A Parallel Discord Discovery Algorithm for a Graphics Processor

Ya. A. Kraeva[a],* and M. L. Zymbler[a],**

[a] *South Ural State University (National Research University), Chelyabinsk, 454080 Russian Federation*

* *e-mail: kraevaya@susu.ru*

** *e-mail: mzym@susu.ru*

Abstract—The detection of anomalous subsequences in a time series is required today in a wide range of computationally intensive applications such as digital industry, the Internet of Things, personal medicine, etc. One of the most attractive approaches to solving this problem is the concept of discord—the subsequence of a time series with the greatest distance to its nearest neighbor—because it requires the analyst to set only one intuitive parameter (subsequence length). The previously proposed DRAG (discord range aware gathering) algorithm for finding discords is exact, but its parallel versions are absent for any hardware architecture. The paper proposes a new approach to parallelizing this algorithm for a graphics processing unit, which is called PD3 (parallel DRAG-based discord discovery). A data preprocessing phase was added to the PD3 to compute mean values and standard deviations of all subsequences of the time series. The results are used further at the candidate selection and refinement phases to calculate the distances between the subsequences of the time series. Each algorithm phase is parallelized separately on the basis of the data parallelism concept and using vector data structures. Experiments show that the PD3 is much ahead of competing parallel algorithms in terms of the average time taken to find one discord.

Keywords: time series, detection of anomalies, discord, parallel algorithm, graphics processing unit (GPU)

DOI: 10.1134/S1054661823020062

INTRODUCTION

At present, subsequence anomaly detection in time series is one of the most pressing research problem of time series mining [1] in demand over a wide range of practical domains, such as predictive maintenance of equipment in digital industry applications [9], smart building management in IoT applications [6, 23], human condition monitoring and proactive disease diagnosis in personal medicine applications [4], etc.

The discord concept [7] formalizes the idea of anomalous subsequence and is currently one of the most promising theoretical developments used to search for anomalies in time series [1]. The discord in a time series is defined as a subsequence which has the largest distance to its nearest neighbor. The nearest neighbor is a subsequence that is non-self-match to the given subsequence and is at a minimum distance away. The discord discovery requires only one intuitive parameter—the subsequence length—and therefore is more attractive for the end user than most other approaches to anomaly detection in time series, which require three to seven parameters that are not always intuitively understandable [8].

The algorithm called HOTSAX (heuristically ordered time series using symbolic aggregate approximation) [7] searches for time series discords in the main memory. However, the HOTSAX uses a symbolic aggregate approximation [10] to index subsequences of a time series and is therefore an approximate algorithm. The later proposed DRAG (discord range aware gathering) algorithm [15] is exact and searches for range discords in a time series stored on disk. The range discord is the one that is at least r away from its nearest neighbor, where r is a parameter.

Discord discovery is a computationally costly task, and so parallel algorithms to search for discords on various hardware platforms are strongly needed. In [20, 21], one of the coauthors of this article proposed an approach to parallelizing the HOTSAX algorithm for Intel many-core processors and NVIDIA GPUs based on OpenMP and OpenACC technologies, respectively. This article continues these studies and proposes a parallel algorithm called PD3 (parallel DRAG-based discord discovery) for range discord detection on a GPU based on the sequential DRAG algorithm [15].

The article is organized as follows. Section 1 contains an overview of related work. Section 2 gives a formal statement of the problem and briefly describes the sequential DRAG algorithm. Section 3 describes the proposed parallel algorithm. Section 4 presents the results of computational experiments to study the performance of the proposed algorithm. The conclusion summarizes the results of the study and outlines the path forward for future research.

Received January 27, 2023; revised January 27, 2023; accepted January 27, 2023

1. OVERVIEW OF RELATED WORK

The concept of time series discord was proposed by Keogh et al. in [7] along with the HOTSAX discord search algorithm. The HOTSAX is an approximate algorithm because it applies Symbolic Aggregate ApproXimation (SAX) [10]. The HOTSAX uses two nested loops to search through all pairs of subsequences, calculating the distance between them, and finds the maximum distance to the nearest neighbor. A prefix tree is used by the HOTSAX to store the subsequences of the time series in memory [5]. Unpromising subsequences are discarded during iteration without calculating distances. An unpromising subsequence is the one with a neighbor closer than the current maximum of distances to all its nearest neighbors. The HOTSAX employs some heuristics to discard more unpromising candidates. To do this, the following four sets of subsequences of the original time series are defined. The set of discord candidates contains the subsequences whose SAX codes are the rarest. The remaining subsequences are considered to be obviously normal. For a given subsequence, the sets of its neighbors and the sets of outsiders contain subsequences with SAX codes that match or differ from its SAX code, respectively. The heuristic dictates to search in the outer loop first through candidates and then through obviously normal subsequences. In the inner loop, neighbors are searched through first and then outsiders.

One of the coauthors of this article proposed in [20, 21] an approach to parallelizing the HOTSAX algorithm for Intel many-core processors and NVIDIA GPUs based on OpenMP and OpenACC technologies, respectively. The parallel algorithm uses the representation of the original time series as a matrix of aligned subsequences, which makes it possible to parallelize efficiently preprocessing calculations (z-normalization and symbolic aggregate approximation of subsequences). A set of matrix data structures is defined, which are necessary for the parallel algorithm to store information corresponding to the prefix tree in the sequential algorithm. The exhaustive search of subsequences in a time series is carried out by two nested loops in the way that was proposed in the original sequential algorithm. The parallel search in these loops occurs separately and in different ways for outer and inner loops and depending on the type of subsequences scanned in these loops. The parallel algorithm is significantly ahead of the sequential algorithm in efficiency, but, being its successor, it remains an approximate algorithm.

Later on, Yankov et al. introduced in [15] an exact DRAG (discord range aware gathering) algorithm for finding range discords in a time series stored on disk. A range discord is at a distance of at least r to its nearest neighbor, where r is a preset parameter. The DRAG algorithm consists of two phases, each of which requires one complete scan of the time series to search for discord candidates and to clean the resulting set of false discords. To select the parameter r, the authors propose to apply the HOTSAX algorithm as follows. Uniform sampling is used to produce the longest possible subsequence of the original time series to fit in the main memory. Then the HOTSAX algorithm finds a discord in the specified subsequence, and the value of the parameter r is assumed to be equal to the distance from the found discord to its nearest neighbor.

Regarding parallelization of the DRAG algorithm, we note the following works. In [16], Yankov et al. proposed an approach to parallelizing the DRAG algorithm on a high-performance computing cluster based on the MapReduce paradigm. The authors of this article developed a parallel version of the DRAG algorithm described in [22] for a high-performance computing cluster with nodes based on Intel many-core processors. Parallel implementations of the DRAG algorithm for a graphics processing unit are absent as far as we know. Nevertheless, we can point out in the overview the following two papers which are devoted to parallel search for discords on a GPU but which do not use the DRAG algorithm as a basis.

