Noname manuscript No.
(will be inserted by the editor)

ParCorr: Efficient Parallel Methods to Identify Similar Time Series Pairs across Sliding Windows

Djamel Edine Yagoubi · Reza Akbarinia · Boyan Kolev · Oleksandra Levchenko · Florent Masseglia · Patrick Valduriez · Dennis Shasha

the date of receipt and acceptance should be inserted later

Abstract Consider the problem of finding the highly correlated pairs of time series over a time window and then sliding that window to find the highly correlated pairs over successive co-temporous windows such that each successive window starts only a little time after the previous window. Doing this efficiently and in parallel could help in applications such as sensor fusion, financial trading, or communications network monitoring, to name a few. We have developed a parallel incremental random vector/sketching approach to this problem (as explained in section 4) and compared it with the state-of-the-art nearest neighbor method iSAX [7]. Whereas iSAX achieves 100% recall and precision for Euclidean distance, the sketching approach is, empirically, at least 10 times faster and achieves 95% recall and 100% precision on real and simulated data. For many applications this speedup is worth the minor reduction in recall. Our method scales up to 100 million time series and scales linearly in its expensive steps (but quadratic in the less expensive ones).

1 Introduction

An easy-to-understand motivating use case for finding sliding windows correlation comes from finance. In that application, the time series consist of prices of trades of different stocks (Figure 1). The problem is to find pairs of stocks whose return profiles look similar over the most recent time period (typically, a few seconds). A pair of time series (e.g. Google and Apple prices) whose returns were similar before and have since diverged, where say Google went up more than Apple, might present a trading

Djamel Edine Yagoubi · Reza Akbarinia · Boyan Kolev · Oleksandra Levchenko · Florent Masseglia · Patrick Valduriez
Inria & LIRMM, Montpellier, France
E-mail: djamel-edine.yagoubi@inria.fr, reza.akbarinia@inria.fr, boyan.kolev@inria.fr, oleksandra.levchenko@inria.fr, florent.masseglia@inria.fr, patrick.valduriez@inria.fr

Dennis Shasha
Department of Computer Science, New York University
E-mail: shasha@cs.nyu.edu

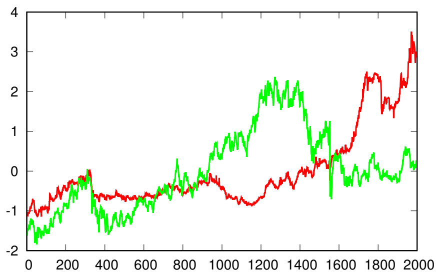

Fig. 1: A pair of normalized time series representing stock prices. The two series are highly correlated over the beginning of the period, after which a divergence can be observed.

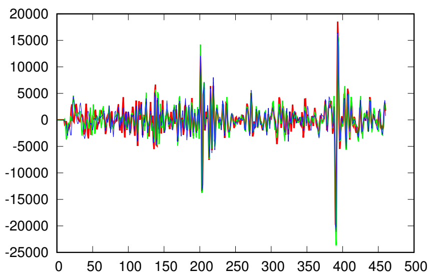

Fig. 2: Correlated signals from three different seismic sensors may suggest that they are all related to the same seismic event.

opportunity: sell the one that has gone up relative to the other and buy the other one. The return at t is the fractional change, $(price(t) - price(t-1))/price(t-1)$.[1]

While prices are stable over time (e.g. a stock whose price is 100 will tend to stay around 100), the returns resemble white noise. We call such time series "uncooperative", because standard dimensionality-reduction techniques such as Fourier or Wavelet Transforms either sacrifice too much accuracy or reduce the dimensionality too little [11]. Random sketch-based methods and some other explicit encoding methods work well for both cooperative and uncooperative time series. Moreover, the sketch-based methods work nearly as well as Fourier/Wavelet methods for cooperative time series. So, for the sake of generality, this paper uses the sketch method of Cole et al. [11], and compares the result with the state-of-the-art explicit encoding method iSAX [7].

The need for speed comes from the increasing number of data sensors available and the advantage of reacting quickly. An irony of improving technology is that sensor speeds and numbers increase substantially faster than computational speed. For this reason, linear or near linear-time algorithms become increasingly vital to give timely responses in the face of the flood of data. In some applications, speed may be of greater importance than completeness because the speed of response is of primary importance, so a minor loss in recall is often acceptable as long as precision is high. In trading, for example, there is only a fictitious monetary loss in missing an opportunity, but the opportunities a system reports should be real and must be timely to be actionable. For another example, consider a set of sensors spaced over several possible earthquake zones. Temporal correlations of pairs of sensors over a time window may suggest that these pairs are responding to the same seismic cause (Figure 2). Missing some correlations is acceptable, because a major event will reveal many correlations so a recall of 90% or more is sufficient.

