Time Series Discord Discovery on Intel Many-Core Systems

Mikhail Zymbler[1(✉)], Andrey Polyakov[1], and Mikhail Kipnis[2]

[1] South Ural State University, Chelyabinsk, Russia
mzym@susu.ru, avpgenium@gmail.com

[2] South Ural State Humanitarian and Pedagogical University, Chelyabinsk, Russia
mmkipnis@gmail.com

Abstract. A discord is a refinement of the concept of an anomalous subsequence of a time series. The task of discovering discords is applied in a wide range of subject areas involving time series: medicine, economics, climate modeling, and others. In this paper, we propose a novel parallel algorithm for discord discovery using Intel MIC (Many Integrated Core) accelerators in the case when time series fit in the main memory. We achieve parallelization through thread-level parallelism and OpenMP technology. 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 Intel MIC platform. Moreover, the algorithm exploits the ability to independently computing Euclidean distances between subsequences of a time series. The algorithm iterates subsequences in two nested loops; it parallelizes the outer and the inner loops separately and differently, depending on both the number of running threads and the cardinality of the sets of subsequences scanned in the loop. The experimental evaluation shows the high scalability of the proposed algorithm.

Keywords: Time series · Discord discovery · OpenMP · Intel Xeon Phi · Data layout · Vectorization

1 Introduction

The problem of finding an anomalous subsequence in a time series (i.e. a subsequence with the least similarity to any other subsequences) is one of the topical issues in time-series data mining and has applications in a wide range of subject areas: medicine, economics, climate modeling, predictive maintenance, energy consumption, and others.

In [9], Keogh *et al.* introduced the term *discord* to refine the concept of an anomalous subsequence. 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. Discords are attractive as anomaly detectors because they only require one intuitive parameter (the length of the subsequence), unlike most anomaly detection algorithms, which typically require many parameters [10]. The HOTSAX algorithm [9] employs SAX (Symbolic Aggregate ApproXimation) [14] transformation of subsequences and Euclidean distance for discord discovery. HOTSAX iterates subsequences according to an effective heuristics that allows for pruning large amounts of unpromising candidates for discord without calculating the distance.

In this paper, we address the task of accelerating the HOTSAX algorithm on Intel MIC (Many Integrated Core) systems [4,17]. MIC accelerators are based on the Intel x86 architecture and provide a large number of computing cores with 512-bit wide vector processing units (VPU) while supporting the same programming methods and tools as a regular Intel Xeon CPU. Intel provides two generations of MIC systems (under the codename of Intel Xeon Phi), namely Knights Corner (KNC), featuring 57 to 61 cores, and Knights Landing (KNL), featuring 64 to 72 cores. The benefits from the use of MIC accelerators usually manifest in applications where the processing of large amounts of data (at least hundreds of thousands of elements) can be arranged as loops that may be submitted to vectorization by a compiler [18]. Vectorization means the compiler's ability to transform the loops into sequences of vector operations [2] of VPUs.

In this study, we propose a parallel algorithm for discord discovery on Intel MIC systems, assuming that all the data involved in the computations fit into the main memory. The paper is structured as follows. In Sect. 2, we give the formal definitions along with a brief description of HOTSAX. Section 3 contains a short overview of related work. Section 4 presents the proposed parallel algorithm. In Sect. 5, we give the results of the experimental evaluation of our algorithm. Finally, Sect. 6 summarizes the results obtained and suggests directions for further research.

2 Problem Statement and the Serial Algorithm

2.1 Notations and Definitions

Below, we follow [9] 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: $T_{i,\,n} = (t_i, t_{i+1}, \ldots, t_{i+n-1})$, $1 \leq n \ll m$, $1 \leq i \leq m - n + 1$. We denote by S_T^n the set of all subsequences of length n in T. Let N denote the number of subsequences in S_T^n, i.e. $N := |S_T^n| = m - n + 1$.

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

We say that two subsequences $T_{i,\,n}, T_{j,\,n} \in S_T^n$ are a *non-self match* to each other at distance $\text{Dist}(T_{i,\,n}, T_{j,\,n})$ if $|i-j| \geq n$. A non-self match of a subsequence $C \in S_T^n$ is denoted by M_C.

A subsequence $D \in S_T^n$ is said to be a *discord* of T if it has the largest distance to its nearest non-self match. That is,

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

As the distance function, we use the ubiquitous Euclidean distance measure, defined as follows. Given two subsequences $X, Y \in S_T^n$, the Euclidean distance between them is calculated as

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

2.2 The Serial Algorithm

The HOTSAX algorithm [9] consists of two stages. At the first stage, HOTSAX converts each subsequence of the input time series into its SAX representation [14]. Then the algorithm forms an array of SAX words and counts how often each word occurs. Afterward, it builds a prefix trie with each leaf containing a list of all array indices that map to that terminal node. At the second stage, the algorithm scans subsequences via the trie and discovers a discord. We below describe these stages in more detail.

The algorithm consequentially applies z-normalization, PAA transformation, and SAX transformation to a given subsequence to produce its SAX word.