Thuy et al. proposed in [13] the algorithm KBF_GPU (brute-force for K-distance discord) that searches on a GPU for a subsequence in a time series such that the sum of distances from it to its K nearest neighbors (where K is a preset parameter) is maximal. The authors call this subsequence “K-distance discord” and apply this concept as a means to solve the “twin freak” problem where the discord does not make it possible to find an anomalous subsequence if it occurs more than once in the time series. The KBF_GPU iterates all time series subsequences through two nested loops, where the inner loop is parallelized and computes the sum of distances. In computational experiments, however, the authors compared their design only with the HOTSAX sequential algorithm, which was expectedly much inferior in performance.

In [18], Zhu et al. presented an exact parallel algorithm of time series discord discovery for a graphics processing unit. The algorithm defines discord based on normalized Euclidean distance that is efficiently computed through Pearson correlation using the technique proposed in [11]. The algorithm uses the following computational patterns in the search for discords, which ensure high performance. According to the first pattern, the algorithm first calculates the minimum distance between the candidate subsequence discord and all other subsequences of the time series which are non-self-match to this candidate and then finds the candidate where the maximum distance among all candidates is achieved. The second computational pattern involves stopping the computations in the above pattern beforehand when the distance between

the candidate and some subsequence is less than the current best (minimum) distance. In this case both the candidate and this subsequence are obviously not discords, and there is no need to compute the distances from the candidate to other non-self-match subsequences. The authors describe as a competing algorithm the parallel SCAMP algorithm [19] for computing a matrix profile on a graphics processor. The matrix profile of a given time series [17] can informally be defined as a time series where the *i*th element is the distance from the *i*th subsequence of the original time series to its non-self-match nearest neighbor. Discords can be found as subsequences of the original time series to which local maxima of the matrix profile correspond. Computational experiments performed by the authors show [18] that the proposed algorithm outperforms its analog in efficiency. However, the computational patterns proposed by the authors limit the search result to only one (albeit most important) discord of the time series, whereas the DRAG algorithm described above ensures finding all range discords.

We can see from this overview that the DRAG algorithm [15] is a promising tool for search for exact discords, for which a parallel version for a graphics process has not been developed yet. Since graphics processors are currently widespread and are one of the most popular manycore platforms [12], we can conclude that the task of developing a parallel version of the DRAG algorithm for discord discovery in time series on a graphics processor is quite urgent.

2. PROBLEM STATEMENT

2.1. Formal Definitions and Notation

This section provides the notation and definitions of terms used according to [7, 15].

The time series T is a sequence of chronologically ordered real values:

$$T = (t_1, \ldots, t_n), \quad t_i \in \mathbb{R}. \tag{1}$$

The number n is denoted by $|T|$ and is called the length of the time series.

The subsequence $T_{i,m}$ of the time series T is a continuous interval of m elements starting from i:

$$T_{i,m} = (t_i, \ldots, t_{i+m-1}), \quad 1 \le m \ll n, \quad 1 \le i \le n - m + 1. \tag{2}$$

We denote the set of all subsequences of the time series T having length m as S_T^m and the cardinality of this set as $N = |S_T^m| = n - m + 1$.

The subsequences $T_{i,m}$ and $T_{j,m}$ of T are called non-self-match if $|i - j| \le m$. We denote the subsequence that is non-self-match to the given subsequence C as M_C.

The subsequence D of the time series T is a discord if

$$\forall C, M_C \in S_T^m \min(\mathrm{Dist}(D, M_D)) > \min(\mathrm{Dist}(C, M_C)), \tag{3}$$

where $\mathrm{Dist}(\cdot, \cdot)$ is a nonnegative symmetric function. In other words, a subsequence of the time series is a discord if it is at the maximum distance to its nearest neighbor, which is the nearest non-self-match subsequence in the sense of the chosen metric.

For a preset parameter r, the discord that is at least r away from its nearest neighbor is called a range discord; i.e., the property $\min(\mathrm{Dist}(D, M_D)) \le r$ is valid for the discord D.

To search for discords, the Euclidean distance (or its modification) is used as the function $\mathrm{Dist}(\cdot, \cdot)$, which is defined as follows. Let there be subsequences X and Y of length m of time series T; then the Euclidean distance ED between X and Y is calculated as

$$\mathrm{ED}(X, Y) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}. \tag{4}$$

The DRAG algorithm assumes that the processed subsequences of the time series were z-normalized is advance. The z-normalization of the subsequence/series T is a subsequence/series $\hat{T} = (\hat{t}_1, \ldots, \hat{t}_m)$, whose elements are calculated as follows:

$$\hat{t}_i = \frac{t_i - \mu}{\sigma}, \quad \mu = \frac{1}{m}\sum_{i=1}^{m} t_i, \quad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m} t_i^2 - \mu^2. \tag{5}$$

The square of normalized Euclidean distance is used in our study as the function $\mathrm{Dist}(\cdot,\cdot)$:

$$\mathrm{Dist}(T_{i,m}, T_{j,m}) = \mathrm{ED}^2_{\mathrm{norm}}(T_{i,m}, T_{j,m}) = \mathrm{ED}^2(\hat{T}_{i,m}, \hat{T}_{j,m}). \tag{6}$$

To calculate normalized Euclidean distance, we use the formula proposed in [11], which makes it possible to perform calculations faster than by formulas (4)–(5):

$$\mathrm{ED}^2_{\mathrm{norm}}(T_{i,m}, T_{j,m}) = 2m\left(1 - \frac{T_{i,m} \cdot T_{j,m} - m\mu_i\mu_j}{m\sigma_i\sigma_j}\right), \tag{7}$$

where the subsequences $T_{i,m}$ and $T_{j,m}$ are considered as vectors in Euclidean space $\mathbb{R}^m$, and μ_i and μ_j, σ_i and σ_j are the arithmetic mean and standard deviation of these vectors, respectively.

