Parallel Discord Discovery

Tian Huang[1], Yongxin Zhu[1], Yishu Mao[1], Xinyang Li[1], Mengyun Liu[1], Yafei Wu[1], Yajun Ha[2], and Gillian Dobbie[3]

[1] School of Microelectronics, Shanghai Jiao Tong University, China
{ian_malcolm,zhuyongxin,maoyishu,all-get, old-ant,wuyf0406}@sjtu.edu.cn
[2] Institute for Infocomm Research, A*STAR, Singapore,
ha-y@i2r.a-star.edu.sg
[3] Department of Computer Science, University of Auckland, New Zealand,
g.dobbie@auckland.ac.nz

Abstract. Discords are the most unusual subsequences of a time series. Sequential discovery of discords is time consuming. As the scale of datasets increases unceasingly, datasets have to be kept on hard disk, which degrades the utilization of computing resources. Furthermore, the results discovered from segmentations of a time series are non-combinable, which makes discord discovery hard to parallelize. In this paper, we propose Parallel Discord Discovery (PDD), which divides the discord discovery problem in a combinable manner and solves its sub-problems in parallel. PDD accelerates discord discovery with multiple computing nodes and guarantees the correctness of the results. PDD stores large time series in distributed memory and takes advantage of in-memory computing to improve the utilization of computing resources. Experiments show that given 10 computing nodes, PDD is seven times faster than the sequential method HOTSAX. PDD is able to handle larger datasets than HOTSAX does. PDD achieves over 90% utilization of computing resources, nearly twice as much as the disk-aware method does.

Keywords: Time series discord, Parallel, Large scale, In-memory computing

1 Introduction

Time series discords are the subsequences of a time series that are maximally different to all the rest subsequences [11, 8]. Discord can be found by computing the pair-wise distances among all subsequences of a time series. Recently, finding time series discord has attracted much attention. The definition, despite its simplicity, captures an important class of anomalies. Its relevance has been shown in several data mining applications [9, 13, 19, 3, 16].

Various memory based methods, e.g. the classical method HOTSAX [11], have been proposed to speed up discord discovery, but they are still time consuming when the datasets are large. For example, to discover the top discord from an ECG time series that contains 648K data instances, a Java implementation of HOTSAX on a modern PC costs about 20 minutes, which is close to the

time period of the ECG time series. In fact, any dataset in existing literature of memory based methods contains no more than 64K data instances. We believe the time consumption is one of the main limitations to discover the discords of larger time series.

Low utilization of computing resources is another issue when discovering discords from large datasets. A disk-aware method [20] discovers discords from 100-million scale time series, which can only be fitted onto hard disks. Disk I/Os take more than 50% of the total time consumption, leaving the computing resources in idle state for most of the time during the discord discovery.

Parallel computing is a potential way to mitigate these issues. However discord discovery is hard to parallelize. The results discovered from segmentations of a time series are non-combinable [11], which means divide-and-conquer methodology may leads to incorrect results that do not conform to the original definition of discord.

To mitigate the above issues, we propose Parallel Discord Discovery (PDD), which divides discord discovery problem in a combinable manner and solves its sub-problems in parallel. PDD stores a long time series in distributed memory of multiple computing nodes. These nodes work together to reduce the time consumption of discord discovery. As far as we know, this is the first work that discovers time series discords in parallel. Our contributions are summarized as follows:

- We accelerate discord discovery by harnessing multiple computing nodes.
- We ensure the correctness of the results by dividing discord discovery problem in a combinable manner.
- We improve the utilization of computing resources in large scale discord discovery by using an in-memory computing framework.

We implement PDD using Apache Spark [18]. Experiments show that given 10 computing nodes, PDD achieves 7 times speedup against HOTSAX. PDD is able to handle larger datasets than traditional memory based methods. PDD achieves nearly twice the utilization of computing resources compared to the disk-aware method [20].