In this paper, we propose ParCorr, an efficient parallel solution for detecting similar time series across sliding windows. ParCorr uses the sketch principle for representing the time series. Our ParCorr solution includes the following contributions:

[1] This is slightly simplified. Often, analysts consider the volume-weighted average price per time unit. So if there are 1000 shares traded at 100 and 1 million shares traded at 110 in a millisecond, then the volume-weighted average price during that millisecond is very close to 110. We compute the returns based on these volume-weighted prices.

- A simple parallel approach for the incremental computation of the sketches in sliding windows that also incorporates window normalization. This approach avoids the need for renormalizing the input and recomputing the sketches from scratch, after modifications in the content of the sliding window.
- A partitioning approach that projects sketch vectors of time series into subvectors and builds a distributed grid structure for the subvectors in parallel. Each subvector projection can be processed in parallel.
- An efficient algorithm for the parallel detection of correlated time series candidates from the distributed grids. In our algorithm, we minimize both the size and the number of messages needed for candidate detection.

Experiments on ParCorr in a distributed environment using real and synthetics datasets show both the high efficiency and almost linear scalability of the proposed solution compared to the state of the art.

The rest of the paper is organized as follows. In Section 2, we formally define the sliding window correlation problem, and in Section 3, we discuss related work. In Section 4, we propose our parallel solution for detecting correlated time series across sliding windows. Section 5 reports the results of our experimental evaluation, and Section 6 concludes.

2 Problem Definition

A time series ts is a sequence of values $ts = \{v_1, ..., v_m\}$. We assume that every time series has a value at every time point $p = 1, 2, ..., m$A streaming time series is a potentially unending series of values in time order. A data stream, for our purposes, is a set of streaming time series. Our method normalizes incrementally each time series window to have zero mean and unit standard deviation. Correlation over windows from the same or different series has many variants. This paper focuses on the synchronous variant (in which we are concerned with co-temporous windows), defined as follows:

Given a data stream of N_s streaming time series, a start time p_s, and a window size w, find, for each time window W of size w, all pairs of streaming time series ts_1 and ts_2 such that ts_1 during time window W is highly correlated (over 0.7 typically) with ts_2 during the same time window.

Euclidean distance is the target metric of the state-of-the-art iSAX algorithm. In addition, the Euclidean distance is related to Pearson correlation as follows:

$$D^2(\hat{x}, \hat{y}) = 2 \times m \times (1 - corr(x, y)) \tag{1}$$

Here $\hat{x}$ and $\hat{y}$ are obtained from the raw time series by computing $\hat{x} = \frac{x - avg(x)}{\sigma_x}$, where $\sigma_x = \sqrt{\sum_{i=1}^{m}(x_i - avg(x))^2}$. m is the length of the time series. So, we offer parallel algorithms for both sliding window Euclidean and correlation metrics in this paper.

3 Related Work Regarding Time Series Similarity

In this paper, we address the problem of all-pair parallel correlation detection across sliding windows of time series. The problem has been studied across data streams using centralized approaches [24,23,32,29,28,27,11,26]. Most of them focus on reducing the computation time of the pairwise distance computation. For example in [24], Mueen et al. propose efficient algorithms based on the Discrete Fourier Transformation (DFT), to reduce the end-to-end response time of an all-pair correlation query. As Cole et al. discuss in [11], this works well when the time series are cooperative (*i.e.,* where the low frequency Fourier coefficients dominate).

The problem of indexing and querying time series using centralized solutions has been widely studied in the literature, e.g. [4,5,12,31,7]. For instance, in [4], Assent et al. propose the TS-tree, an index structure for efficient retrieval and similarity search over time series. In [31], Shieh et al. propose the multiresolution symbolic representation called indexable Symbolic Aggregate approXimation (iSAX) [7] which is based on the SAX representation. The advantage of iSAX over SAX is that it allows the comparison of words with different cardinalities, and even different cardinalities within a single word. iSAX can be used to create efficient indices over very large databases.

In our work, we use an approach based on incremental random sketches [11]. In the literature, several techniques have been used to perform dimensionality reduction on the size of time series. Examples of such techniques that can significantly reduce the time and space required for the index are: singular value decomposition (SVD) [12], the discrete Fourier transformation (DFT) [3], the discrete wavelets transformation (DWT) [9], the piecewise aggregate approximation (PAA) [21], and the adaptive piecewise constant approximation (APCA) [8]. We compare with iSAX [7] because it does not require time series to be cooperative (though it is more efficient when the time series is slowly changing).