We define the *z-normalization* of a subsequence $C \in S_n^T$ as a subsequence $\hat{C} = (\hat{c}_1, \ldots, \hat{c}_n)$ in which

$$\begin{gathered} \hat{c}_i = \frac{c_i - \mu}{\sigma},\ 1 \le i \le n; \\ \mu = \frac{1}{n}\sum_{i=1}^{n} c_i,\ \sigma^2 = \frac{1}{n}\sum_{i=1}^{n} c_i^2 - \mu^2. \end{gathered} \tag{3}$$

The *PAA (Piecewise Aggregate Approximation)* [14] represents a subsequence $C = (c_1, \ldots, c_n)$ as a vector $\overline{C} = (\overline{c}_1, \ldots, \overline{c}_w)$ in a w-dimensional space, for a certain parameter $w \le n$; moreover, the i-th coordinate of $\overline{C}$ is calculated as follows:

$$\overline{c}_i = \frac{w}{n} \cdot \sum_{j=\frac{n}{w}\cdot(j-1)+1}^{\frac{n}{w}\cdot j} c_j. \tag{4}$$

Next, the PAA representation is coded through the *SAX (Symbolic Aggregate approXimation)* [14] transformation. For a given subsequence $C = (c_1, \ldots, c_n)$, its SAX word $\hat{C} = (\hat{c}_1, \ldots, \hat{c}_w)$ is produced as follows. Assume we have an alphabet $\mathcal{A} = (\alpha_1, \ldots, \alpha_{|\mathcal{A}|})$ to map a vector $\overline{C}$ into a word $\hat{C}$; here, $|\mathcal{A}|$ is the alphabet cardinality, and $\alpha_1 =$ 'a', $\alpha_2 =$ 'b', $\alpha_3 =$ 'c', and so on. Then,

$$\hat{c}_i = \alpha_i \Leftrightarrow \beta_{j-1} \le \hat{c}_i < \beta_j, \tag{5}$$

where β_i are *breakpoints* [14] defined as a sorted list of numbers, $\mathcal{B} := (\beta_0, \beta_1, \ldots, \beta_{|\mathcal{A}|-1}, \beta_{|\mathcal{A}|})$, such that $\beta_0 := -\infty$, $\beta_{|\mathcal{A}|} := +\infty$, and the area under the $N(0, 1)$

Gaussian curve between β_i and β_{i+1} equals $\frac{1}{|\mathcal{A}|}$. The breakpoints may be determined by looking them up in a statistical table [9]. It has been confirmed empirically that $w = 3, 4$ and $|\mathcal{A}| = 3, 4$ are suitable values for the discovery of time-series discords in a wide spectrum of subject areas [9].

Further, the algorithm produces an array of SAX words, counts the frequency of each word, and builds a prefix trie to store this information. The *prefix trie* [6] is a tree in which each edge is labeled with a symbol from the $\mathcal{A}$ alphabet in such a manner that all the edges connecting a node with its children are labeled with different symbols. The SAX word that corresponds to a leaf of the prefix trie is obtained by concatenation of all the characters that label the edges from the root to the leaf. Each leaf stores a sorted list of all array indices of the respective SAX word.

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

```
1: $dist_{bsf} \leftarrow 0$; $dist_{min} \leftarrow \infty$
2: for $C_i \in \big(NearDiscords \cdot Others\big)$ 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 $\{pos_{bsf}, dist_{bsf}\}$
```

Algorithm 1 presents the HOTSAX pseudo code. The algorithm takes a time series and a discord length and returns the index of the discord as well as the distance to its nearest neighbor subsequence. The algorithm looks through all the pairs of subsequences of the time series, calculates the Euclidean distance between them, and finds the maximum among the distances to nearest neighbors. The subsequence having the maximum distance to its nearest neighbor is a discord. The subsequences are iterated through two nested loops. Throughout the iterations, unpromising subsequences are pruned without calculating their distances. A subsequence is said to be unpromising if some of its neighbors is closer to it than the current maximum of the distances to all the nearest neighbors.

HOTSAX iterates subsequences according to a heuristic rule that enables pruning large amounts of unpromising subsequences. For this, the algorithm employs the following four sets of subsequences. The $NearDiscords$ set contains subsequences with the least frequent SAX words; the $Others$ set contains the rest of the subsequences. For a given subsequence C, the $Neighbors(C)$ set contains the subsequences whose SAX words match the SAX word of C. Conversely, the $Strangers(C)$ set contains the subsequences whose SAX words differ from that of C. The heuristics prescribes the following order for the iteration of subsequences. In the outer loop, subsequences from the $NearDiscords$ set should be considered first, and then those from the $Others$ set. In the inner loop, subsequences from the $Neighbors$ set are considered first, and then those from the $Strangers$ set.

3 Related Work

Following their introduction in [9], time-series discords and the HOTSAX algorithm have motivated considerable interest and follow-up work. Discords are applied for discovering abnormal heart rhythm in ECG [5], weird patterns of electricity consumption [1], unusual shapes [20], and others.