The rest of the paper is organized as follows. Section 2 presents related works and their issues. Section 3 analyzes the feasibility of parallelization and describes the detailed implementation of the Parallel Discord Discovery (PDD) method. Section 4 presents an empirical evaluation of PDD. Section 5 draws the conclusion.

2 Related Works and Issues

Discords can be discovered by comparing every pair of subsequences with two-layer nested for-loops. The outer loop considers each possible candidate subsequence, and the inner loop is a linear scan to identify the non-overlapping nearest neighbor of the candidates. The computational complexity of a naive discord discovery method is $O(m^2)$, where m is the size of the dataset. Various methods are

proposed to accelerate discord discovery. Keogh et al. propose HOTSAX [11], which applies heuristic sorting techniques and early abandon technique to reduce the computational complexity. The heuristic order of the outer loop helps HOTSAX visit the most unusual subsequences in the first few iterations, while the heuristic order of the inner loop helps HOTSAX visit the subsequences that are similar to the current candidate subsequence. A conditional branch in the inner loop helps HOTSAX skip the calculations of the normal subsequences. HOTSAX speeds up by three orders of magnitude compared with the naive method.

Many efforts have been made to improve discord discovery from various aspects. [3, 7, 12] introduce different feature extraction methods to provide better heuristic inner and outer order of the nested loop. They reduce the dimensionality of time series data and the computational complexity of discord discovery. [1–3, 7, 10, 14, 15] mitigate the negative effects of improper determination of parameters to the computational efficiency of discord discovery. [4, 5, 12] use elaborate indexes to find nearest neighbor distance more efficiently and reduce the computational complexity of discord discovery.

Most existing discovery methods assume that datasets are small enough to be stored in the memory of a single computer. Yankov et al. propose a disk-aware method [20] to deal with the time series whose scale grows beyond the capacity of the memory of a single computing node. The method improves the efficiency of the disk I/O operation by linearly scanning the disk to obtain all subsequences. The disk-aware method eliminates the limitation of the memory based method in terms of the scale of datasets.

Previous methods are all sequential methods. They suffer from the limitation of computing power and storage of a single computing node. As the scale of time series increases unceasingly, the utility of discord discovery deteriorates. Besides, the disk-aware method suffers from the low utilization of computing resources.

Although parallel computing is a potential way to mitigate these issues, discord discovery is hard to parallelize. The results of discord discovery is non-combinable [11]. In other words, divide-and-conquer methodology may lead to incorrect results, which do not conform to the original definition of discord.

In this paper we mitigate the issues by proposing Parallel Discord Discovery (PDD). PDD enables accelerating discord discovery with multiple computing nodes. We implement PDD with Apache Spark so that PDD has better utilization of computing resources than the disk-aware method.

3 Parallel Discord Discovery

To parallelize discord discovery, we must divide the problem into independent sub-problems, and solve each sub-problem respectively in different computing nodes. [11] states that divide-and-conquer methodology may yield incorrect results. We need to find another feasible way to partition discord discovery problem. To better analyze the feasibility of parallel time series discord discovery, we review the concept of discord, analyze the communication computation ratio and the parallelism.

3.1 Dividing the Problem

Before analyzing the formal definition of discord, we describe a set of preliminary notations [11]. We use T to denote a *time series* $t_1, ..., t_m$, where m is the length of the time series, $t_i \in \mathbb{R}$. We denote a *subsequence* of a time series T as $C_{p,n} = t_p, ..., t_{p+n-1}$, where p is the starting position, n is the length of the subsequence. Because subsequences of the same length are compared in a discovery of discords, we use C_p as an abbreviation of $C_{p,n}$ in the rest of the paper. Since every subsequence could be a discord, we will use a *sliding window*, whose size is equal to the length of subsequences, to extract all possible subsequences from T. The process of time series discord discovery can be expressed by the following equation:

$$p^{(1)} = \underset{p}{\operatorname{argmax}}\{nnDist(C_p)|1 \leq p \leq m - n + 1\} \tag{1}$$

$p^{(1)}$ indicates the start position of the first discord. $nnDist(C_p)$ is the nearest neighbor distance of a subsequence C_p. n is the length of the sliding window. m stands for the length of the time series. argmax in Eq.(1) means the subsequence with the largest $nnDist$, which is the discord of the time series.

