Computers, Materials & Continua
DOI:10.32604/cmc.2020.014232
Article

Tech Science Press

A Parallel Approach to Discords Discovery in Massive Time Series Data

Mikhail Zymbler*, Alexander Grents, Yana Kraeva and Sachin Kumar

Department of Computer Science, South Ural State University, Chelyabinsk, 454080, Russian
*Corresponding Author: Mikhail Zymbler. Email: mzym@susu.ru
Received: 08 September 2020; Accepted: 30 September 2020

Abstract: A discord is a refinement of the concept of an anomalous subsequence of a time series. Being one of the topical issues of time series mining, discords discovery is applied in a wide range of real-world areas (medicine, astronomy, economics, climate modeling, predictive maintenance, energy consumption, etc.). In this article, we propose a novel parallel algorithm for discords discovery on high-performance cluster with nodes based on many-core accelerators in the case when time series cannot fit in the main memory. We assumed that the time series is partitioned across the cluster nodes and achieved parallelization among the cluster nodes as well as within a single node. Within a cluster node, the algorithm employs a set of matrix data structures to store and index the subsequences of a time series, and to provide an efficient vectorization of computations on the accelerator. At each node, the algorithm processes its own partition and performs in two phases, namely candidate selection and discord refinement, with each phase requiring one linear scan through the partition. Then the local discords found are combined into the global candidate set and transmitted to each cluster node. Next, a node performs refinement of the global candidate set over its own partition resulting in the local true discord set. Finally, the global true discords set is constructed as intersection of the local true discord sets. The experimental evaluation on the real computer cluster with real and synthetic time series shows a high scalability of the proposed algorithm.

Keywords: Time series; discords discovery; computer cluster; many-core accelerator; vectorization

1 Introduction

Currently, the discovery of anomalous subsequences in a very long time series is a topical issue in a wide spectrum of real-world applications, namely medicine, astronomy, economics, climate modeling, predictive maintenance, energy consumption, and others. For such applications, it is hard to deal with multi-terabyte time series, which cannot fit into the main memory.

Keogh et al. [1] introduced HOTSAX, the anomaly detection algorithm based on the discord concept. A *discord* of a time series can informally be defined as a subsequence that has the largest distance to its nearest non-self match neighbor subsequence. A discord looks attractive as an anomaly detector because it only requires one intuitive parameter (the subsequence length), as opposed to most anomaly detection

algorithms, which typically require many parameters [2]. HOTSAX, however, assumes that time series reside in main memory.

Further, Yankov, Keogh *et al*. proposed a disk-aware algorithm (for brevity, referred to as DADD, Disk-Aware Discord Discovery) based on the *range discord* concept [3]. For a given range r, DADD finds all discords at a distance of at least r from their nearest neighbor. The algorithm requires two linear scans through the time series on a disk. Later, Yankov, Keogh et al. [4] discussed parallelization of DADD based on the MapReduce paradigm. However, in the experimental evaluation, the authors just simulated the above-mentioned parallel implementation on up to eight computers.

Our research is devoted to parallel and distributed algorithms for time series mining. In the previous work [5], we parallelized HOTSAX for many-core accelerators. This article continues our study and contributes as follows. We present a parallel implementation of DADD on the high-performance cluster with the nodes based on many-core accelerators. The original algorithm is extended by a set of index structures to provide an efficient vectorization of computations on each cluster node. We carried out the experiments on the real computer cluster with the real and synthetic time series, which showed a high scalability of our approach.

The rest of the article is organized as follows. In Section 2, we give the formal definitions along with a brief description of DADD. Section 3 provides the brief state of the art literature review. Section 4 presents the proposed methodology. In Section 5, the results of the experimental evaluation of our approach have been provided. Finally, Section 6 summarizes the results obtained and suggests directions for a further research.

2 Problem Statement and the Serial Algorithm

2.1 Notations and Definitions

Below, we follow [4] to give some formal definitions and the statement of the problem.

A *time series* T is a sequence of real-valued elements: $T = (t_1, \ldots, t_m)$, $t_i \in \mathbb{R}$. The length of a time series is denoted by $|T|$.

A *subsequence* $T_{i,n}$ of a time series T is its contiguous subset of n elements that starts at position i: $T_{i,n} = (t_i, \ldots, t_{i+n-1})$, $1 \leq n \ll m$, $1 \leq i \leq m - n + 1$. We denote the set of all subsequences of length n in T by S_T^n. Let N denotes the number of subsequences in S_T^n, i.e., $N = \left|S_T^n\right| = m - n + 1$.

A *distance function* for any two subsequences is a nonnegative and symmetric function $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$.