2.2. Sequential Algorithm

Alg. 1 DRAG (in T, m, r; out $\mathcal{D}$)

Phase 1. Candidate selection

```
1:  C ← {T_{1,m}}
2:  for all s ∈ S_T^m \ T_{1,m} do
3:      isCand ← TRUE
4:      for all c ∈ C and c ∈ M_s do
5:          if ED(s, c) < r then
6:              C ← C \ c
7:              isCand ← FALSE
8:      if isCand then
9:          C ← C ∪ s
10: return C
```

Phase 2. Candidate refinement

```
1:  D ← ∅
2:  for all c ∈ C do
3:      c.dist ← ∞
4:  for all s ∈ S_T^m do
5:      for all c ∈ C and c ∈ M_s do
6:          if s = c then
7:              continue
8:          d ← EarlyAbandon ED(s, c)
9:          if d ≥ r then
10:             D ← D ∪ c
11:             c.dist ← min(c.dist, d)
12:         else
13:             C ← C \ c
14: return D
```

The DRAG algorithm (see Alg. 1) consists of two phases: selection and refinement, wherein the candidate discord subsequences are selected and false positive candidates are removed, respectively. The algorithm scans the time series T at the first phase and checks the candidate c already in the set of discord candidates $\mathcal{C}$ whether it is a true discord for each subsequence $s \in S_T^m$. If a candidate fails the test, it is removed from the set of candidates. At the end of the first phase, the new element s is either added to the candidate set or discarded. At the second phase, the algorithm initializes the distance to its nearest neighbor with $+\infty$ for each candidate selected at the first phase. The time series T is then scanned and the distance between each subsequence $s \in S_T^m$ and each candidate c is calculated. The calculation of the distance uses the early completion of the summation $\sum_{k=1}^{m}(s_k - c_k)^2$ if the position $k = \ell$ $(\ell < m)$, for which $\sum_{k=1}^{\ell}(s_k - c_k)^2 \geq c.\text{dist}^2$ is reached [16]. If the calculated distance is less than r, the candidate is removed from $\mathcal{C}$. If the distance is less than the current distance from the candidate to its nearest neighbor c.dist (and greater than r because otherwise the candidate would have been previously discarded), the current value of the distance to the nearest neighbor is updated. The formal proof of the DRAG algorithm correctness is given in the original paper [16].

3. PARALLEL DISCORD DISCOVERY ALGORITHM

This section presents a parallel algorithm for discord discovery on a graphics processing unit based on the above-described DRAG algorithm [15] and named PD3 (parallel DRAG-based discord discovery). Unlike the original algorithm, the PD3 uses the square of normalized Euclidean distance as the metric to speed up computation. In addition, we added a preprocessing phase to the PD3 algorithm, which consists in parallel computation of mean values and standard deviations of all subsequences of the time series. The results are then used at the candidate selection and refinement phases to calculate the distances between subsequences of the time series by formula (7). Each of these phases of the algorithm is parallelized separately on the basis of the concept of data parallelism and using vector data structures that we developed.

Subsection 3.1 below briefly describes the architecture and parallel programming model of the GPU. Further, subsections 3.2 and 3.3 contain, respectively, the data structures and implementation principles of the developed parallel algorithm.

3.1. The GPU Hardware and Software Architecture

The NVIDIA graphics processing unit (GPU) [12] is one of the most popular manycore accelerators at the moment. The GPU has a hierarchical architecture and consists of symmetric streaming multiprocessors (SM). Each multiprocessor, in turn, consists of CUDA (compute unified device architecture) cores. Modern GPUs have thousands of CUDA cores capable of outperforming CPUs on tasks that involve massive parallel computing coupled with vector data processing.

A parallel application runs on the GPU as a set of threads where each thread is executed by a separate CUDA core and the following thread hierarchy is provided. The top level of the hierarchy correspond-

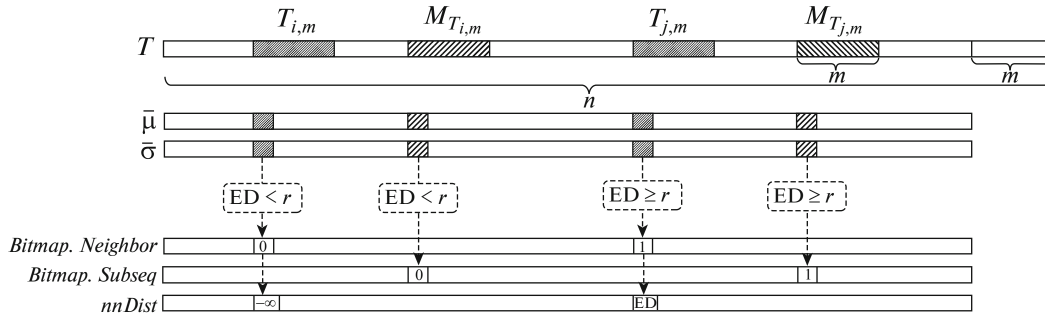

Fig. 1. The PD3 data structures.

ing to all threads is a grid that consists of a one-dimensional or two-dimensional array of symmetric thread blocks. A thread block is a d-dimensional ($1 \leq d \leq 3$) array of threads. The threads are divided inside the block into warps—logical groups of 32 threads each. The warp threads are executed in SIMT (single instruction multiple threads) mode, where each thread executes the same instruction over its own portion of shared data.

The CUDA function is called kernel. When running a kernel on the GPU, the application programmer determines both the number of blocks in the grid and the number of threads in each block. When the application starts, the thread blocks are distributed for execution among thread multiprocessors and are further executed in parallel without the possibility of synchronization. Threads within a block are allowed to be synchronized and have access to shared memory allocated for this block. Global accelerator memory is used to transfer data between threads belonging to different blocks.

3.2. The Data Structures

The following data structures shown in Fig. 1 are used to implement the PD3 algorithm, which include vectors of mean values and standard deviations, anomaly rating, and a bitmap.

The vectors $\bar{\mu}, \bar{\sigma} \in \mathbb{R}^N$ store, respectively, the mean values and values of standard deviations of all subsequences of the original time series: $\bar{\mu}(i) = \mu(T_{i,m})$, $\bar{\sigma}(i) = \sigma(T_{i,m})$.

The anomaly rating is the vector nnDist $\in \mathbb{R}^N$ whose element expresses the distance of the corresponding subsequence of the time series to its nearest neighbor: $nnDist(i) = \min\left(\mathrm{ED}^2_{\mathrm{norm}}\left(T_{i,m}, M_{T_{i,m}}\right)\right)$.

The *Bitmap* is a set of two vectors including the subsequence map and the nearest-neighbor map *Bitmap.Subseq*, *Bitmap.Neighbor* $\in \mathbb{B}^N$. In the vectors, the ith element is TRUE if the subsequence $T_{i,m}$ and its nearest neighbor are discords and FALSE otherwise, respectively. The bitmap is initialized with TRUE values.

3.3. The Algorithm Implementation

3.3.1. Parallel data preprocessing

As compared to the original sequential algorithm, a preprocessing phase is added to the PD3 computational scheme to be followed by the candidate selection and refinement phases. Each phase is parallelized separately. The preprocessing phase involves parallel calculation on the GPU of mean values and of standard deviation values of all subsequences of the time series according to formula (5). The corresponding CUDA kernel forms a one-dimensional grid of N threads in which each *blocksize* of threads is a separate block where the *blocksize* is a parameter of the algorithm and is set multiple to the size of the GPU warp. Each thread computes one element of the vectors $\bar{\mu}$ and $\bar{\sigma}$. The following is the description of methods for parallelizing the candidate selection and refinement phases.