As shown in Eq.(1), the processes of solving each the nearest neighbor distance $nnDist(C_p)$ of each subsequence C_p is independent. Therefore, we divide discord discovery into independent sub-problems. Each sub-problem finds the nearest neighbor of the subsequence C_p and the distance between them. The results of all sub-problems can be combined. The subsequences with the largest $nnDist$ is the discords of the time series.

3.2 Improving Computation Communication Ratio

To find the nearest neighbor of a subsequence C_p, we transmit C_p and any other non-overlapping [6] subsequence to one or more computing nodes to calculate the distance between them. However, the time consumption of transmitting two subsequences and computing the distance between them is of the same order of magnitude. This will results in low Computation Communication Ratio (CCR) and therefore low utilization of computing resources. We have to improve CCR.

We utilize the overlapped region between two adjacent subsequences to improve CCR. We transmit continuous data instances so that the overlapped region can be reused by multiple subsequences. Taking the sliding window of length $n = 100$ as an example, the transmission of the first 100 data forms one subsequence. Each transmitted data instance forms a new subsequence with the previous transmitted 99 data instances. If we transmit 299 continuous data instances, we get $299 - 100 + 1 = 200$ subsequences of length $n = 100$. Therefore, CCR is improved by a factor of $\frac{200}{299} \div \frac{1}{100} \approx 67$.

3.3 Overview of PDD

We divide discord discovery in a combinable manner. Next we design Parallel Discord Discovery (PDD). We first present the overview of PDD.

Data: All subsequences $\mathbb{C}$ and #subsequences/bulk b
Result: Position $bsfPos$ and nearest neighbor distance $bsfDist$ of the discord
1 Initialize $bsfDist = 0$, $bsfPos = null$;
2 Find the estimation $\tilde{d}$ of $nnDist$ of discord;
3 // see Section 3.4 Update $bsfDist = \tilde{d}$;
4 **for** *each b subsequence* $C_p, \ldots, C_{p+b-1}$ *of* $\mathbb{C}$ **do**
5 | // see Section 3.5 $i = \arg\max_i \{nnDist(C_{p+i}) | 0 \geq i \geq b-1\}$;
6 | **if** $bsfDist \leq nnDist(C_{p+i})$ **then**
7 | | Update $bsfDist = nnDist(C_{p+i})$;
8 | | Update $bsfPos = p + i$;
9 | **end**
10 **end**

Algorithm 1: The pseudocode of PDD

Algorithm 1 shows the overview of PDD. $\mathbb{C}$ is the set of all possible subsequences of a time series. b is the number of continuous subsequences in each transmission. This method has two steps. First, PDD estimates global $nnDist$ of the discord with Distributed Discord Estimation (DDE) method (line 2-3). Then PDD linearly scans $\mathbb{C}$ to find the $nnDist$ of the true discord of the time series. This linear scanning step consists of multiple rounds. Continuous subsequences are transmitted in batches among computing nodes for better CCR. In each round, computing nodes work separately and exchange intermediate results at the end of the round (line 5). After each round, the current best-so-far distance ($bsfDist$) is updated (line 6-9). DDE and linear scanning (line 4) are the most time consuming parts. We present the detailed implementation of these parts.

3.4 Distributed Discord Estimation

The first step of discord discovery is to estimate $nnDist$ of discord. The estimation together with an early abandon technique can efficiently reduce the number of calls to the distance function (Algorithm 1, line 2). The closer the estimation is to the ground truth, the less the distance function is invoked. Traditional methods usually set an index to achieve this. However, when a time series is divided into several segments and stored into non-unified memory spaces, creating a centralized index for these distributed data is inefficient because it degrades the CCR. We propose a method called Distributed Discord Estimation (DDE), which estimates the distance and minimizes the communication between computing nodes.