Among the attempts made to improve HOTSAX, we can mention the following. The *iSAX* algorithm [16] and the *HOTiSAX* algorithm [3] are the most direct enhancements to HOTSAX based on the indexable SAX transformation. A different approach is used in the *WAT* algorithm [7], in which Haar wavelets are employed instead of SAX transformations for time-series approximation. Another worth-noting example is the *Hash_DD* algorithm [19], which makes use of a hash table as an alternative to the prefix trie. We should also mention the *BitClusterDiscord* algorithm [13], which resorts to clustering of the bit representation of subsequences.

With reference to parallel discord discovery, we can draw attention to the following developments for computing systems with distributed memory. The *PDD (Parallel Discord Discovery)* algorithm [8] employs a Spark cluster [24] to split time series into fragments that are processed separately. PDD puts forward the Distributed Discord Estimation (DDE) method, which estimates the discord's distance to the nearest neighbor and minimizes the communication between computing nodes. Next, for each subsequence, PDD calculates the distance to its nearest neighbor and updates both pos_{bsf} and $dist_{bsf}$. A bulk of continuous subsequences is transmitted and calculated in batch mode to reduce message passing overhead. During this process, the early abandon technique is used to reduce computational complexity. Although PDD outruns HOTSAX, message passing between cluster nodes is a potential cause of significant degradation of the algorithm's performance as the number of nodes increases.

In [22], Yankov *et al.* propose a disk-aware algorithm (for brevity, referred to as *DADD, Disk-Aware Discord Discovery*) based on the concept of a *range discord*. For a given range r, the algorithm finds all discords at a distance of at least r from their nearest neighbor. The algorithm performs in two phases, namely candidate selection and discord refinement, with each phase requiring one linear scan through the time series on disk. By running HOTSAX for a uniformly random sample of the whole time series, which fits in the main memory, one can obtain the r parameter as the $dist_{bsf}$ value of the discord found [22]. Later, in [23], Yankov *et al.* presented a parallel version of DADD based on the MapReduce paradigm (for brevity, we denote the corresponding algorithm by *MR-DADD*). We should also mention the *DDD* (*Distributed Discord Discovery*) algorithm [21], which parallelizes DADD through a Spark cluster [24]. As opposed to DADD and MR-DADD, DDD computes the distance without taking advantage of an upper bound for early abandoning, which would increase the algorithm's performance.

To the best of our knowledge, no attempts have been made to parallelize HOTSAX for multi-core CPUs or many-core accelerators. Such an algorithm might be useful, though, both when time series fit in the main memory (e.g., to discover discords in ECG time series with tens of millions of elements) and in the case of disk-aware discord discovery as a sub-algorithm to obtain the r parameter quickly.

4 Accelerating Discord Discovery on Intel MIC Systems

The parallelization of the HOTSAX algorithm for the Intel MIC platform employs thread-level parallelism and OpenMP technology [15]. It is based on the following ideas: separate parallelization of outer and inner loops that iterate subsequences, use the square of the Euclidean distance, and use matrix representation of data.

At the first stage of HOTSAX, a *matrix data layout* enables an effective parallelization of computations in (3), (4), and (5) through OpenMP. In addition, matrix data structures are aligned in the main memory, so the calculations are organized with as many vectorizable loops as possible. We avoid unaligned memory access since it can cause inefficient vectorization due to time overhead for loop peeling [2]. Since OpenMP is more suitable to process matrices than trees, we employ a set of matrix data structures that store the same information as the prefix trie.

At the second stage of HOTSAX, distances between subsequences can be calculated independently by different threads of the parallel application. The parallel iteration of subsequences from the $NearDiscords$, $Others$, $Neighbors$, and $Strangers$ sets can be performed *separately* and *differently* for the outer and the inner loops, depending on the number of running threads and the cardinality of the above-mentioned sets. To speed up the computations in (2), *the square-root calculation can be omitted* since this does not change the relative ranking of potential discords (indeed, the ED function is monotonic and concave).

In Sects. 4.1 and 4.2, we will show an approach to the implementation of these ideas.

4.1 Parallel Implementation of the Algorithm

To parallelize HOTSAX, we split the algorithm into two steps, namely *finding* and *refinement* (see Algorithm 2 and Algorithm 3, respectively; cf. Algorithm 1). In the finding step, the iteration in the outer loop involves only subsequences from the $NearDiscords$ set. At the same time, only the inner loop is parallelized since the number of least frequent SAX words (i.e. the cardinality of the $NearDiscords$ set) is potentially less than the number of threads the algorithm is running on.