Recently, the matrix profile structure [34,37,33,36] has been proposed for all-pair similarity detection in very long time series. Given a subsequence of size m, and two time series A and B, the problem is to find for each subsequence of size m in A the most similar (correlated) subsequence in the time series B. The authors proposed efficient algorithms for addressing this problem in a single processor and also on GPU-equiped machines. However, our problem differs because we are interested in similarity detection over windows of millions of time series, not on subsequences of two very long time series. Quantitatively, the matrix profile SCRIMP algorithm[2] takes 10 seconds for the motifs on a time series of size one million to converge (which it does after only 0.1% of its full execution time). Thus, we may find approximate nearest neighbors across all subsequences (almost one million), where each subsequence is of size 500. If we were to apply the matrix profile approach for the frame of one window in our setting (one million series, each of size 500), that would require concatenating all series into a single very long one (of size 500 million) and making million similarity searches against it. To do that, the MASS algorithm [25] cleverly makes use of convolutions to do one similarity search in $O(n \times \log n)$ time, or around 80 seconds for our volume of data on quite powerful hardware. But even if we assume perfect parallelization and convergence at 0.1% of the full execution time, that would take hours to complete. So, while the matrix profile techniques are intriguing, they solve a different problem and applying those to our problem would result in a slower algorithm.

An interesting method for clustering and segmenting multivariate time series data was recently presented by Hallac et al. in [17]. The method analyses a set of streaming variables - typically tens of observations on different features of the same object. Sliding windows of some size w are then assigned to clusters. Each cluster is characterized by a

[2] http://www.cs.ucr.edu/~eamonn/MatrixProfile.html

correlation network that encodes the interdependencies between different observations. The goal is to achieve a segmentation that is longer than w to find repeated long-range patterns in the data that represent particular behaviors of the object. This method efficiently exploits correlations across several time series observations; however, the problem it addresses is quite different from our objective, where we want to discover high correlations across millions of parallel time series.

In [16], Guo et al. propose an approach that uses a δ–Hypercube structure for correlation discovery over streaming time series. Let w be the size of the sliding window, the proposed approach creates w-dimensional orthogonal regular hypercubes, where each hypercube keeps a pointer to each of its neighbors. When constructed, the hypercubes allow the fast discovery of similar time series, since when two time series are similar, their hypercubes will be close. The authors propose a naive algorithm in which the number of neighboring hypercubes is exponential in the size, w, of the sliding window (*i.e.,* increases faster than 2^w). That clearly does not scale. However they then propose a series of clever grouping and centroid-style approaches to prune the search, resulting in an algorithm called AEGIS. The complexity analysis they present is understandably qualitative (section 5.1 of their paper), because the performance depends on how well the data grouping works. When they compare AEGIS with the naive method they get a factor of 30 improvement over the naive method in communication cost and about 1.5 in time. They run their experiments on up to 36,000 time series on which AEGIS takes 3 to 10 seconds. Our algorithm on similar clustering hardware scales out to 100 million time series taking under 2000 seconds, thus yielding a lower cost per time series. So, we do not compare experimentally with AEGIS, though we think some of the grouping ideas of AEGIS could be an avenue for future work.

Sometimes one wants to find clusters of unusual events in time series, e.g. bursts of activity or bursts of unusually high values. In such settings, tiling algorithms [15, 18, 13] apply. Our problem is complementary because we are finding correlations between time series or portions of time series, but are agnostic to the level of interest of individual time points. However, our method could be used to post-process temporal regions that tiling indicates are of interest.

To the best of our knowledge, in the literature there is no solution for parallel correlation detection over sliding windows of streams containing many millions of continuous time series.

4 Algorithmic Approach

Following the method of Cole et al. [11], our basic approach to find similar pairs of sliding windows in time series (whether Euclidean distance or Pearson correlation, for starters) is to compute the dot product of each normalized time series over a window size w with a set of random vectors. That is, for each time series t_i and window size w and time period $k..(k + w - 1)$, we compute the dot product of $t_i[k..k + w - 1]$ with r random $(+1| - 1)$ vectors of size w. The r dot products thus computed constitute the "sketch" of t_i at time period $k..(k + w - 1)$. Next we compare the sketches of the various time series to see which ones are close in the sketch space (if $w >> r$, which is often the case, this is cheaper than working directly on the time series) and then identify the close ones.

The theoretical underpinning of the use of sketches is given by the Johnson-Lindenstrauss lemma [20].

Lemma 1 *Given a collection C of n time series each of length w, for any two time series $\overrightarrow{x}, \overrightarrow{y} \in C$, if $\epsilon < 1/2$ and $r = \frac{8 \log n}{\epsilon^2}$, then*

$$(1-\epsilon) \leq \frac{\| \overrightarrow{s}(\overrightarrow{x}) - \overrightarrow{s}(\overrightarrow{y}) \|^2}{\| \overrightarrow{x} - \overrightarrow{y} \|^2} \leq (1+\epsilon)$$

holds with probability $1/2$, where $\overrightarrow{s}(\overrightarrow{x})$ is the sketch of $\overrightarrow{x}$ of at least r dimensions.