Two subsequences $T_{i,n}, T_{j,n} \in S_T^n$ are *non-trivial matches* [6] with respect to a distance function Dist, if $\exists T_{p,n} \in S_T^n$, $i < p < j$, and $\mathrm{Dist}\left(T_{i,n}, T_{j,n}\right) < \mathrm{Dist}\left(T_{i,n}, T_{p,n}\right)$. Let us denote a non-trivial match of a subsequence $C \in S_T^n$ by M_C.

A subsequence $D \in S_T^n$ is said to be the *most significant discord* in T if the distance to its nearest non-trivial match is the largest. That is,

$$\forall C \in S_T^n \ \min(\mathrm{Dist}(D, M_D)) > \min(\mathrm{Dist}(C, M_C)). \tag{1}$$

A subsequence $D \in S_T^n$ is called the *most significant k-th discord* in T if the distance to its k-th nearest non-trivial match is the largest.

Given a positive parameter r, the discord at a distance at least r from its nearest neighbor is called the *range discord*, i.e., for discord D $\min(\mathrm{Dist}(D, M_D)) \geq r$.

DADD, the original serial disk-based algorithm [4] addresses discovering range discords, and provides researchers with a procedure to choose the r parameter. To accelerate the above-mentioned procedure, our

parallel algorithm [5] for many-core accelerators can be applied, which discovers discords for the case when time series fit in the main memory.

When computing distance between subsequences, DADD demands that the arguments have been previously z-normalized to have mean zero and a standard deviation of one. Here, z-normalization of a subsequence $C \in S_T^n$ is defined as a subsequence $\hat{C} = (\hat{c}_1, \ldots, \hat{c}_n)$ in which

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

In the original DADD algorithm, the Euclidean distance is used as a distance measure yet the algorithm can be utilized with any distance function, which may not necessarily be a metric [4]. Given two subsequences $X, \ Y \in S_T^n$, the Euclidean distance between them is calculated as

$$\mathrm{ED}(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{3}$$

2.2 The Original Algorithm

The DADD algorithm [4] performs in two phases, namely the candidate selection and discord refinement, with each phase requiring one linear scan through the time series on disk. Algorithm 1 depicts a pseudo code of DADD (up to the replacement of the Euclidean distance by an arbitrary distance function). The algorithm takes time series T and range r as an input and outputs set of discords $\mathcal{C}$. For a discord $c \in \mathcal{C}$, we denote the distance to its nearest neighbor as $c.dist$.

Algorithm 1: Disk Aware Discord Discovery (in T, r; out $\mathcal{C}$)

Phase 1. Candidate selection

```
1:  C ← {T_{1,n}}
2:  for all s ∈ S_T^n \ T_{1,n} 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. Discord refinement

