_i) · Strangers(C_i)) do
5:          d ← ED²(C_i, C_j)
6:          if d < dist_bsf then
7:              break
8:          dist_min ← min(dist, dist_min)
9:      dist_bsf ← max(dist_min, dist_bsf)
10:     if dist_bsf < dist_min then
11:         pos_bsf ← i
12: return {pos_bsf, √dist_bsf}
```

тельностей выполняется следующим образом. Пусть $width_{VPU}$ — ширина векторного регистра ускори-

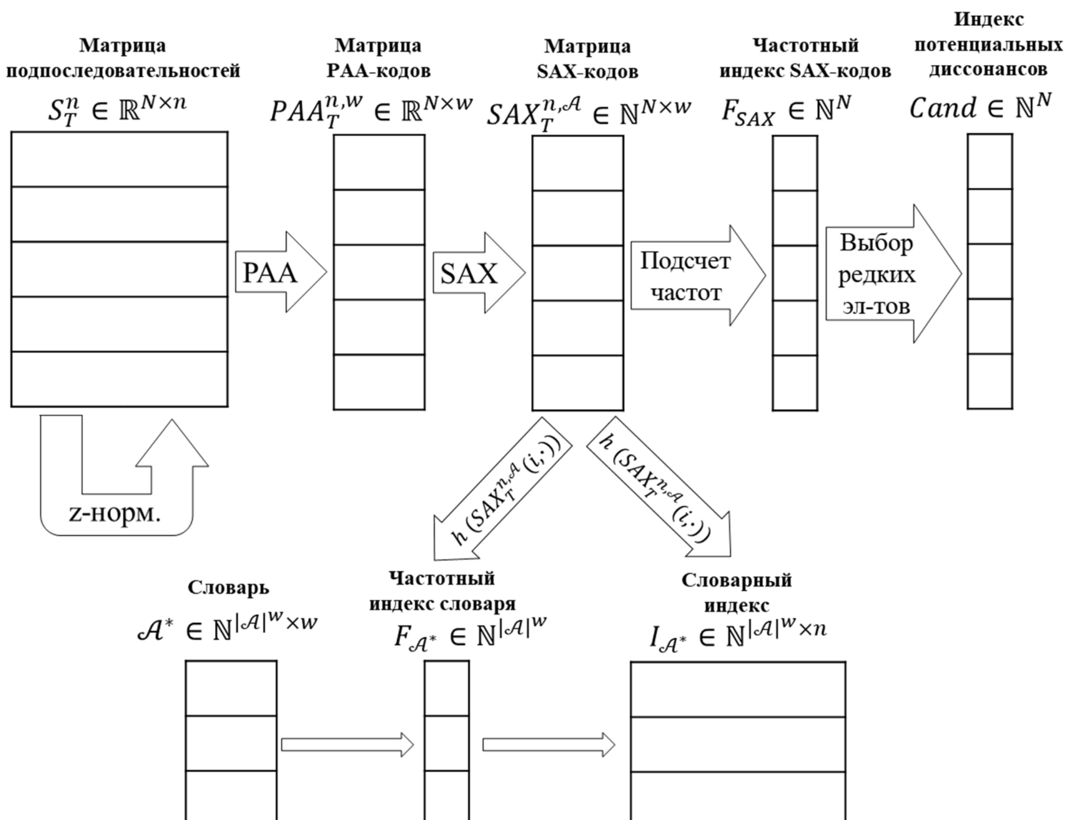


Рис. 1. Структуры данных алгоритма MDD

теля. Если длина искомого диссонанса n не кратна $width_{VPU}$, то каждая подпоследовательность ряда дополняется фиктивными нулевыми элементами. Обозначим количество фиктивных элементов за pad, $pad := width_{VPU} - (n \; mod \; width_{VPU})$, тогда выровненная подпоследовательность $\tilde{T}_{i,n}$ определяется следующим образом:

$$\tilde{T}_{i,n} := \begin{cases} t_i, t_{i+1}, \ldots, t_{i+n-1}, \overbrace{0, 0, \ldots, 0}^{pad} & \text{if } n \; mod \; width_{VPU} > 0 \\ t_i, t_{i+1}, \ldots, t_{i+n-1} & \text{otherwise.} \end{cases} \tag{5}$$

Матрица подпоследовательностей $S_T^n \in \mathbb{R}^{N \times (n+pad)}$ определяется следующим образом:

$$S_T^n(i, j) := \tilde{t}_{i+j-1}. \tag{6}$$

Матрица $PAA_T^{n,\,w} \in \mathbb{R}^{N \times w}$ предназначена для хранения *PAA-кодов*, полученных в соответствии с (3).