The Johnson-Lindenstrauss lemma implies that the distance $\|\mathbf{sketch}(\mathbf{t_i}) - \mathbf{sketch}(\mathbf{t_j})\|$ is a good appproximation of $\|\mathbf{t_i} - \mathbf{t_j}\|$ provided the dimensionality of the sketches (r) is large enough. Specifically, if $\|\mathbf{sketch}(\mathbf{t_i}) - \mathbf{sketch}(\mathbf{t_j})\| < \|\mathbf{sketch}(\mathbf{t_k}) - \mathbf{sketch}(\mathbf{t_m})\|$, then it's likely that $\|\mathbf{t_i} - \mathbf{t_j}\| < \|\mathbf{t_k} - \mathbf{t_m}\|$, because the ratio between the sketch distance and the real distance is close to one.

The sketch approach, as developed by Kushilevitz et al. [22], Indyk et al. [19], and Achlioptas [2] makes use of these guarantees. Note that the sketch approach is closely related to Locality Sensitive Hashing [14], by which similar items are hashed to the same buckets with high probability. In particular, the sketch approach is very similar in spirit to SimHash [10], in which the vectors of data items are hashed based on their angles with random vectors. The major contribution of our paper consists of combining an incremental strategy with a parallel mixing algorithm and an efficient communication strategy.

4.1 The case of sliding windows

In the case of sliding windows, we want to find the most similar time series pairs at jumps of a basic window b, e.g. for windows in time ranges 0 to $w-1$ seconds, b to $b+w-1$ seconds, $2b..2b+w-1$, ... where $b << w$.

There are two main challenges:

1. If we compute the sketches from scratch at each basic window, we are doing some redundant computation. For that reason, we call the recompute-from-scratch approach the naive method. It would be better to compute the sketches incrementally and in parallel. Moreover, if we normalize the entire window at each slide, that would destroy the advantage of incemental update. Instead, we want to enhance the sketch computation to yield values as if computed on normalized data.

2. When scaling this to a parallel system, we want to reduce communication costs as much as possible. We need to develop good strategies for this as communication is quadratic in the number of execution nodes, so constant coefficients matter.

4.2 Parallel incremental computation of sketches

To explain the incremental algorithm consider the example of Figure 3. The sketch for random vector v_1 is the dot product of the time series with v_1, *i.e.,* $1 \times (-1) + 2 \times 1 + ... + 4 \times (-1) = 4$.

Now if the basic window is of size 2, as illustrated by the “outdated” and “incoming” boxes of Figure 3, then we add the next two values of the time series. In our example,

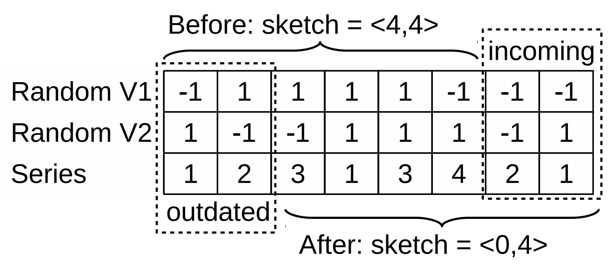

Fig. 3: A streaming time series, two random vectors, and the sketches that correspond to their dot product before and after the update on the data stream. The first sketch of the time series is computed on the six first values, and the second sketch is computed on the six last values. After the update, the "outdated" values are removed and the "incoming" ones are added to the streaming time series, so the work is proportional to the size of the basic window rather than the full window.

we add values $(2, 1)$ and generate two more random $+$/- 1 numbers for each random vector, in this case $(-1, -1)$ for v_1 and $(-1, 1)$ for v_2. To update the dot product for v_1 we subtract the contribution of the oldest two time points, viz. $1 \times (-1) + 2 \times 1 = 1$, and add the contribution of $2 \times (-1) + (1 \times -1) = -3$ yielding a new sketch entry of $4 - 1 + (-3) = 0$.

In addition, we maintain and incrementally update the statistics (mean and standard deviation) of the current window. We then incorporate these statistics directly into the incremental sketch updating as described below, so that the resulting sketch values have the same values as if computed on normalized data. This enhancement allows keeping the input data as is and avoids expensive renormalization as new data arrives.