```
1:  for all c ∈ C do
2:    c.dist ← ∞
3:  for all s ∈ S_T^n do
4:    for all c ∈ C and c ∈ M_s do
5:      if s = c then
6:        continue
7:      d ← EarlyAbandonED(s, c)
8:      if d < r then
9:        C ← C \ c
10:     else
11:       c.dist ← min(c.dist, d)
12: return C
```

At the first phase, the algorithm scans through the time series T, and for each subsequence $s \in S_T^n$ it validates the possibility for each candidate c already in the set $\mathcal{C}$ to be discord. If a candidate c fails the validation, then it is removed from this set. In the end, the new s is either added to the candidates set, if it is likely to be a discord, or it is discarded. The correctness of this procedure is proved in Yankov et al. [4].

At the second phase, the algorithm initially sets distances of all candidates to their nearest neighbors to infinity. Then, the algorithm scans through the time series T, calculating the distance between each subsequence $s \in S_T^n$ and each candidate c. Here, when calculating $\mathrm{ED}(s, c)$, the EarlyAbandonED procedure stops the summation of $\sum_{k=1}^{n} (s_k - c_k)^2$ if it reaches $k = \ell$, such that $1 \leq \ell \leq n$ for which

$\sum_{k=1}^{\ell}(s_k - c_k)^2 \geq c.dist^2$. If the distance is less than r then the candidate is false positive and permanently removed from $\mathcal{C}$. If the above-mentioned distance is less than the current value of $c.dist$ (and still greater than r, otherwise it would have been removed) then the current distance to the nearest neighbor is updated.

3 Related Work

Being introduced in Keogh et al. [1], currently, time-series discords are considered one of the best techniques for the time series anomaly detection [7].

The original HOTSAX algorithm [1] is based on the SAX (Symbolic Aggregate ApproXimation) transformation [8]. Among the improvements of HOTSAX, we can mention the following algorithms, namely *i*SAX [9] and HOT-*i*SAX [10] (indexable SAX), WAT [11] (Haar wavelets instead of SAX), HashDD [12] (use of a hash table instead of the prefix trie), HDD-MBR [13] (application of R-trees), and BitClusterDiscord [14] (clustering of the bit representation of subsequences). However, the above-mentioned algorithms are able to discover discords if the time series fits in the main memory, and have no parallel implementations, to the best of our knowledge.

Further, Yankov, Keogh et al. [3] overcame the main memory size limitation having proposed a disk-aware discord discovery algorithm (DADD) based on the *range discord* concept. For a given range r, DADD finds all discords at a distance of at least r from their nearest neighbor. The algorithm performs in two phases, namely the candidate selection and discord refinement, with each phase requiring one linear scan through the time series on the disk.

There are a couple of worth-noting works devoted to parallelization of DADD. The DDD (Distributed Discord Discovery) algorithm [15] parallelizes DADD through a Spark cluster [16] and HDFS (Hadoop Distributed File System) [17]. DDD distributes time series onto the HDFS cluster and handles each partition in a memory of a computing node. As opposed to DADD, DDD computes the distance without taking advantage of an upper bound for early abandoning, which would increase the algorithm's performance.

The PDD (Parallel Discord Discovery) algorithm [18] also utilizes a Spark cluster but employs transmission of a subsequence and its non-trivial matches to one or more computing nodes to calculate the distance between them. A bulk of continuous subsequences is transmitted and calculated in a batch mode to reduce the message passing overhead. PDD is not scalable since intensive message passing between the cluster nodes leads to a significant degradation of the algorithm's performance as the number of nodes increases.

In their further work [4], Yankov, Keogh *et al*. discussed the parallel version of DADD based on the MapReduce paradigm (hereinafter referred to as MR-DADD), and the basic idea is as follows. Let the input time series T be partitioned evenly across P cluster nodes. Each node performs the selection phase on its own partition with the same r parameter and produces distinct candidate set $\mathcal{C}^i$. Then the combined candidate set $\mathcal{C}_P$ is constructed as $\mathcal{C}_P = \cup_{i=1}^{P}\mathcal{C}^i$ and transmitted to each cluster node. Next, a node performs the refinement phase on its own partition taking $\mathcal{C}_P$ as an input, and produces the refined candidate set $\mathcal{C}^i$. The final discords are given by the set $\mathcal{C}_P = \cap_{i=1}^{P}\mathcal{C}^i$. In the experimental evaluation, the authors, however, just simulated the above-mentioned scheme on up to eight computers resulting in a near-to-linear speedup.