In Algorithm 2, $\mathbb{C}$ represents the set of all subsequences. $\mathbb{S}$ is the segment information of a time series. DDE outputs the estimated $nnDist$, which is $\tilde{d}$. If the real $nnDist$ of the discord is $nnDist(C_d)$, then $\tilde{d} \leq nnDist(C_d)$.

DDE approximates every subsequence with a symbolic representation(line 1-3). The approximated representation of a subsequence C_p is denoted as A_p. Similar subsequences have the same approximation symbol. DDE divides the subsequences into groups according to their symbolic representations. The num-

Data: All subsequences $\mathbb{C}$ and Segmenting Information $\mathbb{S}$
Result: Nearest neighbor distance estimation $\tilde{d}$ of discord
1 **for** *Each* C_p *in* $\mathbb{C}$ **do**
2 | Calculate A_p of C_p;
3 **end**
4 Group C_p by A_p;
5 Find $A_{\tilde{d}}$ that is the A_p with smallest group of C_p;
6 **for** *each* C_p *in* $A_{\tilde{d}}$ **do**
7 | **for** *each* S *in* $\mathbb{S}$ **do**
8 | | Calculate local $nnDist$ of C_p in S;
9 | **end**
10 | Calculate global $nnDist$ of C_p by finding its min local $nnDist$;
11 **end**
12 $\tilde{d} = \max\{\text{global } nnDist(C_p) | C_p \text{that can be approximated as } A_{\tilde{d}}\}$;

Algorithm 2: Distributed Discord Estimation

ber of subsequences in one group reflects the frequency of the group (line 4). DDE selects the group with the least number of members, named $A_{\tilde{d}}$, as the candidates of discords. For each subsequence C_p in this $A_{\tilde{d}}$ group (line 6), DDE finds a local $nnDist(C_p)$ for each S. The minimum of all local $nnDist(C_p)$ is called global $nnDist(C_p)$. The maximum among all global $nnDist(C_p)$ is the estimation of the global discord $nnDist$ of the time series.

The number of subsequence in the $A_{\tilde{d}}$ group, denoted as $|A_{\tilde{d}}|$, affects the accuracy of the estimation. Sometimes $|A_{\tilde{d}}|$ is too small for DDE to get a reliable estimation. Empirically we select 2 to 10 groups of subsequences to get better precision for the estimation.

As we can see, the computing process of the approximation is independent, which means it can be concurrently executed on a computing cluster. Each computing node processes different subsequences at the same time. Then PDD aggregates all data to find the globally estimated $nnDist$.

3.5 Linearly Scanning the Entire Dataset

The second step of PDD is to calculate the $nnDist$ of every subsequence and update $bsfPos$ and $bsfDist$. Linear scanning is the most time consuming part of PDD. In order to improve the CCR, a bulk of continuous subsequences is transmitted and calculated in batch. During this process, early abandon technique is used to reduce the computational complexity. We describe the linear scanning step from two perspectives.

Life Cycle of a Bulk of Subsequences Algorithm 3 explains the life cycle of a bulk of subsequences. The inputs of this pseudocode are formed by a bulk of subsequences $\mathbb{C}_b$, the set of segment $\mathbb{S}$, and $bsfDist$.