In general, the algorithm proceeds as following:

1. Partition time series among parallel sites. Replicate r random $+1$|-1 vectors each of size w to all sites. These random vectors will later be updated in a replicated fashion.
2. For each site,
   (a) Initially, take the first w data points of each time series at that site and form the dot product with all r random vectors. So each time series t will be represented by r dot products. They constitute $sketch(t)$.
   (b) When data for the i_{th} basic window of size b appears for all time series, extend each random vector by a new random $+1$|-1 vector of size b. Then for each time series t and random vector v, update the dot product s of t with v by:
   $\Delta s = -v[0..b-1] \cdot t[(i-1)b - w...ib - w - 1] + v[w..w + b - 1] \cdot t[(i-1)b..ib - 1]$
   (c) To apply window normalization in the same step, we consider the means (μ_i and μ_{i-1}) and standard deviations (σ_i and σ_{i-1}) of the current and previous windows, whose values are maintained incrementally. Then we use the following equation to update the current value (s_i) of the dot product of t with v with respect to the previous one (s_{i-1}):

$\sigma_i \times s_i = \sigma_{i-1} \times s_{i-1} + \Delta s + \mu_{i-1} \times \Sigma v[0..w-1] - \mu_i \times \Sigma v[b..w+b-1]$
This works because the sketches for the i_{th} window should be computed on $(t - \mu_i)/\sigma_i$ instead of on t. Note that the two sums in this equation are on random +1|-1 vectors of size w, which can be computed just once per window and reused to update the sketch of all time series. Thus, this enhanced formula adds normalization to the incremental sketch updating at almost no additional cost.

(d) Change the $sketch(t)$ with all the updated dot products.

This step has time complexity proportional to the number of time series × size of basic windows × the number of random vectors. It is perfectly parallelizable.

Step 1 calls for parallel updates of the local random vectors on each site. It is mandatory that all the sites share the same random vectors. A possible approach would be for the master node, after the completion of each new sliding window, to generate new +1|-1 vectors having the basic window size, and send them to the sites. This takes little time but is awkward to do in Spark, our implementation platform. Our (admittedly hackish) approach is therefore to generate and send oversized random vectors (say, twice the size of the sliding window) at setup time. A site then just has to loop inside the (oversized) random vector, simulating an endless source of +1|-1 values that are the same for all the sites. This deviation from perfect randomness has no effect on the quality of the results in our experiments, though we would prefer the approach where the master node generates new random vectors.

4.3 Parallel mixing

Once the sketch vectors have been constructed incrementally, the next step is to find sketch vector pairs that are close to one another. Such pairs might then indicate that their corresponding time series are highly correlated.

Multi-dimensional search structures do not work well for more than four dimensions in practice [30]. For this reason, as indicated in the following example, we adopt a framework that partitions each sketch vector into subvectors and builds grid structures for the subvectors.

We first describe the steps of the algorithm and then explain how it works by an example.

Algorithm:

1. Partition the sketch vectors, which all have length r, into groups of size k (e.g. if r is 60 and k is 2, then the partition would be 0,1, 2,3, 4,5, ..., 58,59 and we would take indexes 0 and 1 of each sketch vector and put it in the first partition. So each partition would consist of n size 2 mini-vectors, where n is the number of time series.).
2. Each site computes a grid and puts time series identifiers in grid cell (Table 2). So, for each site s,
   (a) for each time series t, place the identifier of t in a grid cell corresponding to sketch(t)$[I_s]$, where I_s are the indexes assigned to site s.
   (b) Next form a partition of the time series identifiers such that each member of the partition corresponds to a non-empty grid cell. So, two time series t_1 and t_2 will fall into the same partition if sketch(t_1)$[I_s]$ maps to the same grid cell

as sketch(t2)[I_s]. Denote the partitioning induced by this grid search on site s as partitioning(s).

(c) Each element of the partition p in partitioning(s) represents a set of time series. If we sort them by their id, then p can represent $ts\_p\_1, ts\_p\_2, ...$

3. If two time series are in the same grid cells in a fraction f of the grids, then they are considered "close". (The parameter f is determined by a calibration step that in turn depends on the desired correlation threshold, as we will explain in the experimental section.) We start by constructing "candidate clusters of time series" based on each grid.
4. Send each candidate cluster of time series identifier to every node corresponding to the time series in that cluster (Table 3). Call the mapping function between time series ids and nodes ts_to_node(), to be defined as the "opt" strategy in the next subsection.[3]
5. At each destination node, two time series are candidates for explicit analysis if they are "close", *i.e.,* they are in the same grid cell for some fraction f of the grids (Table 3). If so, compute the Pearson correlation on all such candidates.