In the refinement step, the iteration in the outer loop involves only subsequences from the $Others$ set; the outer loop is parallelized, in view of the fact

Alg. 2 PhiDiscordDiscovery-Find(in T, n; out $pos_{bsf}, dist_{bsf}$)

```
1: $dist_{bsf} \leftarrow 0$; $dist_{min} \leftarrow \infty$
2: for $C_i \in NearDiscords$ do
3:     #pragma omp parallel for
4:     for $C_j \in \big(Neighbors(C_i) \cdot Strangers(C_i)\big)$ do
5:         $d \leftarrow \mathrm{ED}^2(C_i, C_j)$
6:         if $d < dist_{bsf}$ then
7:             break
8:         $dist_{min} \leftarrow \min(d, dist_{min})$
9:     $dist_{bsf} \leftarrow \max(dist_{min}, dist_{bsf})$
10:    if $dist_{bsf} < dist_{min}$ then
11:        $pos_{bsf} \leftarrow i$
12: return $\{pos_{bsf}, dist_{bsf}\}$
```

Alg. 3 PhiDiscordDiscovery-Refine(in T, n; out $pos_{bsf}, dist_{bsf}$)

```
1: $dist_{bsf} \leftarrow 0$; $dist_{min} \leftarrow \infty$
2: #pragma omp parallel for
3: for $C_i \in Others$ do
4:     for $C_j \in \big(Neighbors(C_i) \cdot Strangers(C_i)\big)$ do
5:         $d \leftarrow \mathrm{ED}^2(C_i, C_j)$
6:         if $d < dist_{bsf}$ then
7:             break
8:         $dist_{min} \leftarrow \min(d, dist_{min})$
9:     $dist_{bsf} \leftarrow \max(dist_{min}, dist_{bsf})$
10:    if $dist_{bsf} < dist_{min}$ then
11:        $pos_{bsf} \leftarrow i$
12: return $\{pos_{bsf}, \sqrt{dist_{bsf}}\}$
```

that the cardinality of the *Others* set is potentially greater than the number of threads the algorithm is running on.

In both steps, the loop is parallelized by the standard OpenMP compiler directive `#pragma omp parallel for`. Pruning unpromising subsequences results in uneven computational loading of threads. Therefore, to increase the efficiency of the parallel algorithm, we add to the above-mentioned `#pragma` the `schedule (dynamic)` parameter, which dynamically distributes loop iterations among threads. The statements in the loop body that calculates the squared Euclidean distances are vectorized by the compiler.

4.2 Internal Data Layout

The parallel algorithm employs the data structures depicted in Fig. 1. The time series is stored as a matrix of aligned subsequences to enable computations over aligned data with as many auto-vectorizable loops as possible. Let us denote by $width_{VPU}$ the number of floats stored in the VPU. If n (i.e. the length of the discord to be discovered) is not a multiple of $width_{VPU}$, then the subsequence

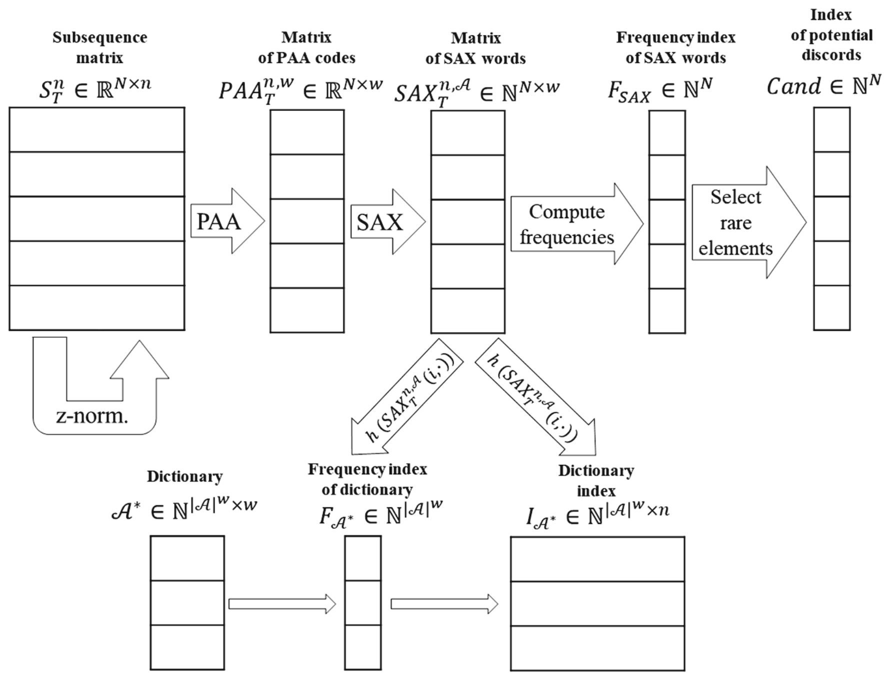

Subsequence matrix
$S_T^n \in \mathbb{R}^{N\times n}$
Matrix of PAA codes
$PAA_T^{n,w} \in \mathbb{R}^{N\times w}$
Matrix of SAX words
$SAX_T^{n,\mathcal{A}} \in \mathbb{N}^{N\times w}$
Frequency index of SAX words
$F_{SAX} \in \mathbb{N}^N$
Index of potential discords
$Cand \in \mathbb{N}^N$
PAA
SAX
Compute frequencies
Select rare elements
z-norm.
$h(SAX_T^{n,\mathcal{A}}(i,:))$
$h(SAX_T^{n,\mathcal{A}}(i,:))$
Dictionary
$\mathcal{A}^* \in \mathbb{N}^{|\mathcal{A}|^w\times w}$
Frequency index of dictionary
$F_{\mathcal{A}^*} \in \mathbb{N}^{|\mathcal{A}|^w}$
Dictionary index
$I_{\mathcal{A}^*} \in \mathbb{N}^{|\mathcal{A}|^w\times n}$