Матрица SAX-кодов, $SAX_T^{n,\,\mathcal{A}} \in \mathbb{N}^{N \times w}$, хранит символьные подпоследовательности в алфавите $\mathcal{A}$, полученные из PAA-кодов в соответствии с (4).

Индекс потенциальных диссонансов представляет собой массив $Cand \in \mathbb{N}^N$, упорядоченный по возрастанию, с номерами тех подпоследовательностей в матрице подпоследовательностей S_T^n, чьи SAX-коды наиболее редко встречаются в матрице $SAX_T^{n,\,\mathcal{A}}$:

$$Cand(i) = k \Leftrightarrow F_{SAX}(k) = \min_{1 \leqslant j \leqslant N} F_{SAX}(j) \quad \wedge \quad \forall i < j \quad Cand(i) < Cand(j). \tag{7}$$

Здесь $F_{SAX} \in \mathbb{N}^N$ представляет собой *частотный индекс SAX-кодов* — массив, используемый для хранения частот слов из матрицы SAX-кодов:

$$F_{SAX}(i) = k \Leftrightarrow \left| \left\{ j \;\middle|\; SAX_T^{n,\,\mathcal{A}}(j, \cdot) = SAX_T^{n,\,\mathcal{A}}(i, \cdot) \right\} \right| = k. \tag{8}$$

Индекс потенциальных диссонансов задает порядок перебора подпоследовательностей, которые могут являться диссонансами.

Словарь — матрица $\mathcal{A}^* \in \mathbb{N}^{dict_{\text{size}} \times w}$, предназначенная для хранения всех возможных слов длины w, составленных из символов алфавита $\mathcal{A}$. Словарь заполняется в соответствии с алгоритмом, описанным у Кнута [14], и организуется таким образом, чтобы все символы каждого слова (элементы в одной строке матрицы) и все слова (строки матрицы) были упорядочены по возрастанию. Мощность словаря $dict_{\text{size}}$ вычисляется как число размещений символов алфавита $\mathcal{A}$ по w символов с повторениями:

$$dict_{\text{size}} = \bar{A}^w_{|\mathcal{A}|} = |\mathcal{A}|^w. \tag{9}$$

Символы алфавита $\mathcal{A}$ трактуются как упорядоченный набор натуральных чисел $1, 2, \ldots, |\mathcal{A}|$, и для доступа к элементам словаря вводится хэш-функция $h : \mathbb{N}^w \to \{1, 2, \ldots, dict_{\text{size}}\}$, определяемая следующим образом:

$$h(a_1, a_2, \ldots, a_w) := \sum_{j=1}^{w+1} a_j \cdot w^{w-j-1}. \tag{10}$$

Небольшие значения степени PAA-сжатия и мощности алфавита SAX-кодирования $w = 4$ и $|\mathcal{A}| = 4$ соответственно, как показывают эксперименты [12], хорошо подходят при поиске диссонансов во временных рядах из различных предметных областей. В силу этого словарь может быть размещен в оперативной памяти ($dict_{\text{size}} \times w = 4^4 \times 4 = 256$ элементов).

Словарный индекс предназначен для хранения индексов слов алфавита $\mathcal{A}$ в матрице SAX-кодов и представляет собой матрицу $I_{\mathcal{A}^*} \in \mathbb{N}^{dict_{\text{size}} \times N}$:

$$I_{\mathcal{A}^*}(i, j) = k \Leftrightarrow \mathcal{A}^*(i, \cdot) = SAX_T^{n,\,\mathcal{A}}(k, \cdot). \tag{11}$$

Словарный индекс используется для задания порядка перебора тех подпоследовательностей, которые не пересекаются с данной.

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

$$F_{\mathcal{A}^*}(i) = k \Leftrightarrow k = \left| \left\{ j \;\middle|\; \mathcal{A}^*(i, \cdot) = SAX_T^{n,\,\mathcal{A}}(j, \cdot) \right\} \right|. \tag{12}$$

С учетом предложенных в формулах (5)–(12) вспомогательных структур данных алгоритма можно сформулировать следующее утверждение о пространственной сложности алгоритма MDD. Под пространственной сложностью алгоритма понимается функция от размера входных данных этого алгоритма, равная максимальному объему памяти, используемой алгоритмом для решения экземпляра задачи указанного размера [10].