Example 1. Suppose we have seven time series with sketch values as follows:

sketch(ts_1) = (11, 12, 23, 24, 15, 16)
sketch(ts_2) = (11, 12, 13, 14, 15, 16)
sketch(ts_3) = (21, 22, 13, 14, 25, 26)
sketch(ts_4) = (21, 22, 13, 14, 25, 26)
sketch(ts_5) = (11, 12, 33, 34, 25, 26)
sketch(ts_6) = (31, 32, 33, 34, 15, 16)
sketch(ts_7) = (21, 22, 33, 34, 15, 16)

First, we partition these into pairs and send the 1st and 2nd values of each sketch vector to site 1 (Table 1) where this will be formed into a grid (and the time series identifiers will be placed in cells (i,j), e.g., (31,32)). Analogously, we send the 3rd and 4th values of each sketch to site 2, and the 5th and 6th values to site 3, where the second and third grids will be formed. In the first grid, ts_1, ts_2, and ts_5 map to the same grid cell; ts6 is by itself; and ts_3, ts_4, and ts_7 all map to the same cell (Table 2). Thus, in grid 1 we have three partitions of time series identifiers. If two time series are in the same partition, then they are close, so they are candidates for a detailed correlation calculation.

Now, we construct a mapping ts_to_node that maps time series identifiers to nodes (for now, think of each node as a single computational site, but one could imagine placing many nodes on a single site or spreading a node among many sites). For this example, let us say ts_to_node is the identity function. So we send the relevant parts of the partition ts_1, ts_2, ts_5 to nodes 1, 2, and 5. Similarly, we send the relevant parts of ts_3, ts_4, and ts_7 to nodes 3, 4, and 7. And so on (see Table 3). Assuming the "opt" communication strategy (see next subsection), the relevant part of a partition with respect to a time series t consists of t itself and the time series with identifiers higher than t. We call that a "candidate cluster of time series". We ignore clusters with just one element, as pairs cannot be derived out of them.

[3] For this discussion, we assume that ts_to_node is 1 to 1. If not, then if a node has say the time series groups corresponding to i_1, i_2, and i_3, then keep those groups separate.

Table 1: Step 1 of the algorithm: sketch partitioning. Each sketch vector is partitioned into three pairs. The ith pair of the sketch vector for each time series s goes to a grid i. The values of the ith pair determine where in that grid the identifer s is placed.

	sketch subvectors		
	[0 1]	**[2 3]**	**[4 5]**
sketch(ts_1)	(11, 12)	(23, 24)	(15, 16)
sketch(ts_2)	(11, 12)	(13, 14)	(15, 16)
sketch(ts_3)	(21, 22)	(13, 14)	(25, 26)
sketch(ts_4)	(21, 22)	(13, 14)	(25, 26)
sketch(ts_5)	(11, 12)	(33, 34)	(25, 26)
sketch(ts_6)	(31, 32)	(33, 34)	(15, 16)
sketch(ts_7)	(21, 22)	(33, 34)	(15, 16)
	assigned to grid / at site		
	1	**2**	**3**

Table 2: Step 2 of the algorithm: grid construction. Time series placed in the same grid cells are grouped in partitions.

grid	**cell**	**time series IDs**
1	(11, 12)	ts_1, ts_2, ts_5
	(21, 22)	ts_3, ts_4, ts_7
	(31, 32)	ts_6
2	(13, 14)	ts_2, ts_3, ts_4
	(23, 24)	ts_1
	(33, 34)	ts_5, ts_6, ts_7
3	(15, 16)	ts_1, ts_2, ts_6, ts_7
	(25, 26)	ts_3, ts_4, ts_5

Table 3: Steps 4 and 5 of the algorithm: finding frequently collocated pairs (in the example, at least 2 out of 3 grids).

node	**TS clusters**	**candidate pairs $f \geq 2/3$**
ts_to_node(ts_1)	ts_1, ts_2, ts_5 ts_1, ts_2, ts_6, ts_7	ts_1, ts_2
ts_to_node(ts_2)	ts_2, ts_5 ts_2, ts_3, ts_4 ts_2, ts_6, ts_7	
ts_to_node(ts_3)	ts_3, ts_4, ts_7 ts_3, ts_4 ts_3, ts_4, ts_5	ts_3, ts_4
ts_to_node(ts_4)	ts_4, ts_7 ts_4, ts_5	
ts_to_node(ts_5)	ts_5, ts_6, ts_7	
ts_to_node(ts_6)	ts_6, ts_7 ts_6, ts_7	ts_6, ts_7

As mentioned above, we require that some fraction f of the grids should put two time series in the same grid cell for us to be willing to consider that pair of time series to be worth checking in detail. For this example, set f to 2/3 (Figure 4).