As the life cycle of a bulk of subsequences $\mathbb{C}_b$ starts at the beginning of Algorithm 3, the $nnDist$ of all subsequences in the bulk is initialized as positive infinity (line 1). $\mathbb{C}_b$ visits all computing nodes, each of which contains different

Data: A bulk of subsequence $\mathbb{C}_b$, Segmenting Information $\mathbb{S}$ and $bsfDist$
Result: Position $bsfPos$ and nearest neighbor distance $bsfDist$ of the discord

```
1  nnDist[:] = positive infinity;
2  for each S in 𝕊 do
3  |  for each C_p in ℂ_b do
4  |  |  for each C_q in S do
5  |  |  |  if nnDist[p] > dist(C_p, C_q) then
6  |  |  |  |  nnDist[p] = dist(C_p, C_q);
7  |  |  |  end
8  |  |  |  if nnDist[p] < bsfDist then
9  |  |  |  |  ℂ_b = ℂ_b/C_p;
10 |  |  |  |  continue to next C_p;
11 |  |  |  end
12 |  |  end
13 |  end
14 |  if nnDist[p] ≥ bsfDist then
15 |  |  bsfPos = p;
16 |  end
17 end
18 bsfDist = nnDist(bsfPos);
```

Algorithm 3: Linear scanning: life cycle of a bulk of subsequences

segment of the time series (line 3). During this process, the local $nnDist$ of the subsequence C_p with respect to the current segment is calculated (line 4-7). If the global best so far Distance ($bsfDist$) is larger than the $nnDist[p]$ (line 8), subsequence C_p could be abandoned from $\mathbb{C}_b$ (line 9), and the follow-up calculating about C_p is skipped (line 10). Heuristic access order improves the efficiency of the early abandon technique (line 5). After calculating the distance between C_p and all other subsequences in this segment without triggering the early abandon technique, the $nnDist[p]$ becomes the global $nnDist$ of C_p. Under this circumstance it must hold $nnDist[p] \geq bsfDist$. Hence $bsfPos$ should be pointed to current subsequences(line 14-15). $bsfDist$ must be synchronously shared by all computing nodes, and updated after a bulk visit to all computing nodes (line 18).

During the linear scanning, relevant intermediate results of bulk $\mathbb{C}_b$, including the $nnDist$ of every subsequence $nnDist[:]$, are transmitted between computing nodes (line 3). The DDE method mentioned above initiates the $bsfDist$ as much as possible in order to call the Early Abandon Technology more frequently(line 8-10). Therefore most subsequences of bulk $\mathbb{C}_b$ are dumped and the number of subsequences in bulk $\mathbb{C}_b$ continually decreases. So the number of data and intermediate results being transmitted is quite small.

Communication among Computing Nodes In each round, computing nodes receive bulk, work separately and exchange intermediate results with other computing nodes. We give an example to explain the details. The time series is divided into segments $S_0 \ldots S_{(a-1)}$ and restored in nodes $node_0 \ldots node_{(a-1)}$.

The subsequences are grouped into bulks $b_0, b_1, b_2 \ldots$. Table 1 explains the timing of transmissions among computing nodes.

Table 1. The timing of transmission among computing nodes

	$node_0(S_0)$	$node_1(S_1)$	...	$node_{a-1}(S_{a-1})$
R_0	b_0	b_1	...	b_{a-1}
R_1	$b_a + b_{a-1}$	$b_{a+1} + b_1$	...	$b_{2a-1} + b_{a-2}$
R_2	$b_{2a} + b_{2a-1} + b_{a-2}$	$b_{2a+1} + b_a + b_{a-1}$	...	$b_{3a-1} + b_{2a-2} + b_{a-3}$
...	...	...	...	...
R_{a-1}	$b_{a(a-1)} + \sum_{i=0}^{a-2} b_{ai+i+1}$	$b_{a(a-1)+1} + \sum_{i=0}^{a-2} b_{ai+i+2}$	...	$b_{a(a-1)+a-1} + \sum_{i=0}^{a-2} b_{ai+i+aa}$
...	...	...	...	...
R_t	$b_{at} + \sum_{i=t-a}^{t-1} b_{ai+i+1}$	$b_{at} + \sum_{i=t-a}^{t-1} b_{ai+i+2}$	...	$b_{at} + \sum_{i=t-a}^{t-1} b_{ai+i+a}$

Allocating new bulk At the beginning of each round, each node receives a new bulk, which has never visited other nodes. The *nnDist* of every subsequence in this new bulk is set as $+\inf$ (line 1 in Algorithm 3). For example, the node $node_{a-1}$ receives new bulk $b_{a-1}, b_{2a-1}, b_{3a-1}, \ldots$ at round $R_0, R_1, R_2, \ldots$. This process continues until all new bulks are allocated to nodes.