Утверждение. Пусть имеется временной ряд T ($|T| = m$) и задана длина искомого диссонанса n. Тогда пространственная сложность алгоритма MDD поиска диссонанса равна $O(mn)$.

Доказательство. По построению алгоритм использует следующие вспомогательные структуры данных: матрица подпоследовательностей $S_T^n \in \mathbb{R}^{N\times(n+pad)}$, матрица PAA-кодов $PAA_T^{n,\,w} \in \mathbb{R}^{N\times w}$, матрица SAX-кодов $SAX_T^{n,\,\mathcal{A}} \in \mathbb{N}^{N\times w}$, индекс потенциальных диссонансов $Cand \in \mathbb{N}^N$, частотный индекс SAX-кодов $F_{SAX} \in \mathbb{N}^N$, словарь $\mathcal{A}^* \in \mathbb{N}^{|\mathcal{A}|^w\times w}$, словарный индекс $I_{\mathcal{A}^*} \in \mathbb{N}^{|\mathcal{A}|^w\times N}$ и частотный индекс словаря $F_{\mathcal{A}^*} \in \mathbb{N}^{|\mathcal{A}|^w}$. Суммирование размерностей указанных структур данных дает общий объем используемых алгоритмом данных

$$N\cdot(n+pad+2\cdot(w+1)+|\mathcal{A}|^w)+|\mathcal{A}|^w\cdot(w+1),$$

где $N := m-n+1$. Поскольку $pad < n \ll m$ и используемые на практике значения степени сжатия и мощности алфавита равны $w = 4$ и $|\mathcal{A}| = 4$ [12] соответственно, то, положив в качестве констант $C_1 := pad+2\cdot(w+1)+|\mathcal{A}|^w$ и $C_2 := |\mathcal{A}|^w\cdot(w+1)$, получим величину

$$mn+n(1-n)+C_1(m-n)+C_1+C_2,$$

которая, очевидно, может быть ограничена сверху значением Cmn, где константа $C = 2$, что равносильно тому, что пространственная сложность алгоритма есть $O(mn)$. □

5. Вычислительные эксперименты.

5.1. Цели, аппаратная платформа и наборы данных экспериментов. Для исследования эффективности разработанного алгоритма были проведены вычислительные эксперименты. В качестве аппаратной платформы экспериментов использованы вычислительные мощности Сибирского суперкомпьютерного центра ИВМиМГ СО РАН [2] (см. характеристики в табл. 1).

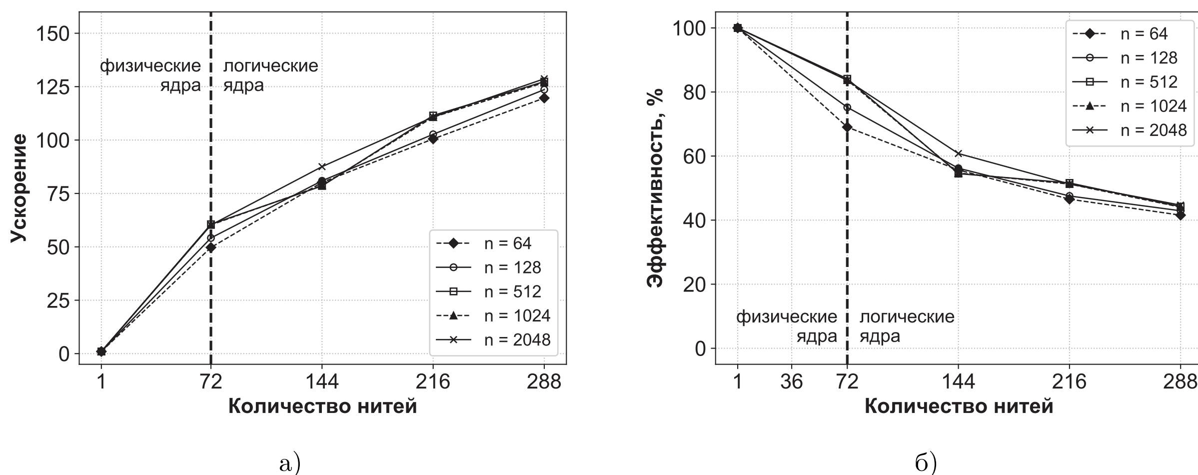
Таблица 1

Аппаратная платформа экспериментов

Характеристика	Ускоритель Intel MIC	Ускоритель NVIDIA
Модель	Xeon Phi 7290	GeForce GTX 1080ti
Кол-во физ. ядер	72	3 584
Гиперпоточность	4×	—
Кол-во лог. ядер	288	—
Частота, ГГц	1.5	1.6
Пиковая произв-ть, TFLOPS	3.456	10.6

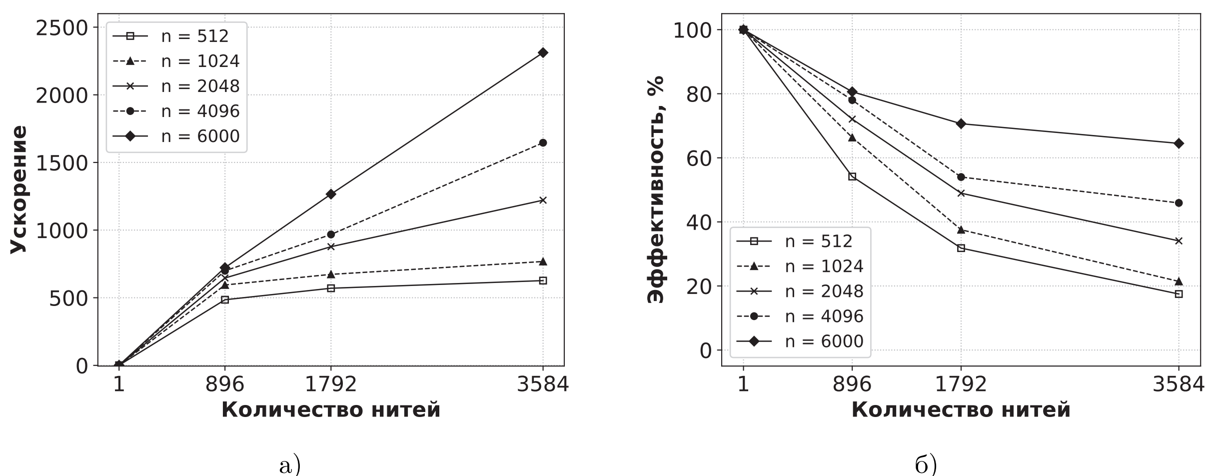