Each node takes care of those time series that map to that node. So for example, node 1 shows that ts_1 and ts_2 satisfy the requirement. Node 2 shows nothing new concerning ts_2. Node 3 shows that ts_3 and ts_4 satisfy the requirement. Node 4 and node 5 show nothing new concerning ts_4 and ts_5 respectively. Node 6 shows that ts_6

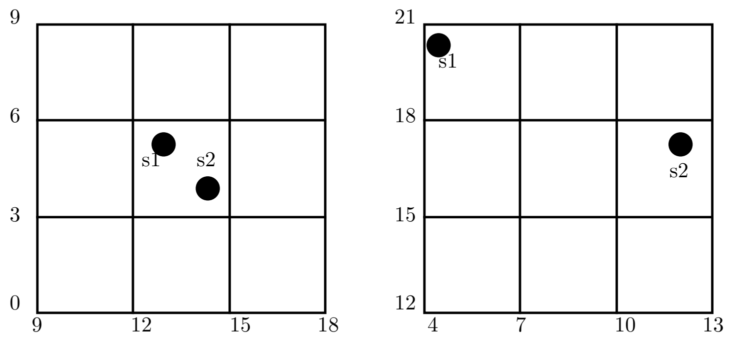

Fig. 4: Two series (s_1 and s_2) may be similar in some dimensions (here, illustrated by $Grid_1$) and dissimilar in other dimensions ($Grid_2$). If the series are close in a large fraction of the grids, they are likely to be similar. So, if that fraction exceeds some threshold f (2/3 in the toy example), then the algorithm performs an explicit correlation calculation.

and ts_7 satisfy the requirement. Node 7 shows nothing new (in fact the last node will never show anything new, so it does not need to be considered). All those that satisfy the requirement can be tested for direct correlation. In this example, that would entail computing correlations on the last windows of length w of ts_1 and ts_2; ts_3 and ts_4; and ts_6 and ts_7 (Table 3).

4.4 Communication strategies for detecting correlated candidates

Step 4 of the above algorithm requires the communication of information about each pair (t_i, tj) to one node of the system where its *grid score* (*i.e.,* , the number of grids in which the two time series are in the same cell) is computed. This communication may be done using different strategies, which in turn can have a large impact on the performance of our approach. This should come as no surprise: parallel approaches often require an optimization of communication. We compare three strategies for communicating the pairs of each grid cell:

– **All pairs communication (basic)**: In this strategy, for each cell c that contains $|count(c)|$ time series, all pairs (t_i, t_j) are generated and sent to a reducer whose address is based on i and j. This ensures that all information about a pair will be sent to one reducer where its grid score can be compared with threshold f. This is the straightforward approach and will be denoted as "basic" in the rest of this paper.
– **All time series to each responsible reducer (semi-opt)**: In this strategy, for each time series t there is a reducer r_t that is responsible for detecting the candidate time series that are correlated to t. Given a grid cell c, for each time series identifier $t \in contents(c)$, all time series identifiers of $contents(c)$ are sent to r_t. If, among the time series that reducer r_t receives, the number of occurrences of a time series t' is more than the threshold f, then the pair (t, t') is considered

to be close and therefore will undergo detailed correlation analysis. This is the semi-optimized (semi-opt in the rest of this paper) strategy.

– **Part of time series to each responsible reducer (opt)**: In this strategy (embodied in step 4 of the algorithm of the previous sub-section), as in the previous strategy, for each time series t there is a reducer r_t that is responsible for detecting the time series that are potentially correlated to t. But here, only some of time series of the cell are sent to r_t. Let's assume a total order on the ids of the time series, say the identifiers are related as follows: $t_1 < t_2 < \ldots < t_n$. Given a grid cell $contents(c) = \{t_1, \ldots, t_s\}$, for each time series identifier $t \in contents(c)$, the time series with ids greater than that of t are sent to r_t. The idea behind this strategy is that for a potential candidate pair (t_i, t_j), we need only to count its occurrences in the reducer corresponding to the lower identifier of the pair, not in both of them. As we will see below, this strategy requires the least amount of communication and is denoted "opt" in the rest of this paper.

To illustrate the different strategies in the context of our example, let us consider cell (15, 16) of grid 3 (see Table 2), which contains four time series identifiers. To handle its contents, the "basic" strategy would emit all the 6 pairs, *i.e.,* 12 ids in 6 messages in total. The "opt" strategy, by contrast, would emit a total of 9 ids in three messages, as per step 4 of the algorithm (Table 3). Thus, the "opt" uses less communication, but the "basic" strategy allows pairs to be emitted as soon as a new time series is assigned to a grid cell. Thus, the "basic" strategy waits less at the cost of more data exchange.

Generalizing from the example, we analyze the communication cost of the three strategies in terms of the size and the number of messages to be communicated for each cell. In the "basic" strategy, for the contents of each cell $contents(c) = \{t_1, \ldots, t_s\}$, all pairs (t_i, t_j) are generated and sent to the reducers. Thus, the number of messages for the cell c is equal to $count(c) \times (count(c) - 1)/2$. The size of each message is 2, so the size of data transferred for cell c is $count(c) \times (count(c) - 1)$. Note that in a distributed system, the number of messages is the principal contributor to communication cost of the algorithms. Thus, this approach suffers from a high communication cost, as the number of messages for a cell c is $O(count(c)^2)$.