Transmission between computing nodes Every bulk is sent to all computing nodes to find the global *nnDist* of every subsequence. After one node finishes the *nnDist* of subsequences of a bulk against one segment (e.g. S_0), the bulk is sent to the next node (e.g. $node_1$). There are many paths to transmit between nodes, and we choose the rotation shift method. For example, at R_1, bulks $b_{a+1} + b_0$ are in the $node_1$. At R_2, the bulks are sent to $node_2$. At R_{a-1}, the bulks are sent to $node_{a-1}$. At this time bulk b_0 has traversed to all nodes (which means all data segments) and will not be sent to $node_0$ at R_a. The computing of b_0 finishes here.

3.6 Improving Utilization of Computing Resources

Load imbalance may occurs among computing nodes. Some computing nodes finish computations and exchanges earlier and enter an idle state before the round ends. The idle state degrades the utilization of computing resource.

The idle state can be alleviated by allocating extra bulks and using a shared input queue. In each round, PDD creates a shared queue which contains bulks several times the number of the computing nodes. Once a computing node finishes the processing of a bulk, the node is assigned to the next bulk from the shared queue. The round is finished when all bulks in the shared queue are processed.

3.7 Guarantee on Correctness of PDD

PDD discovers exact results, which completely conform to the definition of discord. PDD provides the guarantee on the correctness of results by discovering discords according to the definition of discord.

More specifically, PDD finds the $nnDist$ of a candidate subsequence C_p (line 3 in Algorithm 3) by calculating the distances (line 5-7 in Algorithm 3) of C_p and every other subsequence C_q (line 2 and line 4 in Algorithm 3) of a time series. PDD finds the $nnDist$ of all candidate subsequences (line 4 in Algorithm 1) and returns the subsequence with the maximum $nnDist$ (line 6-9 in Algorithm 1) as the top discords.

DDE improves the performance of PDD but has no impact on the correctness of PDD. PDD still produces correct results when it skips the step of DDE and sets the initial value of $bsfDist$ in Algorithm 3 to positive infinity. Other design details of PDD, such as transmission among nodes and allocation of bulks, do not affect the correctness of PDD.

4 Empirical Evaluation

In this section we empirically evaluate the practical performance of PDD. We use randomly generated time series datasets for the following reasons: (1) Random time series is generally more challenging than real-world time series in terms of the time consumption of discord discovery methods [3, 7, 11, 12, 14, 15, 20]. The experimental results of time consumption on random time series are more convincing. (2) Large scale random time series is easier to acquire. For better reproducibility we choose random time series for our datasets.

We implement PDD using Apache Spark [18]. PDD makes use of the in-memory-computing feature of Spark to accelerate the detection of discord. We establish a Spark cluster consisting of 10 computing nodes, each of which is equipped with 512 MB memory. The time series are initially stored in Hadoop distributed file system [17], which is accessible to all computing nodes.

4.1 Scalability

PDD method can be scaled to discover discords. In this paper, we evaluate the scalability of PDD in terms of parallelism and data sizes, and compare the time consumptions between PDD and the classic method HOTSAX . For better comparability, we implement HOTSAX with Java and run HOTSAX on one of the computing nodes of the Spark cluster.

Table 2 shows the scalability of PDD. First we use PDD with different parallelism to detect the top discord from a 1×10^5 dataset. PDD achieves a speed-up ranging from 1.54 to 6.75 compared to HOTSAX (column 5 and row 2-6). Then we detect the top discord from the datasets of sizes ranging from 1×10^5 to 1×10^6 with 10 computing nodes of the Spark cluster. PDD achieves a speed-up ranging from 6.75 to 8.04 compared to HOTSAX (column 5 and row 6-11).