В экспериментах исследовались быстродействие и масштабируемость алгоритма MDD. При исследовании *быстродействия* учитывалось время работы алгоритма за вычетом времени загрузки данных в память и выдачи результата. *Масштабируемость* параллельного алгоритма означает его способность адекватно адаптироваться к увеличению количества параллельно работающих вычислительных элементов (процессов, процессоров, ядер, нитей и др.) и характеризуется ускорением и параллельной эффективностью, которые определяются следующим образом [1].

Ускорение и *параллельная эффективность* параллельного алгоритма, запускаемого на k нитях, вычисляются как $s(k) = \dfrac{t_1}{t_k}$ и $e(k) = \dfrac{s(k)}{k}$ соответственно, где t_1 и t_k — время работы алгоритма на одной и k нитях соответственно. Данные показатели рассматривались в зависимости от изменения параметра n (длина искомого диссонанса). Исследование масштабируемости производилось на синтетическом ряде из 10^7 элементов, который использовался в экспериментах по исследованию эффективности алгоритма PDD [11].



а) б)

Рис. 2. Масштабируемость алгоритма MDD на платформе ускорителя Intel MIC:
а) ускорение, б) параллельная эффективность



а) б)

Рис. 3. Масштабируемость алгоритма MDD на платформе ускорителя NVIDIA GPU:
а) ускорение, б) параллельная эффективность

Быстродействие алгоритма MDD сравнивалось с быстродействием следующих параллельных алгоритмов: PDD [11], DDD [21] и MR-DADD [23], рассмотренных в разделе 3. В экспериментах использовались те же наборы данных, на которых исследовалась эффективность алгоритмов-конкурентов в вышеуказанных работах. Данные о быстродействии алгоритмов-конкурентов взяты из соответствующих статей. Следует отметить, что при этом для алгоритмов DDD и MR- DADD не учитывается время, которое требовалось для вычисления параметра r (см. раздел 3). Запуск алгоритма MDD осуществлялся на ускорителе Intel MIC (см. табл. 1). Для обеспечения справедливых условий сравнения количество вычислительных ядер, на котором запускался алгоритм MDD, уменьшалось таким образом, чтобы обеспечить примерное равенство пиковой производительности текущей аппаратной платформы с аппаратной платформой алгоритма-конкурента, упомянутой в соответствующей статье.