Fig. 1. Data layout of the algorithm

is padded with zeroes, with a number of zeroes $pad := width_{VPU} - (n \bmod width_{VPU})$. The aligned subsequence $\tilde{T}_{i,n}$ is defined as follows:

$$\tilde{T}_{i,n} := \begin{cases} t_i, t_{i+1}, \ldots, t_{i+n-1}, \overbrace{0, 0, \ldots, 0}^{pad}, & \text{if } n \bmod width_{VPU} > 0; \\ t_i, t_{i+1}, \ldots, t_{i+n-1}, & \text{otherwise.} \end{cases} \tag{6}$$

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

$$S_T^n(i,j) := \tilde{t}_{i+j-1}. \tag{7}$$

The *matrix of PAA codes* $PAA_T^{n,w} \in \mathbb{R}^{N\times w}$ contains the data calculated in accordance with (4).

The *matrix of SAX words* $SAX_T^{n,\mathcal{A}} \in \mathbb{N}^{N\times w}$ contains the data calculated according to (5).

The *index of potential discords* is an ascending-ordered array $Cand \in \mathbb{N}^N$ that contains the indices of those subsequences of S_T^n having the least frequent SAX words in $SAX_T^{n,\mathcal{A}}$. The index of potential discords determines the order of iteration of subsequences that can be discords and is formally defined in the following way:

$$\begin{aligned} Cand(i) = k \Leftrightarrow F_{SAX}(k) = \min_{1\le j\le N} F_{SAX}(j) \wedge \\ \forall i < j \quad Cand(i) < Cand(j). \end{aligned} \tag{8}$$

In (8), $F_{SAX} \in \mathbb{N}^N$ is an array that stores the *frequency indices of SAX words* and is defined as

$$F_{SAX}(i) = k \Leftrightarrow |\{j \mid SAX_T^{n,\mathcal{A}}(j,\cdot) = SAX_T^{n,\mathcal{A}}(i,\cdot)\}| = k. \tag{9}$$

The *dictionary* is a matrix $\mathcal{A}^* \in \mathbb{N}^{dict_{size} \times w}$ that contains all possible w-length words in the alphabet $\mathcal{A}$. The dictionary is generated according to the algorithm proposed in [11], and all symbols in a word (the elements in a row of the matrix), as well as all words (the rows of the matrix), are ascending-ordered. The cardinality of the dictionary $dict_{size}$ is calculated as follows:

$$dict_{size} := \bar{A}^w_{|\mathcal{A}|} = |\mathcal{A}|^w. \tag{10}$$

It has been empirically confirmed that such small values as $w = 4$ and $|\mathcal{A}| = 4$ are the best suited for virtually any time-series task from any subject area [9]. Therefore, the dictionary fits well in the main memory (indeed, $dict_{size} \times w = 4^4 \times 4 = 256$ elements).

Treating the $\mathcal{A}$ alphabet as an ordered sequence of natural numbers $1, \ldots, |\mathcal{A}|$, we can define a hash function $h \colon \mathbb{N}^w \to \{1, \ldots, dict_{size}\}$ to map the words of the alphabet, namely

$$h(a_1, \ldots, a_w) := \sum_{j=1}^{w+1} a_j \cdot w^{w-j-1}. \tag{11}$$

Next, the *dictionary index* is a matrix $I_{\mathcal{A}^*} \in \mathbb{N}^{dict_{size} \times N}$ comprising the indices of alphabet words contained in the matrix of SAX words. The dictionary index is defined as

$$I_{\mathcal{A}^*}(i,j) = k \Leftrightarrow \mathcal{A}^*(i,\cdot) = SAX_T^{n,\mathcal{A}}(k,\cdot). \tag{12}$$

Finally, the *frequency index of the dictionary* is an array $F_{\mathcal{A}^*} \in \mathbb{N}^{|\mathcal{A}|^w}$ whose elements are the numbers of occurrences of the words of the dictionary in the matrix of SAX words. The frequency index is defined in the following manner:

$$F_{\mathcal{A}^*}(i) = k \Leftrightarrow k = |\{j \mid \mathcal{A}^*(i,\cdot) = SAX_T^{n,\mathcal{A}}(j,\cdot)\}|. \tag{13}$$

5 The Experiments

5.1 The Experimental Setup

We evaluated the proposed algorithm in experiments conducted on Intel many-core systems at the Siberian Supercomputing Center[1] and the South Ural State University [12] (see Table 1 for a summary of the hardware involved).

In the first series of experiments, we assessed the performance and scalability of the algorithm while varying the discord length. We measured the run time (after deduction of the I/O time required for reading input data and writing the

[1] Hardware specifications of the Siberian Supercomputing Center.