Concluding this brief review, we should also mention the matrix profile (MP) concept proposed by Keogh et al. [19]. MP is a data structure that annotates a time series, and can be applied to solve an impressively large list of time series mining problems including discords discovery but at computational cost of $O(m^2)$ where m is the time series length [19,20]. Recent parallel algorithms of the MP computation include GPU-STAMP [19] and MP-HPC [21], which are implementations for graphic

processors through CUDA (Compute Unified Device Architecture) technology and computer cluster through MPI (Message Passing Interface) technology, respectively.

4 Discords Discovery on Computer Cluster with Many-Core Accelerators

The parallelization employs a two-level parallelism, namely across cluster nodes and among threads of a single node. We implemented these levels through partitioning of an input time series and MPI technology, and OpenMP technology, respectively. Within a single node, we employed the matrix representation of data to effectively parallelize computations through OpenMP. Below, we will show an approach to the implementation of these ideas.

4.1 Time Series Representation

To provide parallelism at the level of the cluster nodes, we perform time series partitioning across the nodes as follows. Let P be the number of nodes in the cluster, then k-th partition ($0 \leq k \leq P-1$) of the time series is defined as $T_{start,\, len}$ where

$$start = k \cdot \left\lfloor \frac{N}{P} \right\rfloor + 1;\ len = \begin{cases} \left\lfloor \frac{N}{P} \right\rfloor + n - 1 + (N \bmod P), & k = P-1 \\ \left\lfloor \frac{N}{P} \right\rfloor + n - 1, & otherwise. \end{cases} . \tag{4}$$

This means the head part of every partition except the first overlaps with the tail part of the previous partition in $n-1$ data points. Such a technique prevents us from a loss of the resulting subsequences in the junctions of two neighbor partitions. To simplify the presentation of the algorithm, hereinafter in this section, we use symbol T and the above-mentioned related notions implying a partition on the current node but not the whole input time series.

The time series partition is stored as a matrix of aligned subsequences to enable computations over aligned data with as many auto-vectorizable loops as possible. We avoid the unaligned memory access since it can cause an inefficient vectorization due to time overhead for the loop peeling [22].

Let us denote the number of floats stored in the VPU (vector processing unit of the many-core accelerator) by $width_{VPU}$. If the discord length n is not a multiple of $width_{VPU}$, then each subsequence is padded with zeroes where the number of zeroes is calculated as $pad = width_{VPU} - (n \bmod width_{VPU})$. Thus, the aligned (and previously z-normalized) subsequence $\tilde{T}_{i,n}$ is defined as follows:

$$\tilde{T}_{i,n} = \begin{cases} (\tilde{T}_{i,n}, \overbrace{0, \ldots, 0}^{pad}), & if\ n \bmod width_{VPU} > 0 \\ \tilde{T}_{i,n}, & otherwise. \end{cases} . \tag{5}$$

The *subsequence matrix* $S_T^n \in \mathbb{R}^{N \times (n+pad)}$ is defined as

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

4.2 Internal Data Layout

The parallel algorithm employs the data structures depicted in Fig. 1. Defining structures to store data in the main memory of a cluster node, we suppose that each structure is shared by all threads the algorithm is running on, and each thread processes its own data segment independently. Let us denote the amount of threads employing by the algorithm on a cluster node by p, and let iam ($0 \leq iam \leq p-1$) denotes the number of the current thread.

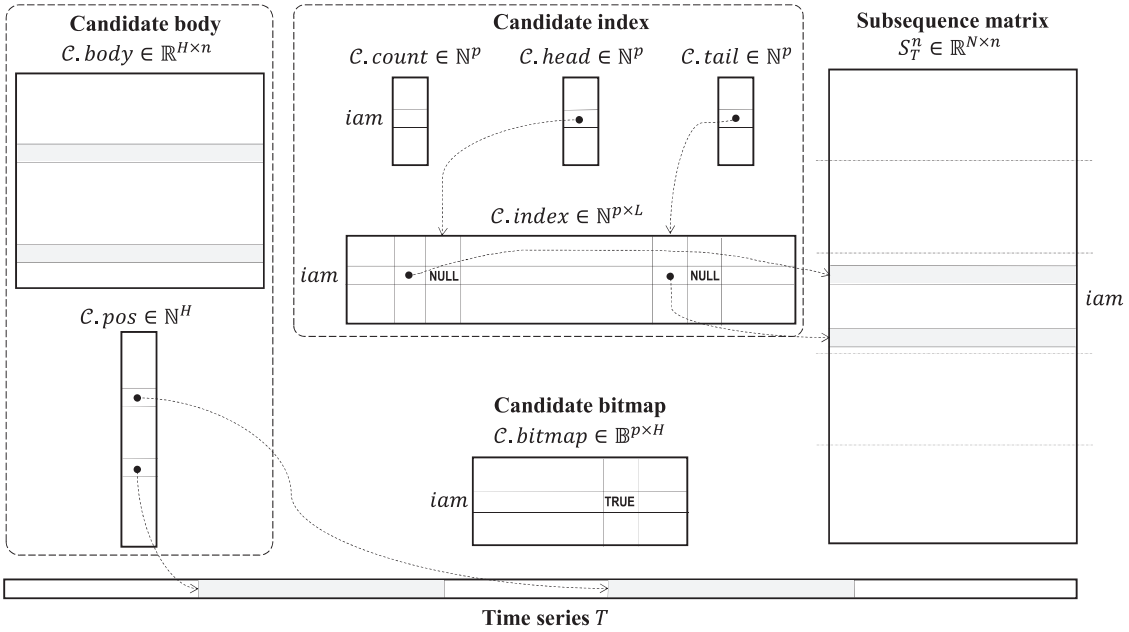

Figure 1: Data layout of the algorithm

Set of discords $\mathcal{C}$ is implemented as an object with two basic attributes, namely *candidate index* and *candidate body*, to store indices of all potential discord subsequences and their values themselves, respectively.