5.2. Результаты экспериментов. Результаты экспериментов *по исследованию масштабируемости* алгоритма на ускорителях архитектур Intel MIC и NVIDIA GPU представлены на рис. 2 и 3 соответственно.

Можно видеть, что алгоритм MDD показывает хорошую масштабируемость на обеих платформах. На платформе Intel MIC ускорение составляет 50–60 раз и параллельная эффективность 70–80% (в зависимости от длины искомого диссонанса), если количество нитей, на которых запущен алгоритм, совпадает

с количеством физических ядер системы. На платформе NVIDIA GPU при запуске одной нити на одном вычислительном ядре на максимальном количестве ядер 3584 лучшие значения ускорения и параллельной эффективности составляют 2450 раз и 65% соответственно. При этом наилучшие показатели ускорения и параллельной эффективности ожидаемо наблюдаются при большем значении параметра n (длина искомого диссонанса), обеспечивающем алгоритму наибольшую вычислительную нагрузку.

Результаты экспериментов *по исследованию быстродействия* алгоритма представлены на рис. 4. Можно видеть, что на платформе NVIDIA GPU алгоритм MDD работает существенно быстрее, чем на ускорителе архитектуры Intel MIC. Данный факт объясняется следующим образом: даже в случае, когда запуск алгоритма осуществляется на количестве ядер GPU, которое дает равную пиковую производительность с ускорителем Intel, графический ускоритель обеспечивает существенно большее (примерно в 4 раза) количество вычислительных ядер.

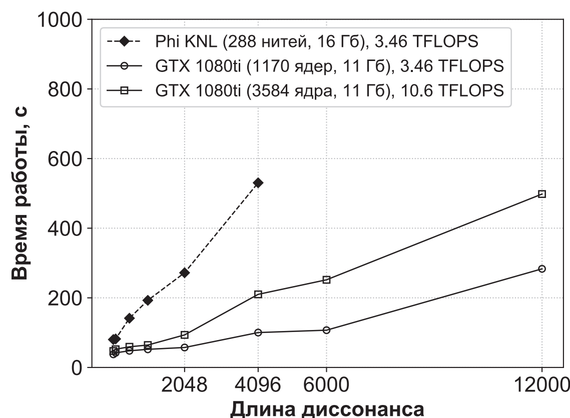


Рис. 4. Быстродействие алгоритма MDD

Результаты экспериментов *по сравнению с аналогами* представлены в табл. 2. Можно видеть, что алгоритм MDD опережает аналоги. Алгоритм PDD значительно проигрывает разработанному алгоритму, поскольку массово использует обмены данными между узлами кластера, что существенно снижает производительность. Алгоритмы DDD и MR-DADD, хотя и не используют массовые обмены, тоже уступают по быстродействию разработанному алгоритму примерно в три и два раза соответственно. В качестве основной причины следует указать накладные расходы на дисковый ввод-вывод, которые обусловлены природой алгоритма, использующего хранение данных на диске, а не в оперативной памяти: как показывают эксперименты [23], такие расходы могут составить более половины общего времени работы алгоритма. Кроме того, выполняемая алгоритмом MDD параллельная обработка матричных структур данных в оперативной памяти эффективно векторизуется компилятором.

Таблица 2

Сравнение алгоритма MDD с аналогами

Условия эксперимента				Быстродействие, с	
Аналог		Длина ряда	К-во ядер (нитей) Intel MIC	MDD	Аналог
Алгоритм	Платформа				
MR-DADD [23]	8 CPU 3.0 ГГц	10^6	8 (32)	101.6	240
DDD [21]	4 CPU 2.13 ГГц	10^7	4 (16)	1 745.3	5 382
PDD [11]	10 CPU 1.2 ГГц	10^7	10 (40)	833.3	399 600

6. Заключение. В настоящей статье рассмотрена проблема поиска диссонанса во временном ряде. Диссонанс является уточнением понятия аномальной подпоследовательности временного ряда. Данная задача возникает в широком спектре приложений интеллектуального анализа временных рядов: моделирование климата, финансовые прогнозы, медицинские исследования и др.

Предложен новый параллельный алгоритм MDD (Many-core Discord Discovery) поиска диссонанса во временном ряде для многоядерных ускорителей архитектур Intel MIC (Many Integrated Core) и NVIDIA GPU для случая, когда входные данные могут быть размещены в оперативной памяти. MDD расширяет последовательный алгоритм HOTSAX, предложенный Кеогом и др. Распараллеливание алгоритма HOTSAX выполняется с помощью технологий OpenMP и OpenAcc соответственно и основано на следующих основных идеях: раздельное распараллеливание циклов перебора подпоследовательностей, использование квадрата евклидова расстояния и матричное представление данных.