3.3.2. Parallelizing candidate selection

We use the concept of data parallelism illustrated in Fig. 2 for the parallel implementation of the candidate selection phase. The time series is divided into segments $T^{(i)}$ $\left(1 \leq i \leq \left\lceil \frac{N}{segN} \right\rceil\right)$ of equal length, where each segment is processed by a separate thread block. The thread block considers the subsequences of its segment as (local) candidates for discord and it scans and processes the subsequences of the time series, which are non-self-match to the candidates and are located to the right of the segment.

The subsequences are processed as follows. If the distance from the candidate to the subsequence in question is less than the parameter r, the candidate and subsequence are excluded from further processing as certainly not being a discord and the corresponding bitmap flags are set to FALSE. In this case, if all local candidates are discarded, the block terminates ahead of time. The block scans the subsequences of the time series to the right of the local candidates in chunks with the number of elements therein being equal to the length of the segment. The first chunk of these elements starts with the m-th element in the segment. This technique

makes it possible to avoid redundant checks at the intersection of candidates and chunk subsequences.

The segment length is an algorithm parameter and is set multiple to the size of the GPU warp. It is necessary for thread load balancing that the number of processed subsequences of the time series with the length m be multiple to the number of subsequences of this length in the segment. If it is not multiple, the time series is complemented on the right by dummy elements of value $+\infty$. We introduce the following notation to formalize the above scheme for selecting the segment length. We designate the segment length as *seglen*. The cardinality of the set of segment subsequences of length m is denoted *segN*; then $segN = seglen - m + 1$. We designate the number of dummy elements in the rightmost segment of the time series as *pad*; then

$$pad = \begin{cases} m-1, & N \bmod segN = 0 \\ \left\lceil \frac{N}{segN} \right\rceil \cdot segN + 2(m-1) - n, & otherwise. \end{cases} \quad (8)$$

Alg. 2 ParSelectCandidates (in T, m, r, $\bar{\mu}$, $\bar{\sigma}$; out $\mathscr{C}$)

1: **for each segment** $T^{(i)}$ **of** T **do** ▷ **Parallel**
2: **for each chunk** $Chunk^{(j)}$ of $T^{(i)}$ where $i \le j$ do
3: **if** $i = j$ **then**
4: $QTrow \leftarrow$ CalculateDotProducts($Chunk^{(j)}_{1,m}$, $T^{(i)}$)
5: **else**
6: $QTrow \leftarrow$ UpdateDotProducts($Chunk^{(j)}_{1,m}$, $T^{(i)}$)
7: $QTcol \leftarrow$ CalculateDotProducts($T^{(i)}_{1,m}$, $Chunk^{(j)}$)
8: $dist \leftarrow$ CalculateEDnorm($Chunk^{(j)}_{1,m}$, $T^{(i)}$, $QTrow$, $\bar{\mu}$, $\bar{\sigma}$)
9: **if** $dist < r$ **then**
10: $Bitmap.Subseq(i \cdot segN + tid) \leftarrow$ `FALSE`
11: $Bitmap.Neighbor(j \cdot segN + 1) \leftarrow$ `FALSE`
12: **else**
13: $nnDist(i \cdot segN + tid) \leftarrow \min(dist, nnDist(i \cdot segN + tid))$
14: **if** $\bigvee_{k=i \cdot segN}^{(i+1) \cdot segN} Bitmap.Subseq(k) =$ `FALSE` **then**
15: **break**
16: **for each subsequence** $Chunk^{(j)}_{k,m}$ **of** $Chunk^{(j)} \setminus Chunk^{(j)}_{1,m}$ **do**
17: $QTrow \leftarrow$ UpdateDotProducts($QTrow$, $QTcol$, $Chunk^{(j)}_{k,m}$, $T^{(i)}$)
18: $dist \leftarrow$ CalculateEDnorm($Chunk^{(j)}_{k,m}$, $T^{(i)}$, $QTrow$, $\bar{\mu}$, $\bar{\sigma}$)
19: **if** $dist < r$ **then**
20: $Bitmap.Subseq(i \cdot segN + tid) \leftarrow$ `FALSE`
21: $Bitmap.Neighbor(j \cdot segN + k) \leftarrow$ `FALSE`
22: **else**
23: $nnDist(i \cdot segN + tid) \leftarrow \min(dist, nnDist(i \cdot segN + tid))$
24: **if** $\bigvee_{k=i \cdot segN}^{(i+1) \cdot segN} Bitmap.Subseq(k) =$ `FALSE` **then**
25: **break**
26: **for each subsequence** $T_{k,m}$ **of** T **do** ▷ **Parallel**
27: $Bitmap.Subseq(k) \leftarrow Bitmap.Subseq(k) \wedge Bitmap.Neighbor(k)$
28: $\mathscr{C} \leftarrow \{T_{k,m} \mid Bitmap.Subseq(k) =$ `TRUE`$\}$
29: **return** $\mathscr{C}$

Alg. 2 shows the implementation of the parallel candidate selection. The algorithm is implemented as two CUDA kernels called one after another. The first CUDA kernel forms a one-dimensional grid of $\left\lceil \frac{N}{segN} \right\rceil$ blocks, each having *segN* threads. A thread block com-

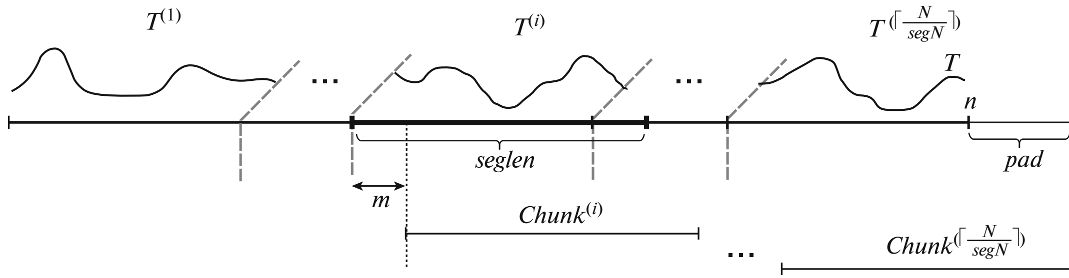

Fig. 2. The series segmentation and processing scheme at the candidate selection phase.