Table 1. Hardware environment for the experiments

Characteristic	Intel Xeon CPU		Intel Xeon Phi Accelerator	
Model	E5-2630v4	E5-2697v4	SE10X (KNC)	7290 (KNL)
Physical cores	2×10	2×16	60	72
Hyperthreading factor	$2\times$	$2\times$	$4\times$	$4\times$
Logical cores	40	64	240	288
Frequency, GHz	2.2	2.6	1.1	1.5
VPU size, bit	256	256	512	512
Peak performance, TFLOPS	0.390	0.600	1.076	3.456

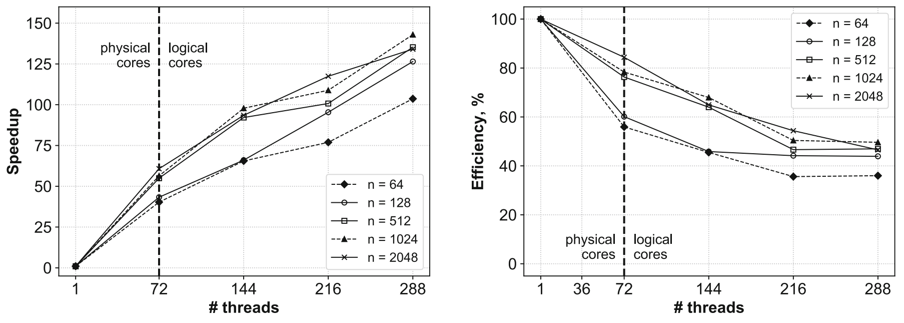
results) and calculated the algorithm's speedup and parallel efficiency, which are defined as follows. The speedup and the parallel efficiency of a parallel algorithm employing k threads are calculated, respectively, as $s(k) = \frac{t_1}{t_k}$ and $e(k) = \frac{t_1}{k \cdot t_k}$, where t_1 and t_k are the run times of the algorithm when one and k threads, respectively, are employed. We used two synthetic time series considered in [8] for evaluation of the PDD algorithm, namely SCD-1M and SCD-10M (with 10^6 and 10^7 elements, respectively).

In the second series of experiments, we compared the performance of our algorithm against analogs we have already considered: PDD, DDD, and MR-DADD (see Sect. 3). Throughout the experiments, we used the same datasets that were employed for the evaluation of the competitors. We ran our algorithm on an Intel Xeon Phi KNL system (see Table 1) with a reduced number of cores to make the peak performance of our accelerator approximately equal to that of the system on which the corresponding competitor was evaluated. For comparison purposes, we used the run times reported in the respective papers [8,21,23] (for the DDD and MR-DADD algorithms, we excluded the run time needed to calculate the r parameter).

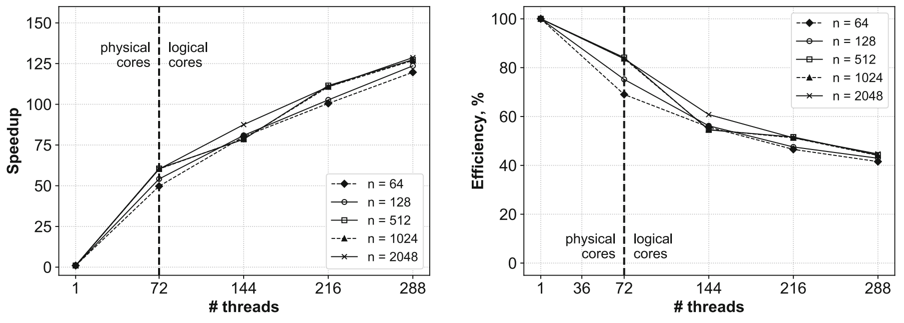
5.2 Results and Discussion

Figure 2 depicts experimental results regarding the algorithm's scalability on the Intel Xeon Phi KNL system. The algorithm showed a 40 to 60× speedup and a 50 to 85% parallel efficiency (with respect to the length of the discord that is to be found) when the number of threads matches the number of physical cores the algorithm runs on. As expected, the algorithm performs at its best with greater values of the m and n parameters (time-series length and discord length, respectively) because this provides a higher computational load. When more than one thread per physical core is employed, the algorithm shows sub-linear speedup and an accordingly diminished parallel efficiency, but without a tendency to stagnate or degrade.

Experimental results concerning the algorithm's performance on various Intel many-core systems are depicted in Fig. 3. The algorithm performs better on systems with greater numbers of cores. At the same time, when the algorithm

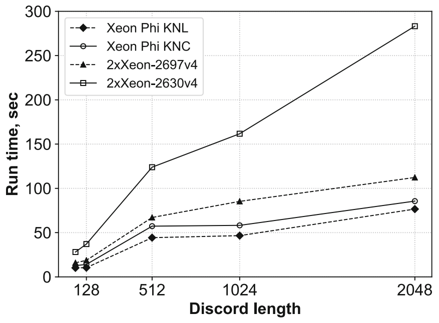


(a) SCD-1M dataset

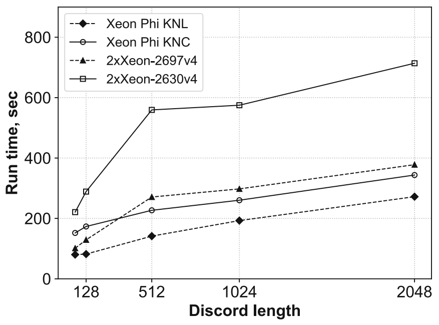


(b) SCD-10M dataset

Fig. 2. Scalability of the algorithm



(a) SCD-1M dataset



(b) SCD-10M dataset

Fig. 3. Performance of the algorithm

runs on the Intel MIC accelerator, it performs better than when it runs on a node equipped with two Intel Xeon CPUs.