Представление исходного ряда в виде матрицы выровненных подпоследовательностей позволяет выполнить эффективное распараллеливание вычислений предварительной обработки данных (z-нормализация, кусочная агрегатная аппроксимация и символьная агрегатная аппроксимация). Определен набор матричных структур данных, которые необходимы параллельному алгоритму для хранения информа-

ции, соответствующей префиксному дереву в последовательном алгоритме. Доказано, что пространственная сложность алгоритма MDD составляет $O(mn)$, где m — длина исходного ряда, n — длина искомого диссонанса.

На этапе поиска с помощью построенных на предыдущем этапе индексных структур осуществляется перебор подпоследовательностей ряда и вычисление между ними евклидовых расстояний, которое выполняется независимо различными нитями параллельного приложения. Для ускорения обработки при подсчете евклидова расстояния вычисление квадратного корня опускается. Подпоследовательности, заведомо не являющиеся диссонансом, отбрасываются без вычисления расстояний до их соседей. Перебор подпоследовательностей осуществляется посредством двух вложенных циклов в порядке, который учитывает тип подпоследовательности (“потенциальные диссонансы” и “остальные”, “соседи” и “чужаки”) и обеспечивает наибольшее количество отброшенных подпоследовательностей, как предложено в алгоритме HOTSAX. В алгоритме MDD, однако, параллельное выполнение перебора в указанных циклах происходит раздельно и различным образом для внешнего и внутреннего циклов и для перебираемых в них подпоследовательностей различного типа.

Представлены результаты вычислительных экспериментов, показывающие хорошую масштабируемость алгоритма MDD и его преимущество над аналогами.

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 17-07-00463), Правительства РФ в соответствии с Постановлением № 211 от 16.03.2013 (соглашение № 02.A03.21.0011) и Министерства образования и науки РФ (государственное задание 2.7905.2017/8.9). В работе использованы ресурсы ЦКП Сибирского суперкомпьютерного центра ИВМиМГ СО РАН.

СПИСОК ЛИТЕРАТУРЫ

1. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
2. Перечень оборудования Центра коллективного пользования Сибирского суперкомпьютерного центра ИВМиМГ СО РАН. http://www.sscc.icmmg.nsc.ru/hardware.html.
3. *Ameen J., Basha R.* Mining time series for identifying unusual sub-sequences with applications // Proc. of the 1st Int. Conf. on Innovative Computing, Information and Control (ICICIC 2006). New York: IEEE Press, 2006. 574–577.
4. *Bacon D.F., Graham S.L., Sharp O.J.* Compiler transformations for high-performance computing // ACM Comput. Surv. 1994. **26**, N 4. 345–420.
5. *Chuah M.C., Fu F.* ECG anomaly detection via time series analysis // Lecture Notes in Computer Science. Vol. 4743. Berlin: Springer, 2007. 123–135.
6. *Dean J., Ghemawat S.* MapReduce: simplified data processing on large clusters // Commun. ACM. 2008. **51**, N 1. 107–113.
7. *Duran A., Klemm M.* The Intel Many Integrated Core architecture // Proc. of the 2012 Int. Conf. on High Performance Computing and Simulation. New York: IEEE Press, 2012. 365–366.
8. *Fredkin E.* Trie memory // Commun. of the ACM. 1960. **3**, N 9. 490–499.
9. *Fu A.W.-C., Leung O.T.-W., Keogh E., Lin J.* Finding time series discords based on Haar transform // Lecture Notes in Computer Science. Vol. 4093. Berlin: Springer, 2006. 31–41.
10. *Goldreich O.* Computational complexity: a conceptual perspective. New York: Cambridge University Press, 2008.
11. *Huang T., Zhu Y., Mao Y., et al.* Parallel discord discovery // Lecture Notes in Computer Science. Vol. 9652. Cham: Springer, 2016. 233–244.
12. *Keogh E., Lin J., Fu A.* HOT SAX: efficiently finding the most unusual time series subsequence // Proc. of the 5th IEEE Int. Conf. on Data Mining. New York: IEEE Press, 2005. 226–233.
13. *Keogh E., Lonardi S., Ratanamahatana C.A.* Towards parameter-free data mining // Proc. of the 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. New York: ACM Press, 2004. 206–215.