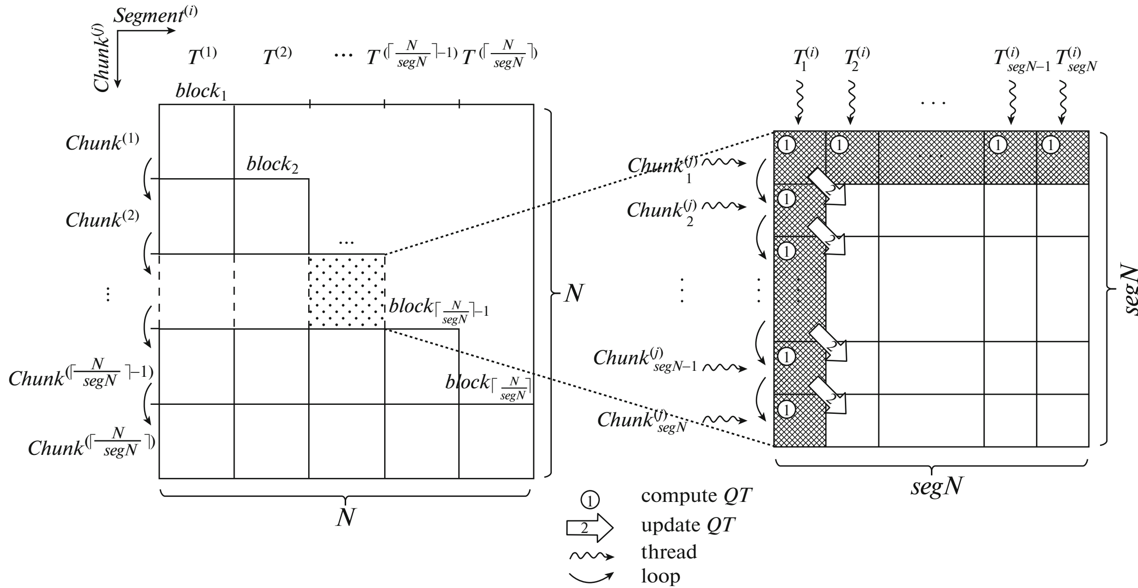

Fig. 3. The thread block operation scheme in the ParSelectCandidates algorithm.

putes the distances from all subsequences of its segment to all subsequences of the current chunk. The operation of the threads inside the block is detailed in Fig. 3.

The thread block loads its segment into shared memory once prior to starting computations and loads the current chunk to the right of the segment into the same memory at each scanning step. This technique reduces the number of accesses to global memory to read elements of the time series, which increases the performance of the algorithm. Then the threads compute scalar products by storing the results in shared memory first between the first subsequence of the current chunk and all subsequences of the segment and then between the first subsequence of the segment and all subsequences of the current chunk (vectors *QTrow* and $QTcol \in \mathbb{R}^{segN}$ in rows 4, 6, and 7 in Alg. 2, respectively).

On the basis of the results (vector *QTrow*) and the previously computed vectors $\bar{\mu}$ and $\bar{\sigma}$, formula (7) is used to calculate the distances between the first subsequence of the chunk and all subsequences of the segment (see row 8 in Alg. 2). The unpromising discord candidates in the segment and in the current chunk are discarded using the calculated distance (see rows 9–11 in Alg. 2). If the candidates are not discarded, the anomaly rating of the candidate in the segment is updated (see row 13 in Alg. 2). If all candidates are discarded as a result of segment processing, the thread block terminates early (see rows 14–15 in Alg. 2).

Thereafter the threads perform similar operations on the remaining subsequences of the current chunk, calculating, however, the scalar products more efficiently (see rows 16–25 in Alg. 2). The computation of scalar products between the current subsequence of the chunk being processed and all subsequences of the segment (vector *QTrow*) is based on *QTcol* computed earlier and *QTrow* computed during the previous iteration (see row 17 in Alg. 2). Each thread processes the segment subsequence whose number coincides with the number of the thread in the block. The thread that computes one scalar product between the kth $(1 < k \le segN)$ subsequence of the $Chunk(j)$ and the segment subsequence $T^{(i)}$ uses the following formula:

$$QTrow(tid) = \begin{cases} QTrow(tid-1) - T^{(i)}_{tid-1,m}(1) \cdot Chunk^{(j)}_{k-1,m}(1) \\ + T^{(i)}_{tid,m}(m) \cdot Chunk^{(j)}_{k,m}(m), \quad 1 < tid \le segN \\ QTcol(k), \quad tid = 1, \end{cases} \tag{9}$$

where *tid* denotes the thread number in the block. Since the value of the first summand is not computed in our proposed formula (9) but is taken from the previous iteration, the complexity of calculating the scalar product is $O(1)$ instead of $O(m)$ for the case of a direct computation.

The second CUDA kernel performs elementwise conjunction of the *Bitmap.Subseq* and *Bitmap.Neighbor* vectors, writing its result into *Bitmap.Subseq*. This operation makes it possible to additionally discard the subsequences that are nearest neighbors of the subsequences discarded during the work of the first CUDA kernel. The kernel is organized as a one-dimensional grid of $\left\lceil \frac{N}{blocksize} \right\rceil$ blocks consisting of *blocksize* threads, where each thread computes one element of the resulting *Bitmap.Subseq* vector (see rows 26–27 in Alg. 2).

The result of the candidate selection phase is the set $\mathscr{C}$ of the subsequences for which the elements in *Bitmap.Subseq* are TRUE (see row 28 in Alg. 2).

3.3.3. Parallelizing Candidate Refinement

Alg. 3 ParRefineCandidates (in T, m, r, $\bar{\mu}$, $\bar{\sigma}$, $\mathscr{C}$; out $\mathscr{D}$)

```
1:  for each segment T^(i) of T do                                   ▷ Parallel
2:      if ⋀_{k=i·segN}^{(i+1)·segN} Bitmap.Subseq(k) = FALSE then
3:          continue
4:      for each chunk Chunk^(j) of T^(i) where i ≥ j do
5:          if i = j then
6:              QTrow ← CalculateDotProducts(Chunk^(j)_{1,m}, T^(i))
7:          else
8:              QTrow ← UpdateDotProducts(Chunk^(j)_{1,m}, T^(i))
9:          QTcol ← CalculateDotProducts(Chunk^(j)_{1,m}, T^(i))
10:         dist ← CalculateEDnorm(Chunk^(j)_{1,m}, T^(i), QTrow, μ̄, σ̄)
11:         if dist < r then
12:             Bitmap.Subseq(i·segN + tid) ← FALSE
13:         else
14:             nnDist(i·segN + tid) ← min(dist, nnDist(i·segN + tid))
15:         if ⋁_{k=i·segN}^{(i+1)·segN} Bitmap_cand(k) = FALSE then
16:             break
17:         for each subsequence Chunk^(j)_{k,m} of Chunk^(j) \ Chunk^(j)_{1,m} do
18:             QTrow ← UpdateDotProducts(QTrow, QTcol, Chunk^(j)_{k,m}, T^(i))
19:             dist ← CalculateEDnorm(Chunk^(j)_{k,m}, T^(i), QTrow, μ̄, σ̄)
20:             if dist < r then
21:                 Bitmap.Subseq(i·segN + tid) ← FALSE
22:             else
23:                 nnDist(i·segN + tid) ← min(dist, nnDist(i·segN + tid))
24:         if ⋁_{k=i·segN}^{(i+1)·segN} Bitmap.Subseq(k) = FALSE then
25:             break
26: 𝒟 ← {T_{k,m} | Bitmap.Subseq(k) = TRUE}
27: return 𝒟
```