Let us denote a ratio of candidates selected at a cluster node and all subsequences of the time series by ξ. The exact value of the ξ parameter is a subject of an empirical choice. In our experiments, $\xi = 0.01$ was enough to store all candidates. Thus, we denote the number of candidates as $L = \lceil \xi \cdot N \rceil$ and assume that $L \ll N$.

The *candidate index* is organized as a matrix $\mathcal{C}.index \in \mathbb{N}^{p \times L}$, which stores indices of candidates in the subsequence matrix S_T^n found by each thread, i.e., *i*-th row keeps indices of the candidates that have been found by *i*-th thread. Initially, the candidate index is filled by NULL values.

To provide a fast access to the candidate index during the selection phase, it is implemented as a deque (double-ended queue) with three attributes, namely *count*, *head*, and *tail*. The *deque count* is an array $\mathcal{C}.count \in \mathbb{N}^{p}$, which for each thread keeps the amount of non-NULL elements in the respective row of the candidate index matrix. The *deque head* and *tail* are arrays $\mathcal{C}.head$ and $\mathcal{C}.tail \in \mathbb{N}^{p}$, respectively, which represent the second-level indices that for each thread keep a number of column in $\mathcal{C}.index$ with the most recent NULL value, and with the least recent non-NULL value, respectively.

Let H ($H < L \ll N$) be the number of candidates selected at a cluster node during the algorithm's first phase. Then the *candidate body* is the matrix $\mathcal{C}.cand \in \mathbb{R}^{H \times n}$, which represents the candidate subsequences itself. The candidate body is accompanied by an array $\mathcal{C}.pos \in \mathbb{N}^{H}$, which stores starting points of candidate subsequences in the input time series.

After the selection phase, all the nodes exchange the candidates found to construct the combined candidate set, so at each cluster node the candidate body will contain potential discords from all the nodes. At the second phase, the algorithm refines the combined candidate set comparing the parameter r and distances between each element of the candidate body and each element of the subsequence matrix.

To parallelize this activity, we process rows of the subsequence matrix in the segment-wise manner and employ an additional attribute of the candidate body, namely *bitmap*. The *bitmap* is organized as a matrix $\mathcal{C}.bitmap \in \mathcal{B}^{p \times H}$, which indicates the fact that an element of the candidate body has been successfully

validated against all elements in a segment of the subsequence matrix. Thus, after the algorithm's second phase, i-th element of the candidate body is successfully validated if $\wedge_{\ell=1}^{P} \mathcal{C}.bitmap(\ell, i)$ is true.

4.3 Parallel Implementation of the Algorithm

In the implementation, we apply the following parallelization scheme at the level of the cluster nodes. Let the input time series T be partitioned evenly across P cluster nodes. Each node performs the selection phase on its own partition with the same threshold parameter r and produces distinct candidate set $\mathcal{C}^i$.

Next, as opposed to MR-DADD [4], each node refines its own candidate set $\mathcal{C}^i$ with respect to the r value. Indeed, a candidate cannot be the true discord if it is pruned in the refinement phase in at least one cluster node. Thus, by the local refinement procedure, we try to reduce each candidate set $\mathcal{C}^i$ and, in turn, the combined candidate set $\mathcal{C}^P = \cup_{i=1}^{P} \mathcal{C}^i$. In the experiments, this allows us to reduce the size of the combined candidate set at times.

Then the combined candidate set $\mathcal{C}^P$ is constructed and transmitted to each cluster node. Next, a node refines $\mathcal{C}^P$ over its own partition, and produces the result $\mathcal{C}^i$. Finally, the true discords set is constructed as $\mathcal{C}^P = \cap_{i=1}^{P} \mathcal{C}^i$.

The parallel implementation of the candidate selection and refinement phases is depicted in Algorithm 2 and Algorithm 3, respectively. To speed up the computations at a cluster node, we omit the square root calculation since this does not change the relative ranking of the candidates (indeed, the ED function is monotonic and concave).

Algorithm 2: Parallel Candidate Selection (in T, r; out $\mathcal{C}$)