Summing up, the proposed algorithm efficiently utilizes the vectorization capabilities of the many-core system and shows high scalability, especially in

Table 2. Comparison of the proposed algorithm with analogs

Experimental setup				Performance, s	
Analog		Time series length	# cores (threads) of Intel MIC	Our algorithm	Competitor
Competitor	Hardware				
MR-DADD [23]	8 CPU 3.0 GHz	10^6	8 (32)	101.6	240
DDD [21]	4 CPU 2.13 GHz	10^7	4 (16)	1 745.3	5 382
PDD [8]	10 CPU 1.2 GHz	10^7	10 (40)	833.3	399 600

case of high computational load due to greater time-series length and discord length (tens of millions and tens of thousands of elements, respectively).

Table 2 summarizes the performance of the proposed algorithm compared with analogs. We can see that our algorithm outruns its competitors. PDD is far behind due to a significant overhead caused by message passing among cluster nodes. The reason for DDD and MR-DADD being inferior to our algorithm is disk I/O overhead, which can amount to a half of the whole run time of the algorithm. In addition to the use of the main memory rather than the disk, our algorithm also takes advantage of the vectorization capabilities of the Intel MIC accelerator. We may conclude that the parallel algorithm we have proposed should be preferred over similar disk-aware parallel algorithms for discord discovery in the case when time series fit in the main memory.

6 Conclusions

In this paper, we addressed the task of accelerating the discovery of time series discords on Intel MIC (Many Integrated Core) systems. A discord is a refinement of the concept of an anomalous subsequence (i.e. a subsequence that is the least similar to all the other subsequences) of a time series. Discord discovery is applied in a wide range of subject areas involving time series: medicine, economics, climate modeling, and others.

We proposed a novel parallel algorithm for discord discovery on Intel MIC systems in the case of time series that fit in the main memory. Our algorithm parallelizes the serial HOTSAX algorithm by Keogh *et al.* [9]. The parallelization makes use of thread-level parallelism and OpenMP technology and is based on the following ideas: separation of parallelization of iteration loops, use of the squared Euclidean distance, and application of matrix layout for algorithm's data.

The proposed algorithm showed high scalability throughout the experimental evaluation, especially in the case of high computational load due to greater time series length and discord length (tens of millions and tens of thousands of elements, respectively). Moreover, the experiments showed that our algorithm outperforms analogous disk-aware parallel algorithms for discord discovery when time series fit in the main memory.

In further studies, we plan to elaborate versions of the algorithm for other hardware platforms, namely GPU accelerators and cluster systems with nodes based on Intel MIC or GPU accelerators.

Acknowledgments. This work was financially supported by the Russian Foundation for Basic Research (grant No. 17-07-00463) and by the Ministry of Science and Higher Education of the Russian Federation (government orders 2.7905.2017/8.9 and 14.578.21.0265).

The authors thank the Siberian Supercomputer Center (Novosibirsk, Russia) and the South Ural State University (Chelyabinsk, Russia) for the computational resources provided.

References

1. Ameen, J., Basha, R.: Mining time series for identifying unusual sub-sequences with applications. In: First International Conference on Innovative Computing, Information and Control, ICICIC 2006, Beijing, China, 30 August–1 September 2006, pp. 574–577. IEEE Computer Society (2006). https://doi.org/10.1109/ICICIC.2006.115
2. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**(4), 345–420 (1994). https://doi.org/10.1145/197405.197406
3. Buu, H.T.Q., Anh, D.T.: Time series discord discovery based on iSAX symbolic representation. In: 3rd International Conference on Knowledge and Systems Engineering, KSE 2011, Hanoi, Vietnam, 14–17 October 2011, pp. 11–18. IEEE Computer Society (2011). https://doi.org/10.1109/KSE.2011.11
4. Chrysos, G.: Intel® Xeon Phi coprocessor (codename Knights Corner). In: 2012 IEEE Hot Chips 24th Symposium (HCS), Cupertino, CA, USA, 27–29 August 2012, pp. 1–31 (2012). https://doi.org/10.1109/HOTCHIPS.2012.7476487
5. Chuah, M.C., Fu, F.: ECG anomaly detection via time series analysis. In: Thulasiraman, P., He, X., Xu, T.L., Denko, M.K., Thulasiram, R.K., Yang, L.T. (eds.) ISPA 2007. LNCS, vol. 4743, pp. 123–135. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74767-3_14
6. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960). https://doi.org/10.1145/367390.367400
7. Fu, A.W., Leung, O.T.-W., Keogh, E., Lin, J.: Finding time series discords based on haar transform. In: Li, X., Zaïane, O.R., Li, Z., et al. (eds.) ADMA 2006. LNCS (LNAI), vol. 4093, pp. 31–41. Springer, Heidelberg (2006). https://doi.org/10.1007/11811305_3
8. Huang, T., et al.: Parallel discord discovery. In: Bailey, J., Khan, L., Washio, T., Dobbie, G., Huang, J.Z., Wang, R. (eds.) PAKDD 2016. LNCS (LNAI), vol. 9652, pp. 233–244. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31750-2_19
9. Keogh, E.J., Lin, J., Fu, A.W.: HOT SAX: efficiently finding the most unusual time series subsequence. In: Proceedings of the 5th IEEE International Conference on Data Mining, ICDM 2005, Houston, Texas, USA, 27–30 November 2005, pp. 226–233. IEEE Computer Society (2005). https://doi.org/10.1109/ICDM.2005.79
10. Keogh, E.J., Lonardi, S., Ratanamahatana, C.A.: Towards parameter-free data mining. In: Kim, W., Kohavi, R., Gehrke, J., DuMouchel, W. (eds.) Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, 22–25 August 2004, pp. 206–215. ACM (2004). https://doi.org/10.1145/1014052.1014077