Table 2. Scalability of PDD

Data size	HOTSAX time	PDD time	#node	Speedup	Speedup/node
1×10^5	3374 s	2184 s	2	1.54	0.77
		1190 s	4	2.84	0.71
		761 s	6	4.43	0.74
		570 s	8	5.92	0.74
		500 s	10	6.75	0.68
2×10^5	4.0 h	2058 s		7.00	0.7
4×10^5	11 h	1.6 h		6.88	0.69
6×10^5	28 h	3.9 h		7.18	0.71
8×10^5	60 h	8.0 h		7.5	0.75
1×10^6	82 h	10.2 h		8.04	0.80
1×10^7	NA	111 h		NA	NA

The speed-up provided by each computing node indicates the scalability of PDD (column 6 of Table 2). On average, each additional computing node of PDD provides more than 0.7 times speed-up compared to HOTSAX. The speed-up per node is not sensitive to the size of the dataset or the parallelism of PDD.

We also perform experiments of PDD and HOTSAX on a 1×10^7 datasets. HOTSAX collapses because of the limitation of memory capacity of a single computing node. PDD successfully discovers the top discord with non-united memory spaces, which is 10 times bigger than that of one computing node. This experiment indicates PDD relieves the limitation of single computer memory in large scale time series discord discovery.

4.2 Utilization of computing resources

In this part we evaluate the utilization of computing resources of PDD. We detect the top discord from a 1×10^5 dataset with 10 computing nodes of the Spark cluster. Table 3 shows the utilization of computing resources of PDD and the disk-aware method [20].

In the first row, PDD allocates 10 data blocks per round (BPR), which means each computing node gets one data block. The idling time caused by load imbalance accounts for 18.9% of the total PDD time consumption. Along with the increment of BPR, idling time decreases. When BPR = 500, the idling time drops to 4.71% of the total time consumption, and the total time consumption of PDD also drops from 580s to 500s.

When BPR equals to 1000, the total time rises to 547 seconds. This is because computing nodes receive all possible subsequences in a single round such that early abandon technique has less effect on reducing computation complexity (line 8-11 in Algorithm 3). Empirically, BPR $\times$ (#subsequences/bulk) should be set no larger than one tenth of the total number of subsequences of a time series so that early abandon technique functions adequately.

We also use Computing time to Running time Ratio (CRR) in the last two columns of Table 3 to evaluate the utilization. The CRR of PDD is about 95%

Table 3. Utilization of the computational resources

BPR	#subsequences/bulk	Total time	Idle	CRR of PDD	CRR of Yankov [20]
10	200	580 s	18.9%	96%	$\leq 50\%$
100		527 s	6.25%	95%	
500		500 s	4.71%	95%	
1000		547 s	3.04%	94%	

in different conditions of BPR. Besides computing time, the running time also includes the time for backstage job of the Spark, such as scheduling, task deserialization, Java garbage collection, data transmission between computing nodes, etc. Compared with the disk-aware method [20], the utilization of computing resources of PDD is about twice more than that of [20].

5 Conclusion

Discords are subsequences that are maximally different to all the other subsequences of a time series. Existing methods of discord discovery are all sequential. They suffer from the limitation of computing power and storage of a single computing node. Also, because the results discovered from segmentations of a time series are non-combinable, discord discovery is hard to parallelize. In this paper, we propose Parallel Discord Discovery (PDD), which divides discord discovery in a combinable manner and solves its sub-problems in parallel. Experiments show that given 10 computing nodes, PDD is 7 times faster than the classical discord discovery method HOTSAX. PDD handles larger datasets, which cannot be handled by the memory based methods. Experiments indicate the computing time accounts for more than 90% of the execution time, and reduces the negative effects on performance caused by disk I/O operations.