```
#pragma omp parallel
iam ← omp_get_thread_num()
#pragma omp for
for i from 1 to N do
  isCand ← TRUE
  for j from 1 to C.tail(iam) do
    if C.index(iam,j) = NULL or |C.index(iam,j) − i| < n then
      continue
    if ED²(S_T^n(i,·), S_T^n(C.index(iam,i),·)) < r² then
      isCand ← FALSE; C.count(iam) ← C.count(iam) − 1
      C.index(iam,j) ← NULL; C.head(iam) ← j
  if isCand then
    C.count(iam) ← C.count(iam) + 1
    if C.index(iam, C.head(iam)) = NULL then
      C.index(iam, C.head(iam)) ← i
    else
      C.index(iam, C.tail(iam)) ← i; C.tail(iam) ← C.tail(iam) + 1
return C
```

In the selection phase, we parallelize the outer loop along the rows of the subsequence matrix while in the inner loop along the candidates, each thread processes its own segment of the candidate index. By the end of the phase, the candidates found by each thread are placed into the candidate body, and all the cluster nodes

exchange the resulting candidate bodies by the MPI_Send and MPI_Recv functions to form the combined candidate set, which serves as an input for the second phase.

In the refinement phase, we also parallelize the outer loop along the rows of the subsequence matrix, and in the inner loop along the candidates, each thread processes its own segments of the candidate body and bitmap. In this implementation, we do not use the early abandoning technique for the distance calculation relying on the fact that vectorization of the square of the Euclidean distance may give us more benefits. By the end of the phase, the column-wise conjunction of the elements in the bitmap matrix will result in a set of true discords found by the current cluster node. An intersection of such sets is implemented by one of the cluster nodes where the rest nodes send their resulting sets.

Algorithm 3: Parallel Discord Refinement (in T, r; in out $\mathcal{C}$)

1: $\mathcal{C}.bitmap \leftarrow \text{TRUE}_{p \times H}$
2: **#pragma omp parallel**
3: $iam \leftarrow$ **omp_get_thread_num()**
4: **#pragma omp for**
5: **for** i **from** 1 **to** N **do**
6: **for** j **from** 1 **to** H **do**
7: $\mathcal{C}.bitmap(iam,j) \leftarrow \mathcal{C}.bitmap(iam,j)$ **and** $\left(\text{ED}^2\left(S_T^n(i,\cdot),\ \mathcal{C}.cand(j,\cdot)\right) \geq r^2\right)$
8: **return** $\mathcal{C}$

5 Experiments

We evaluated the proposed algorithm during the experiments conducted on the Tornado SUSU computer cluster [23] with the nodes based on the Intel MIC accelerators [24]. Each cluster node is equipped by the Intel Xeon Phi SE10X accelerator with a peak performance 1.076 TFLOPS (60 cores at 1.1 GHz with hyper-threading factor 4×). In the experiments, we investigated scalability of our approach and compared it with analogs, and the results are given below in Sections 5.1 and 5.2, respectively.

5.1 The Algorithm's Scalability

In the first series of the experiments, we assessed the algorithm's scaled speedup, which is defined as the speedup obtained when the problem size is increased linearly with the number of the nodes added to the computer cluster [25]. Being applied to our problem, the algorithm's scaled speedup is calculated as

$$s_{scaled} = \frac{n \cdot P \cdot \left|\mathcal{C}_{(P \cdot m)}\right|}{t_{P \cdot (P \cdot m)}}, \tag{7}$$

where n is the discord length, P is the number of the cluster nodes, m is a factor of the time series length, $\mathcal{C}_{(P \cdot m)}$ is a set of all the candidate discords selected by the algorithm at its first phase from a time series of length $P \cdot m$ and $t_{P \cdot (P \cdot m)}$ is the algorithm's run time when the time series is processed on P nodes.

For the evaluation, we took ECG time series [26] (see Tab. 1 for the summary of the data involved). In the experiments, we discovered discords on up to 128 cluster nodes with the time series factor $m = 10^6$, and varied the discord's length n while the range parameter r was chosen empirically to provide the algorithm's best performance.

The results of the experiments are depicted in Fig. 2. As can be seen, our algorithm adapts well to increasing both the time series length and number of cluster nodes, and demonstrates the linear scaled speedup. As expected, the algorithm shows a better scalability with larger values of the discord length because this provides a higher computational load.

Table 1: Time series involved in the experiments on the algorithm's scalability

# cluster nodes, P	Time series length, $P \cdot m$	$n = 128$, $r = 10$ # discords		$n = 256$, $r = 16$ # discords		$n = 512$, $r = 24.5$ # discords	
		candidate, $\lvert\mathcal{C}_{(P \cdot m)}\rvert$	refined, $\lvert\mathcal{C}\rvert$	candidate, $\lvert\mathcal{C}_{(P \cdot m)}\rvert$	refined, $\lvert\mathcal{C}\rvert$	candidate, $\lvert\mathcal{C}_{(P \cdot m)}\rvert$	refined, $\lvert\mathcal{C}\rvert$