14. *Knuth D.E.* The art of computer programming. Vol. 4. Fascicle 3: generating all combinations and partitions. Reading: Addison-Wesley, 2005.
15. *Li G., Bräysy O., Jiang L., et al.* Finding time series discord based on bit representation clustering // Knowl.-Based Syst. 2013. **54**. 243–254.
16. *Lin J., Keogh E., Lonardi S., Chiu B.* A symbolic representation of time series, with implications for streaming algorithms // Proc. of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery. New York: ACM Press, 2004. 2–11.
17. *Mattson T.* Introduction to OpenMP // Proc. of the ACM/IEEE SC2006 Conf. on Supercomputing. New York: ACM Press, 2006. doi 10.1145/1188455.1188673.
18. *Owens J.* GPU architecture overview // Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques. New York: ACM Press, 2007. doi 10.1145/1281500.1281643.
19. *Reyes R., López I., Fumero J.J., de Sande F.* A preliminary evaluation of OpenACC implementations // The Journal of Supercomputing. 2013. **65**, N 3. 1063–1075.

20. *Thuy H.T.T., Anh D.T., Chau V.T.N.* An effective and efficient hash-based algorithm for time series discord discovery // Proc. of the 2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science. New York: IEEE Press, 2016. 85–90.
21. *Wu Y., Zhu Y., Huang T., et al.* Distributed discord discovery: Spark based anomaly detection in time series // Proc. of the 17th IEEE Int. Conf. on High Performance Computing and Communications. New York: IEEE Press, 2015. 154–159.
22. *Yankov D., Keogh E.J., Rebbapragada U.* Disk aware discord discovery: finding unusual time series in terabyte sized datasets // Proc. of the 7th IEEE Int. Conf. on Data Mining. New York: IEEE Press, 2007. 381–390.
23. *Yankov D., Keogh E.J., Rebbapragada U.* Disk aware discord discovery: finding unusual time series in terabyte sized datasets // Knowl. Inf. Syst. 2008. **17**, N 2. 241–262.
24. *Zaharia M., Chowdhury M., Franklin M.J., et al.* Spark: Cluster computing with working sets // Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing. Berkeley: USENIX Association, 2010.

Поступила в редакцию
05.05.2019

A Parallel Discord Discovery Algorithm for Time Series on Many-Core Accelerators

M. L. Zymbler[1]

[1] *South Ural State University, School of Electronic Engineering and Computer Science; prospekt Lenina 76, Chelyabinsk, 454080, Russia; Ph.D., Associate Professor, Head of Department, e-mail: mzym@susu.ru*

Received May 5, 2019

Abstract: Discord is a refinement of the concept of anomalous subsequence of a time series. The discord discovery problem frequently occurs in a wide range of application areas related to time series: medicine, economics, climate modeling, etc. In this paper we propose a new parallel discord discovery algorithm for many-core systems in the case when the input data fit in the main memory. The algorithm exploits the ability to independently calculate the Euclidean distances between the subsequences of the time series. Computations are paralleled using OpenMP and OpenAcc for the Intel MIC (Many Integrated Core) and NVIDIA GPU platforms, respectively. The algorithm consists of two stages, namely precomputations and discovery. At the precomputation stage, we construct the auxiliary matrix data structures to ensure the efficient vectorization of computations on an accelerator. At the discovery stage, the algorithm searches for a discord based on the constructed structures. A number of numerical experiments confirm a high scalability of the proposed algorithm.

Keywords: time series, discord discovery, parallel algorithm, vectorization, OpenMP, OpenAcc, Intel Xeon Phi, NVIDIA GPU.

References

1. V. V. Voevodin and Vl. V. Voevodin, *The Parallel Computing* (BHV-Petersburg, St. Petersburg, 2002) [in Russian].

2. Hardware Specifications of the Siberian Supercomputing Center. http://www.sscc.icmmg.nsc.ru/hardware.html. Cited June 5, 2019.

3. J. Ameen and R. Basha, "Mining Time Series for Identifying Unusual Sub-sequences with Applications," in *Proc. 1st Int. Conf. on Innovative Computing, Information and Control (ICICIC 2006), Beijing, China, August 30–September 1, 2006* (IEEE Press, New York, 2006). 574–577.

4. D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformations for High-Performance Computing," ACM Comput. Surv. **26** (4), 345–420 (1994).

5. M. C. Chuah and F. Fu, "ECG Anomaly Detection via Time Series Analysis," in *Lecture Notes in Computer Science* (Springer, Berlin, 2007), Vol. 4743, pp. 123–135.

6. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Commun. ACM **51** (1), 107–113 (2008).

7. A. Duran and M. Klemm, "The Intel Many Integrated Core Architecture," in *Proc. 2012 Int. Conf. on High Performance Computing and Simulation, Madrid, Spain, July 2–6, 2012* (IEEE Press, New York, 2012), pp. 365–366.

8. E. Fredkin, "Trie Memory," Commun. ACM **3** (9), 490–499 (1960).

9. A.W.-C. Fu, O.T.-W. Leung, E. Keogh, and J. Lin, “Finding Time Series Discords Based on Haar Transform,” in *Lecture Notes in Computer Science* (Springer, Berlin, 2006), Vol. 4093, pp. 31–41.

10. O. Goldreich, *Computational Complexity: A Conceptual Perspective* (Cambridge Univ. Press, New York, 2008).

11. T. Huang, Y. Zhu, Y. Mao, et al., “Parallel Discord Discovery,” in *Lecture Notes in Computer Science* (Springer, Cham, 2016), Vol. 9652, pp. 233–244.

12. E. Keogh, J. Lin, and A. Fu, “HOT SAX: Efficiently Finding the Most Unusual Time Series Subsequence,” in *Proc. 5th IEEE Int. Conf. on Data Mining, Houston, USA, November 27–30, 2005* (IEEE Press, New York, 2005), pp. 226–233.

13. E. Keogh, S. Lonardi, and C. A. Ratanamahatana, “Towards Parameter-Free Data Mining,” in *Proc. 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Seattle, USA, August 22–25, 2004* (ACM Press, New York, 2004), pp. 206–215.

14. D. E. Knuth, *The Art of Computer Programming*, Vol. 4: *Fascicle 3: Generating All Combinations and Partitions* (Addison-Wesley, Reading, 2005).

15. G. Li, O. Bräysy, L. Jiang, et al., “Finding Time Series Discord Based on Bit Representation Clustering,” Knowl. Based Syst. **54**, 243–254 (2013).

16. J. Lin, E. Keogh, S. Lonardi, and B. Chiu, “A Symbolic Representation of Time Series, with Implications for Streaming Algorithms,” in *Proc. 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, San Diego, USA, June 13, 2003* (ACM Press, New York, 2004), pp. 2–11.

17. T. Mattson, “Introduction to OpenMP,” in *Proc. 2006 ACM/IEEE Conf. on Supercomputing, Tampa, USA, November 11–17, 2006* (ACM Press, New York, 2006), doi 10.1145/1188455.1188673

18. J. Owens, “GPU Architecture Overview,” in *Proc. Int. Conf. on Computer Graphics and Interactive Techniques, San Diego, USA, August 5–9, 2007* (ACM Press, New York, 2007), doi 10.1145/1281500.1281643

19. R. Reyes, I. López, J. J. Fumero, and F. de Sande, “A Preliminary Evaluation of OpenACC Implementations,” J. Supercomput. **65** (3), 1063–1075 (2013).

20. H. T. T. Thuy, D. T. Anh, and V. T. N. Chau, “An Effective and Efficient Hash-Based Algorithm for Time Series Discord Discovery,” in *Proc. 3rd National Foundation for Science and Technology Development Conf. on Information and Computer Science, Danang, Vietnam, September 14–16, 2016* (IEEE Press, New York, 2016), pp. 85–90.

21. Y. Wu, Y. Zhu, T. Huang, et al., “Distributed Discord Discovery: Spark Based Anomaly Detection in Time Series,” in *Proc. 17th IEEE Int. Conf. on High Performance Computing and Communications, New York, USA, August 24–26, 2015* (IEEE Press, New York, 2015), pp. 154–159.

22. D. Yankov, E. Keogh, and U. Rebbapragada, “Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Datasets,” in *Proc. 7th IEEE Int. Conf. on Data Mining, Omaha, October 28–31, 2007* (IEEE Press, New York, 2007), pp. 381–390.

23. D. Yankov, E. Keogh, and U. Rebbapragada, “Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Datasets,” Knowl. Inf. Syst. **17** (2), 241–262 (2008).

24. M. Zaharia, M. Chowdhury, M. J. Franklin, et al., “Spark: Cluster Computing with Working sets,” in *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing, Boston, USA, June 22–25, 2010* (USENIX Association, Berkeley, 2010),
`https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf`. Cited June 5, 2019.