11. Knuth, D.: The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions. Addison-Wesley Professional, Boston (2005)
12. Kostenetskiy, P., Semenikhina, P.: SUSU supercomputer resources for industry and fundamental science. In: 2018 Global Smart Industry Conference (GloSIC), Chelyabinsk, Russia, 13–15 November 2018, p. 8570068 (2018). https://doi.org/10.1109/GloSIC.2018.8570068
13. Li, G., Bräysy, O., Jiang, L., Wu, Z., Wang, Y.: Finding time series discord based on bit representation clustering. Knowl.-Based Syst. **54**, 243–254 (2013). https://doi.org/10.1016/j.knosys.2013.09.015
14. Lin, J., Keogh, E.J., Lonardi, S., Chiu, B.Y.: A symbolic representation of time series, with implications for streaming algorithms. In: Zaki, M.J., Aggarwal, C.C. (eds.) Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD 2003, San Diego, California, USA, 13 June 2003, pp. 2–11. ACM (2003). https://doi.org/10.1145/882082.882086
15. Mattson, T.: Introduction to OpenMP. In: Proceedings of the ACM/IEEE SC 2006 Conference on High Performance Networking and Computing, Tampa, FL, USA, 11–17 November 2006, p. 209. ACM Press (2006). https://doi.org/10.1145/1188455.1188673
16. Shieh, J., Keogh, E.J.: *i*SAX: indexing and mining terabyte sized time series. In: Li, Y., Liu, B., Sarawagi, S. (eds.) Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, 24–27 August 2008, pp. 623–631. ACM (2008). https://doi.org/10.1145/1401890.1401966
17. Sodani, A.: Knights Landing (KNL): 2nd generation Intel® Xeon Phi processor. In: 2015 IEEE Hot Chips 27th Symposium (HCS), Cupertino, CA, USA, 22–25 August 2015, pp. 1–24. IEEE (2015). https://doi.org/10.1109/HOTCHIPS.2015.7477467
18. Sokolinskaya, I., Sokolinsky, L.: Revised pursuit algorithm for solving non-stationary linear programming problems on modern Computing clusters with manycore accelerators. In: Voevodin, V., Sobolev, S. (eds.) RuSCDays 2016. CCIS, vol. 687, pp. 212–223. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-55669-7_17
19. Thuy, H.T.T., Anh, D.T., Chau, V.T.N.: An effective and efficient hash-based algorithm for time series discord discovery. In: 2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS), Danang, Vietnam, 14–16 September, pp. 85–90 (2016). https://doi.org/10.1109/NICS.2016.7725673
20. Wei, L., Keogh, E.J., Xi, X.: SAXually explicit images: finding unusual shapes. In: Proceedings of the 6th IEEE International Conference on Data Mining, ICDM 2006, Hong Kong, China, 18–22 December 2006, pp. 711–720. IEEE Computer Society (2006). https://doi.org/10.1109/ICDM.2006.138
21. Wu, Y., Zhu, Y., Huang, T., Li, X., Liu, X., Liu, M.: Distributed discord discovery: Spark based anomaly detection in time series. In: 17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, 24–26 August 2015, pp. 154–159. IEEE (2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.228

22. Yankov, D., Keogh, E.J., Rebbapragada, U.: Disk aware discord discovery: finding unusual time series in terabyte sized datasets. In: Proceedings of the 7th IEEE International Conference on Data Mining, ICDM 2007, Omaha, Nebraska, USA, 28–31 October 2007, pp. 381–390. IEEE Computer Society (2007). https://doi.org/10.1109/ICDM.2007.61
23. Yankov, D., Keogh, E.J., Rebbapragada, U.: Disk aware discord discovery: finding unusual time series in terabyte sized datasets. Knowl. Inf. Syst. **17**(2), 241–262 (2008). https://doi.org/10.1007/s10115-008-0131-9
24. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Nahum, E.M., Xu, D. (eds.) 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2010, Boston, MA, USA, 22 June 2010. USENIX Association (2010)