1	10^6	5,407	772 (14%)	28,916	533 (2%)	17,366	548 (3%)
2	$2 \cdot 10^6$	15,115	3,245 (21%)	15,979	2,124 (13%)	28,376	1,174 (4%)
4	$4 \cdot 10^6$	23,990	3,199 (13%)	29,075	2,098 (14%)	54,281	1,169 (2%)
8	$8 \cdot 10^6$	45,989	3,167 (7%)	53,678	1,992 (4%)	109,890	1,070 (0.9%)
16	$1.6 \cdot 10^7$	117,659	2,779 (2%)	129,528	1,533 (1%)	299,157	1892 (0.3%)
32	$3.2 \cdot 10^7$	374,536	2,666 (0.7%)	294,844	1,288 (0.4%)	732,517	721 (0.09%)
64	$6.4 \cdot 10^7$	587,795	2,517 (0.4%)	707,956	1,036 (0.1%)	1,785,410	570 (0.03%)
128	$1.28 \cdot 10^8$	1,145,661	2,324 (0.2%)	1,541,099	764 (0.05%)	4,743,032	765 (0.02%)

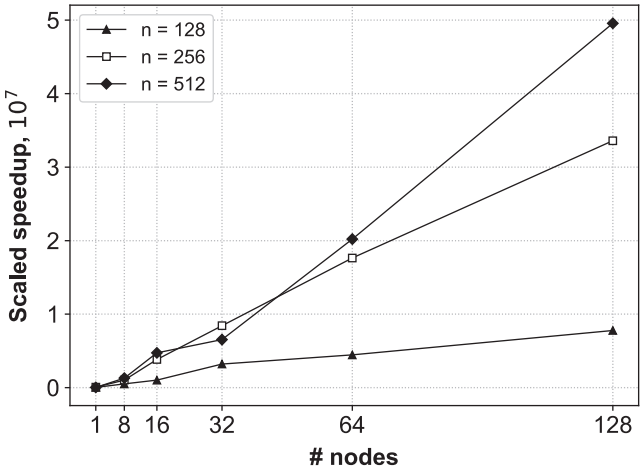

Figure 2: The scaled speedup of the algorithm

5.2 Comparison with Analogs

In the second series of the experiments, we compared the performance of our algorithm against the analogs we have already considered in Section 3, namely DDD [15], MR-DADD [4], GPU-STAMP [19], and MP-HPC [21]. We omit the PDD algorithm [18] since in our previous experiments [5], PDD was

substantially far behind our parallel in-memory algorithm due to the overhead caused by the message passing among the cluster nodes.

Throughout the experiments, we used the synthetic time series generated according the Random Walk model [27] as that ones were employed for the evaluation by the competitors. For comparison purposes, we used the run times reported by the authors of the respective algorithms. To perform the comparison, we ran our algorithm on Tornado SUSU with a reduced number of nodes and cores at a node to make the peak performance of our hardware platform approximately equal to that of the system on which the corresponding competitor was evaluated.

Tab. 2 summarizes the performance of the proposed algorithm compared with the analogs. We can see that our algorithm outruns its competitors. As expected, direct analogs DDD and MR-DADD are inferior to our algorithm since they do not employ parallelism within a single cluster node. Additionally, indirect analogs GPU-STAMP and MP-HPC are behind our algorithm since they initially aim to solve a computationally more complex problem of computing the matrix profile, which can also be used for discords discovery among many other time series mining problems.

Table 2: Comparison of the proposed algorithm with analogs

Analog		Our algorithm, hardware		Time series		Performance, s	
Algorithm	Hardware	# Cluster nodes	# Cores (threads) per node	$\|T\|$	n	Analog	Our algorithm
DDD	4 CPU @2.13 GHz	2	4 (16)	10^7	512	5,382	745
MR-DADD	8 CPU @3.0 GHz	2	8 (32)	10^6	512	240	99
GPU-STAMP	2880 CUDA cores @0.745 GHz	2	30 (120)	2^{21}	256	11,664	83
MP-HPC	39 CPU @2.6 GHz	4	60 (240)	$8{\cdot}10^5$	1000	6,000	32

6 Conclusion

In this article, we addressed the problem of discovering the anomalous subsequences in a very long time series. Currently, there is a wide spectrum of real-world applications where it is typical to deal with multi-terabyte time series, which cannot fit in the main memory: medicine, astronomy, economics, climate modeling, predictive maintenance, energy consumption, and others. In the study, we employ the discord concept, which is a subsequence of the time series that has the largest distance to its nearest non-self match neighbor subsequence.

We proposed a novel parallel algorithm for discords discovery in very long time series on the modern high-performance cluster with the nodes based on many-core accelerators. Our algorithm utilizes the serial disk-aware algorithm by Yankov, Keogh et al. [4] as a basis. We achieve parallelization among the cluster nodes as well as within a single node. At the level of the cluster nodes, we modified the original parallelization scheme that allowed us to reduce the number of candidate discords to be processed. Within a single cluster node, we proposed a set of the matrix data structures to store and index the subsequences of a time series, and to provide an efficient vectorization of computations on the many-core accelerator.

The experimental evaluation on the real computer cluster with the real and synthetic time series shows the high scalability of the proposed algorithm. Throughout the experiments on real computer cluster over real

and synthetic time series, our algorithm showed the linear scalability, increasing in the case of a high computational load due to a greater discord length. Also, the algorithm's performance was ahead of the analogs that do not employ both computer cluster and many-core accelerators.