Acknowledgments

This paper is sponsored by National Natural Science Foundation of China (No. 61373032), the National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) program and the National High Technology and Research Development Program of China (863 Program, 2015AA050204).

References

1. Ameen, J., Basha, R.: Higherrarchical data mining for unusual sub-sequence identifications in time series processes. In: Innovative Computing, Information and Control, 2007. ICICIC'07. Second International Conference on. pp. 177–177. IEEE (2007)

2. Basha, R., Ameen, J.: Unusual sub-sequence identifications in time series with periodicity. International Journal of Innovative Computing, Information and Control 3(2), 471–480 (2007)
3. Bu, Y., Leung, O.T.W., Fu, A.W.C., Keogh, E.J., Pei, J., Meshkin, S.: Wat: Finding top-k discords in time series database. In: SDM. pp. 449–454. SIAM (2007)
4. Buu, H.T.Q., Anh, D.T.: Time series discord discovery based on isax symbolic representation. In: Knowledge and Systems Engineering (KSE), 2011 Third International Conference on. pp. 11–18. IEEE (2011)
5. Camerra, A., Palpanas, T., Shieh, J., Keogh, E.: isax 2.0: Indexing and mining one billion time series. In: Data Mining (ICDM), 2010 IEEE 10th International Conference on. pp. 58–67 (Dec 2010)
6. Chiu, B., Keogh, E., Lonardi, S.: Probabilistic discovery of time series motifs. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 493–498. ACM (2003)
7. Fu, A.W.C., Leung, O.T.W., Keogh, E., Lin, J.: Finding time series discords based on haar transform. In: Advanced Data Mining and Applications, pp. 31–41. Springer (2006)
8. Fu, T.c.: A review on time series data mining. Engineering Applications of Artificial Intelligence 24(1), 164–181 (2011)
9. Huang, T., Zhu, Y., Wu, Y., Bressan, S., Dobbie, G.: Anomaly detection and identification scheme for vm live migration in cloud infrastructure. Future Generation Computer Systems (2015)
10. Jones, M., Nikovski, D., Imamura, M., Hirata, T.: Anomaly detection in real-valued multidimensional time series (2014)
11. Keogh, E., Lin, J., Fu, A.: Hot sax: Efficiently finding the most unusual time series subsequence. In: Data mining, fifth IEEE international conference on. pp. 8–pp. IEEE (2005)
12. Li, G., Bräysy, O., Jiang, L., Wu, Z., Wang, Y.: Finding time series discord based on bit representation clustering. Knowledge-Based Systems 54, 243–254 (2013)
13. Lin, J., Keogh, E., Fu, A., Van Herle, H.: Approximations to magic: Finding unusual medical time series. In: Computer-Based Medical Systems, 2005. Proceedings. 18th IEEE Symposium on. pp. 329–334. IEEE (2005)
14. Luo, W., Gallagher, M.: Faster and parameter-free discord search in quasi-periodic time series. In: Advances in Knowledge Discovery and Data Mining, pp. 135–148. Springer (2011)
15. Luo, W., Gallagher, M., Wiles, J.: Parameter-free search of time-series discord. Journal of computer science and technology 28(2), 300–310 (2013)
16. Miller, C., Nagy, Z., Schlueter, A.: Automated daily pattern filtering of measured building performance data. Automation in Construction 49, 1–17 (2015)
17. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. pp. 1–10. IEEE (2010)
18. Spark, A.: Apache spark–lightning-fast cluster computing (2014)
19. Wei, L., Keogh, E.J., Xi, X.: Saxually explicit images: Finding unusual shapes. In: ICDM. vol. 6, pp. 711–720 (2006)
20. Yankov, D., Keogh, E., Rebbapragada, U.: Disk aware discord discovery: finding unusual time series in terabyte sized datasets. Knowledge and Information Systems 17(2), 241–262 (2008)