The parallel candidate refinement procedure (see Alg. 3) has an ideological similarity to the procedure of parallel candidate selection described above. Only the row segments whose set of local candidates is not empty participate in the refinement (see rows 2–3 in Alg. 3). The refinement of candidates belonging to some segment consists in scanning and processing the row subsequences that do not overlap with the candidates and are located to the left of the given segment (see row 4 in Alg. 3). If the distance from the candidate to the subsequence in question is less than the parameter r, the candidate is obviously not included in the resulting set of discords and is discarded.

4. COMPUTATIONAL EXPERIMENTS

To evaluate the efficiency of the PD3 algorithm, we conducted computational experiments whereby we investigated the algorithm's performance and compared it with competitors. The performance was understood to be the algorithm running time excluding the overhead cost of reading data from disk, data storing time, etc. The PD3 was run ten times during the experiments, and the mean value was used as the final running time. We took as competitors KBF_GPU [13] and the algorithm from Zhu et al. [18] discussed above in the overview of related work (see Section 1). The authors of these algorithms did not provide their source codes, so to ensure a fair comparison in the experiments, the PD3 algorithm was studied under the same experimental conditions as the competing algorithms. It was run on the same hardware platforms and processed the same time series with the same discord length. The relevant hardware and data information was collected from the papers that proposed these algorithms. The time series used by the authors of the competing algorithms were taken from industrial and medical domains and were in a publicly accessible archive [2].

The results of the PD3 algorithm were compared with those published by the authors of competing algorithms in the above papers. In addition, since the competing algorithms limit the search to a single discord of a given length while our algorithm finds all discords for the given length and parameter r, we compared the average time taken by the algorithms to find one discord. The PD3 parameters were set in all experiments as follows. The segment length (see Subsection 3.3.2) was taken as $seglen = 512$. The parameter r was selected under the procedure based on the application of the HOTSAX algorithm described above (see Section 1).

The experiments to compare with KBF_GPU were conducted on the equipment of the Supercomputer Center of South Ural State University (SUSU) [3], including an NVIDIA Tesla V100 GPU (5120 cores @1.3 GHz) with a peak performance of 7 TFLOPS (for double-precision numbers). The experiment results are shown in Fig. 4. It can be seen that the PD3 significantly (up to two orders of magnitude) outperforms KBF_GPU. This was expected because the KBF_GPU algorithm only parallelizes the complete subsequence search, while the PD3 uses subsequence discarding, advanced data structures, etc.

The experiments to compare with the algorithm of Zhu et al. [18] were conducted using the Lomonosov-2 supercomputer installed at Moscow State University [14], including an NVIDIA Tesla P100 GPU (3584 cores @1.19 GHz) with a peak performance of 4 TFLOPS (for double-precision numbers). The results of the experiments are presented in Fig. 5. The diagram shows the average time taken by the algorithms to find one discord for various time series. This parameter coincides with the total running time of the algorithm proposed by Zhu et al. since this algorithm searches for a single (most important) discord. The average time to find one discord by the PD3 algorithm is shown for different values of the parameter K, the number of the desired (most important) discords being 1, 10, 50, and 100. It can be seen that the algorithm of Zhu et al. significantly (by an order of magnitude) outperforms PD3 when searching for a single discord. However, as the parameter K increased, we could see a change in the picture in favor of PD3. Already when searching for 10 discords, PD3 is either behind the competing algorithm by 1.5 to 2 times (see the ECG, ECG2, and Respiration time series) or spends about as much time on average to find one discord as the competing algorithm (see the Space Shuttle time series) or is already 1.8 times ahead (see Power demand time series). When searching for top 50 discords, the PD3 algorithm is already 2 to 5 times ahead of its competitor. When searching for the top 100 discords, this gap increases by 5 to 17 times. Consequently, in applications where it is required to find all possible anomalies of a time series (rather than the most important one), the PD3 algorithm will be more valuable than the algorithm of Zhu et al.

CONCLUSIONS

This paper reviews the problem of finding anomalous subsequences (continuous intervals) of time series, which is currently relevant in a wide range of applications, including the digital industry, Internet of Things, personal medicine, etc. The study uses the concept of discord [7], which is a subsequence of the time series farthest from its nearest neighbor, while the nearest neighbor of this subsequence is a non-self-match at a minimum distance away. The use of discord discovery to find anomalies in a time series is preferable to other approaches because it is based on an intuitive parameter—the length of the subsequence. The HOTSAX algorithm [7] implements the search for time series discords, but is an approximate algorithm because it transforms the subsequences of the time series using symbolic aggregate approximation [10]. The DRAG algorithm [15] performs an exact search

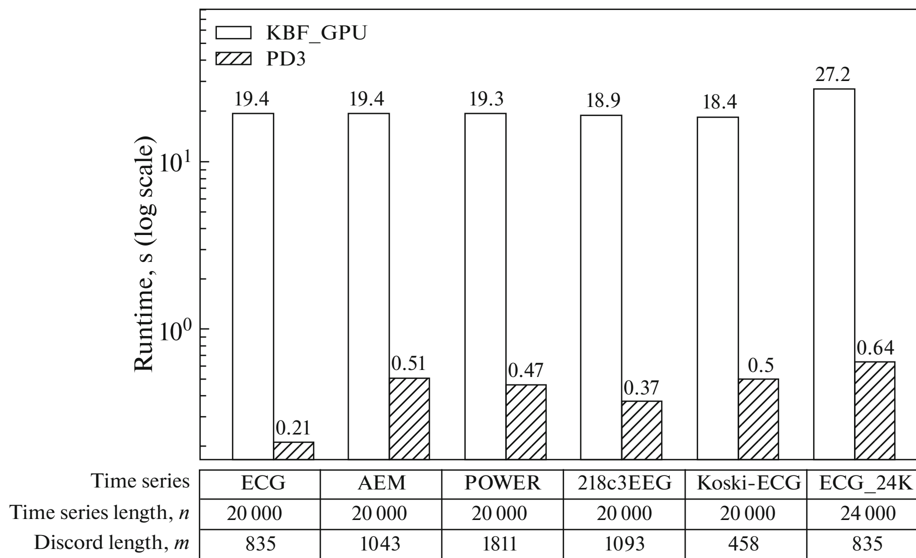

Fig. 4. The performance of the PD3 algorithm as compared to KBF_GPU.