In further studies, we plan to elaborate versions of the algorithm for computer clusters with GPU nodes.

Funding Statement: This work was financially supported by the Russian Foundation for Basic Research (Grant No. 20-07-00140) and by the Ministry of Science and Higher Education of the Russian Federation (Government Order FENU-2020-0022).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] E. J. Keogh, J. Lin and A. W. Fu, "HOT SAX: Efficiently finding the most unusual time series subsequence," in *Proc. ICDM*, Houston, TX, USA, pp. 226–233, 2005.

[2] E. J. Keogh, S. Lonardi and C. A. Ratanamahatana, "Towards parameter-free data mining," in *Proc. KDD*, Seattle, WA, USA, pp. 206–215, 2004.

[3] D. Yankov, E. J. Keogh and U. Rebbapragada, "Disk aware discord discovery: Finding unusual time series in terabyte sized datasets," in *Proc. ICDM*, Omaha, NE, USA, pp. 381–390, 2007.

[4] D. Yankov, E. J. Keogh and U. Rebbapragada, "Disk aware discord discovery: Finding unusual time series in terabyte sized datasets," *Knowledge and Information Systems*, vol. 17, no. 2, pp. 241–262, 2008.

[5] M. Zymbler, A. Polyakov and M. Kipnis, "Time series discord discovery on Intel many-core systems," in *Proc. PCT*, Kaliningrad, Russia, pp. 168–182, 2019.

[6] B. Y. Chiu, E. J. Keogh and S. Lonardi, "Probabilistic discovery of time series motifs," in *Proc. KDD*, Washington, D.C., USA, pp. 493–498, 2003.

[7] V. Chandola, D. Cheboli and V. Kumar, "Detecting anomalies in a time series database." Technical Report, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA, 2009.

[8] J. Lin, E. J. Keogh, S. Lonardi and B. Y. Chiu, "A symbolic representation of time series, with implications for streaming algorithms," in *Proc. DMKD*, San Diego, California, USA, pp. 2–11, 2003.

[9] J. Shieh and E. J. Keogh, "iSAX: Indexing and mining terabyte sized time series," in *Proc. KDD*, Las Vegas, Nevada, USA, pp. 623–631, 2008.

[10] H. T. Q. Buu and D. T. Anh, "Time series discord discovery based on iSAX symbolic representation," in *Proc. KSE*, Hanoi, Vietnam, pp. 11–18, 2011.

[11] A. W. Fu, O. T. Leung, E. J. Keogh and J. Lin, "Finding time series discords based on Haar transform," in *Proc. ADMA*, Xi'an, China, pp. 31–41, 2006.

[12] 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. NICS*, Danang, Vietnam, pp. 85–90, 2016.

[13] P. M. Chau, B. M. Duc and D. T. Anh, "Discord detection in streaming time series with the support of R-tree," in *Proc. ACOMP*, Ho Chi Minh City, Vietnam, pp. 96–103, 2018.

[14] G. Li, O. Bräysy, L. Jiang, Z. Wu and Y. Wang, "Finding time series discord based on bit representation clustering," *Knowledge-Based Systems*, vol. 54, pp. 243–254, 2013.

[15] Y. Wu, Y. Zhu, T. Huang, X. Li, X. Liu *et al.,* "Distributed discord discovery: Spark based anomaly detection in time series," in *Proc. HPCC, Proc. CSS, Proc. ICESS*, New York, NY, USA, pp. 154–159, 2015.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. HotCloud*, Boston, MA, 2010.

[17] S. Ghemawat, H. Gobioff and S. Leung, "The Google file system," in *Proc. SOSP*, Bolton Landing, NY, USA, pp. 29–43, 2003.

[18] T. Huang, Y. Zhu, Y. Mao, X. Li, M. Liu *et al.,* "Parallel discord discovery," in *Proc. PAKDD*, Auckland, New Zealand, pp. 233–244, 2016.

[19] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding *et al.,* "Time series joins, motifs, discords and shapelets: A unifying view that exploits the matrix profile," *Data Mining and Knowledge Discovery*, vol. 32, no. 1, pp. 83–123, 2018.

[20] Y. Zhu, C. M. Yeh, Z. Zimmerman, K. Kamgar and E. J. Keogh, "Matrix profile XI: SCRIMP++: Time series motif discovery at interactive speeds," in *Proc. ICDM*, Singapore, pp. 837–846, 2018.

[21] G. Pfeilschifter, "Time series analysis with matrix profile on HPC systems," Ph.D. dissertation. Department of Informatics, Technical University of Munich, Munich, Bavaria, Germany, 2019.

[22] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, 1994.

[23] P. Kostenetskiy and P. Semenikhina, "SUSU supercomputer resources for industry and fundamental science," in *Proc. GloSIC*, Chelyabinsk, Russia, 8570068, 2018.

[24] G. Chrysos, "Intel® Xeon Phi coprocessor (codename Knights Corner)," in *Proc. HCS*, Cupertino, CA, USA, pp. 1–31, 2012.

[25] A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*. 2nd ed. Boston, MA, USA: Addison-Wesley, 2003.

[26] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov *et al.,* "PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000.

[27] K. Pearson, "The problem of the random walk," *Nature*, vol. 72, no. 1865, pp. 294, 1905.