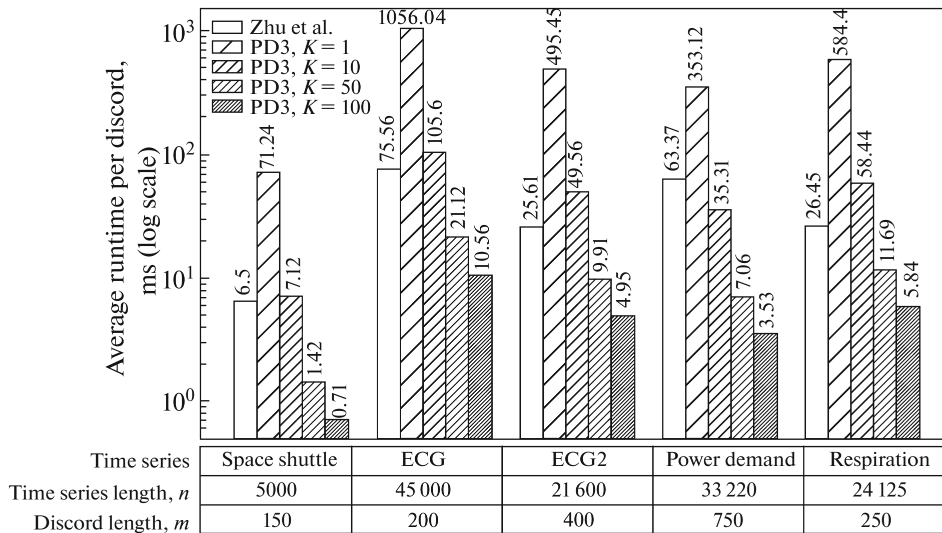

Fig. 5. The performance of the PD3 as compared to the algorithm of Zhu et al.

for time series discords that have a distance of at least r to the nearest neighbor, where r is an expert-set parameter. DRAG consists of two phases: selection of discord candidates and refinement of the resulting set from false discords. A review of the literature shows that no parallelizing schemes for the DRAG algorithm have yet been proposed.

A parallel algorithm named PD3 (parallel DRAG-based discord discovery) is proposed on the basis of the DRAG algorithm to search for range discords of time series on a GPU. To accelerate the calculations, the PD3 uses the square of normalized Euclidean distance as a metric and is supplemented with a preprocessing phase at which the mean values and standard deviations of all subsequences of the time series are computed. The results are then applied at the selection and refinement phases to calculate the distances between the subsequences of the time series. Each phase of the algorithm is parallelized separately on the basis of the concept of data parallelism and using vector data structures. The time series is divided into segments of equal length, each segment being processed by a separate GPU thread block. At the parallel candidate selection phase, the thread block considers the subsequences of its segment as (local) discord candidates and processes the subsequences of the time series which are located to the right of the given segment and do not overlap with the candidates. The parallel

refinement phase uses a similar technique, but it involves only the time series segments whose set of local candidates is not empty, and the subsequences of the time series that are located to the left of the given segment are subject to processing. The processing of subsequences consists in calculating the distance from the candidate to the subsequence in question: if it is less than the parameter r, the candidate and subsequence are excluded from further processing as not being discords. As a result, the subsequences that were not discarded make up the resulting set of the desired discords of the original time series.

The computational experiments involved comparing the performance of the PD3 and of two GPU-based parallel discord discovery algorithms proposed in [13, 18] in so far as the literature review showed the absence of other competitors. Both algorithms limit the search result to a single (most important) discord of the time series. However, the first algorithm, KBF_GPU [13], offers modified discord discovery to solve the problem of "twin freaks" (similar anomalies that cannot be detected using the discord concept) and implements this search by parallelizing complete enumeration of the subsequences of the time series. Expectedly, our algorithm significantly (up to two orders of magnitude) is ahead of KBF_GPU in performance. The second algorithm of Zhu et al. [18] applies computational patterns to discard unpromising discord candidates and thus to improve performance. The PD3 algorithm, while searching for all discords of a given length, lags (by an order of magnitude) behind its counterpart, which searches for only one (most important) of them. However, in terms of the average time taken to search for one discord, the algorithm we have developed is ahead of its competitor (up to one order of magnitude), which can be seen already when searching for the top 50 discords. Thus, the PD3 will be more valuable than the competing algorithm in applications where one needs to find all possible anomalies of the time series (rather than the most important one).

As a possible area for future research, we consider using the PD3 algorithm to develop a method for detecting anomalies in a streaming time series whose values arrive in real time.

ACKNOWLEDGMENTS

The research is carried out using the equipment of the shared research facilities of HPC computing resources at Moscow State University and supercomputer resources of the South Ural State University.

FUNDING

This work was financially supported by the Russian Science Foundation (grant no. 23-21-00465).

CONFLICT OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

1. A. Blázquez-García, A. Conde, U. Mori, and J. A. Lozano, "A review on outlier/anomaly detection in time series data," ACM Comput. Surv. **54**, 56 (2021). https://doi.org/10.1145/3444690
2. H. A. Dau, E. J. Keogh, K. Kamgar, Ch.-Ch. M. Yeh, Ya. Zhu, Sh. Gharghabi, Ch. A. Ratanamahatana, Ya. Chen, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista, and Hexagon-ML, The UCR Time Series Classification Archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/. Cited December 15, 2021.
3. N. Dolganina, E. Ivanova, R. Bilenko, and A. Rekachinsky, "HPC resources of South Ural State University," in *Parallel Computational Technologies. PCT 2022*, Ed. by L. Sokolinsky and M. Zymbler, Communications in Computer and Information Science, Vol. 1618 (Springer, Cham, 2022), pp. 43–55. https://doi.org/10.1007/978-3-031-11623-0_4
4. V. V. Epishev, A. P. Isaev, R. M. Miniakhmetov, A. V. Movchan, A. S. Smirnov, L. B. Sokolinsky, M. L. Zymbler, and V. V. Ehrlich, "Physiological data mining system for elite sports," Vestn. Yuzhno-Ural. Gos. Univ. Ser. Vychislit. Mat. Inf. **2** (1), 44–54 (2013). https://doi.org/10.14529/cmse130105
5. E. Fredkin, Trie memory, Commun. ACM **3**, 490–499 (1960). https://doi.org/10.1145/367390.367400
6. S. Ivanov, K. Nikolskaya, G. Radchenko, L. Sokolinsky, and M. Zymbler, "Digital twin of city: Concept overview," in *2020 Global Smart Industry Conf. (GloSIC), Chelyabinsk, Russia, 2020* (IEEE, 2020), pp. 178–186. https://doi.org/10.1109/GloSIC50886.2020.9267879
7. E. Keogh, J. Lin, and A. Fu, "HOT SAX: Efficiently finding the most unusual time series subsequence," in *Fifth IEEE Int. Conf. on Data Mining (ICDM'05), Houston, Texas, 2005* (IEEE, 2005), pp. 226–233. https://doi.org/10.1109/ICDM.2005.79
8. E. Keogh, S. Lonardi, and Ch. Ratanamahatana, "Towards parameter-free data mining," in *10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Seattle, 2004* (Association for Computing Machinery, New York, 2004), pp. 206–215. https://doi.org/10.1145/1014052.1014077
9. S. Kumar, P. Tiwari, and M. Zymbler, "Internet of Things is a revolutionary approach for future technology enhancement: A review," J. Big Data **6**, 111 (2019). https://doi.org/10.1186/s40537-019-0268-2
10. J. Lin, E. Keogh, S. Lonardi, and B. Chiu, "A symbolic representation of time series, with implications for streaming algorithms," in *8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, San Diego, Calif., 2003* (Association for Computing Machinery, New York, 2003), pp. 2–11. https://doi.org/10.1145/882082.882086
11. A. Mueen, S. Nath, and J. Liu, "Fast approximate correlation for massive time-series data," in *Proc. ACM*

SIGMOD Int. Conf. on Management of Data, Indianapolis, 2010 (Association for Computing Machinery, New York, 2010), pp. 171–182.
https://doi.org/10.1145/1807167.1807188

12. J. Owens, "GPU architecture overview," in *Proc. Int. Conf. on Computer Graphics and Interactive Techniques, SIGGRAPH '07, San Diego, Calif., 2017* (Association for Computing Machinery, New York, 2017), p. 2.
https://doi.org/10.1145/1281500.1281643

13. T. T. T. Huynh, A. T. Duong, and Ch. T. N. Vo, "A new discord definition and an efficient time series discord detection method using GPUs," in *3rd Int. Conf. on Software Engineering and Development (ICSED), Xiamen, China, 2021* (Association for Computing Machinery, New York, 2021), pp. 63–70.
https://doi.org/10.1145/3507473.3507483

14. V. V. Voevodin, A. S. Antonov, D. A. Nikitenko, P. A. Shvets, S. I. Sobolev, I. Yu. Sidorov, K. S. Stefanov, V. V. Voevodin, and S. A. Zhumatiy, "Supercomputer Lomonosov-2: Large scale, deep monitoring and fine analytics for the user community," Supercomput. Front. Innovations **6** (2), 4–11 (2019).
https://doi.org/10.14529/jsfi190201

15. D. Yankov, E. Keogh, and U. Rebbapragada, "Disk aware discord discovery: Finding unusual time series in terabyte sized datasets," in *Seventh IEEE Int. Conf. on Data Mining (ICDM 2007), Omaha, Neb., 2007* (IEEE, 2007), pp. 381–390.
https://doi.org/10.1109/ICDM.2007.61

16. D. Yankov, E. Keogh, and U. Rebbapragada, "Disk aware discord discovery: Finding unusual time series in terabyte sized datasets," Knowl. Inf. Syst. **17**, 241–262 (2008).
https://doi.org/10.1007/s10115-008-0131-9

17. Ch.-Ch. M. Yeh, Ya. Zhu, L. Ulanova, N. Begum, Yi. Ding, H. A. Dau, Z. Zimmerman, D. F. Silva, A. Mueen, and E. Keogh, "Time series joins, motifs, discords and shapelets: A unifying view that exploits the matrix profile," Data Min. Knowl. Discovery **32**, 83–123 (2018).
https://doi.org/10.1007/s10618-017-0519-9

18. B. Zhu, Yo. Jiang, M. Gu, and Ya. Deng, "A GPU acceleration framework for motif and discord based pattern mining," IEEE Trans. Parallel Distrib. Syst. **32**, 1987–2004 (2021).
https://doi.org/10.1109/TPDS.2021.3055765

19. Z. Zimmerman, K. Kamgar, N. Sh. Senobari, B. Crites, G. Funning, Ph. Brisk, and E. Keogh, "Matrix profile XIV: Scaling time series motif discovery with GPUs to break a quintillion pairwise comparisons a day and beyond, in *Proc. ACM Symp. on Cloud Computing, Santa Cruz, Calif., 2019* (Association for Computing Machinery, New York, 2019), pp. 74–86.
https://doi.org/10.1145/3357223.3362721

20. M. L. Zymbler, "A parallel discord discovery algorithm for time series on many-core accelerators," Vychislit. Metody Programm. **20**, 211–223 (2019).
https://doi.org/10.26089/NumMet.v20r320

21. M. Zymbler, A. Polyakov, and M. Kipnis, "Time series discord discovery on Intel many-core systems," in *Parallel Computational Technologies. PCT 2019*, Ed. by L. Sokolinsky and M. Zymbler, Communications in Computer and Information Science, Vol. 1063 (Springer, 2019), pp. 168–182.
https://doi.org/10.1007/978-3-030-28163-2_12

22. M. Zymbler, A. Grents, Ya. Kraeva, and S. Kumar, "A parallel approach to discords discovery in massive time series data," Comput., Mater, Continua **66**, 1867–1876 (2021).
https://doi.org/10.32604/cmc.2020.014232

23. M. Zymbler, Ya. Kraeva, E. Latypova, S. Kumar, D. Shnayder, and A. Basalaev, "Cleaning sensor data in smart heating control system," in *2020 Global Smart Industry Conf. (GloSIC), Chelyabinsk, Russia, 2020* (IEEE, 2020), pp. 375–381.
https://doi.org/10.1109/GloSIC50886.2020.9267813

Translated by D. Sventsitsky

Ya.A. Kraeva, Master of Computer Science. Head of Department at the Research and Educational Center for Artificial Intelligence and Quantum Technologies, Senior Lecturer at the Department of Systems Programming of the School of Electronic Engineering and Computer Science at South Ural State University (Chelyabinsk, Russian Federation). Scientific interests: machine learning, artificial neural networks, time series mining, parallel algorithms. Holder of the Scholarship of the President of the Russian Federation for research.

M.L. Zymbler, Dr. Sci. (Phys.–Math.). Deputy Director of the Research and Educational Center for Artificial Intelligence and Quantum Technologies, Professor at the Department of Systems Programming of the School of Electronic Engineering and Computer Science at South Ural State University (Chelyabinsk, Russian Federation). Scientific interests: machine learning, time series mining, parallel algorithms, parallel database management systems. Holder of grants from the Russian Science Foundation and Russian Foundation for Basic Research.