In the "semi-opt" strategy, for each cell c, the node containing each grid communicates $(count(c) - 1)$ time series to the node that must compute the grid score. This means that the number of sent messages is $count(c)$, and the total size of the communicated data is $count(c) \times count(c) - 1$ time series ids per grid cell. In this strategy the number of messages is $O(count(c))$, which is much better than the basic strategy.

In the "opt" strategy, for each cell $contents(c) = \{t_1, \ldots, t_s\}$, we communicate $count(c)$ messages per grid cell, *i.e.,* one message to each node that is responsible for a time series in $contents(c)$. The size of the message depends on the id of the time series. Let $t_1, \ldots, t_s$ be the order of the time series ids. Then, we send $\{t_2, \ldots, t_s\}$ to r_{t_1}, $\{t_3, \ldots, t_s\}$ to r_{t_2}, etc. Therefore the total size of communicated data for cell c is $(count(c) - 1) + (count(c) - 2) + \ldots + 1 = count(c) \times (count(c) - 1)/2$. This strategy sends the same number of messages as "semi-opt" (*i.e.,* $O(count(c))$) for each cell, but the size of communicated data is smaller. Our experiments illustrate the benefits of this reduction in size.

4.5 Complexity analysis of parallel mixing

Let us analyze the time and space needed by our approach to perform parallel mixing.

The time of Step 1 is proportional to the number of time series times the number of random vectors (because the number of random vectors equals the size of each sketch vector). Note that it is independent of the size of the window. This step is completely parallelizable (because there is no communication cost) at the level of nodes and linear in the number of time series. Step 2 (inserting into grids) is also linear in the number of time series times the number of random vectors, but is cheaper than Step 1 because of smaller constant factors. The communication cost is minimal, because the sketches are so much smaller than the size of the windows.

The dominant time of our approach is that of Steps 3 and 4 in which the responsible node of each grid constructs the candidate clusters of time series, and sends them to the corresponding node based on ts_to_node. If ts_to_node is many to one, then even in the worst case the number of messages is proportional to the number of destination nodes and the total message traffic from a node is $O(n^2 \times idsize)$, where n is the number of time series and $idsize$ is the size of an id. That is a very pessimistic worst case because it corresponds to all time series mapping to the same grid cell in every grid. As we will see in the experimental section, the total traffic per node is linear in the number of time series in practice. Because time series ids are under 32 bits, the total traffic is light.

The last step, *i.e.,* 5, is proportional to the size of the output, because a large fraction of pairs that pass the sketch-filtering step in fact meet the correlation threshold. We will show this when we discuss the precision and recall of the approach.

The bulk of the space required for our approach is the space needed for keeping the grids for indexing the sliding windows. This space depends on the number of grids and the number of time series. The number of grids itself depends on the size of the sketches in the sliding window, and the group size (number of dimensions in each grid). Let g be the group size, s be the sketch vector size, and n the number of time series. Then, the number of grids required for indexing the sliding window data is $\frac{s}{g}$, where s is the sketch vector size and g the group size. In each grid, we need to keep the id of each time series in its corresponding cell. Thus, the total space required for storing the grids is $O(n \times idsize \times \frac{s}{g})$, where $idsize$ is the size of an id. Notice that in practice, the size of our grid-based index is much less than the space required for keeping the time series in the sliding windows. For example, suppose the group size is 2, and the sketch vector size is 32 (for a sliding window of size 256). Then, the space required to store all grids is equivalent to $16 \times n$ identifiers, which is less than the space needed to store n time series of size 256.

So, in practice, the entire procedure requires work that is the sum of (i) (formation of sketch vectors): $n \times b \times r$, where n is the number of time series, b is the basic window size, and r is the number of random vectors; (ii) (parallel mixing, grid computation): $n \times r$; (iii) (parallel mixing, candidate identification) for each grid cell, $nc^2 \times idsize$, where nc is the number of time series assigned to the grid cell and $idsize$ is the size of an id; (iv) (for verification of candidate pairs): $cands \times w$, where w is the sliding window size and $cands$ is the number of frequently collocated pairs. This work, except for the communication step (which depends on the communication infrastructure), is entirely parallelizable. Which term dominates depends on how high the correlation threshold is. For very high thresholds (the user is interested only in highly correlated pairs), part iv will be negligible, iii will be small, and so i and ii will dominate. If the threshold is low (not normally an interesting case) the algorithm could be nearly as expensive as comparing every pair of time series.

5 Experiments

In this section, we report experimental results that show the quality and the performance of our parallel incremental sketching approach, illustrating performance, scalability, recall, and precision. We compare our work with iSAX and show vastly improved speed at some cost in recall.

The parallel experimental evaluation was conducted on a cluster of 32 machines, with operating system Linux x86_64 kernel 3.10.0, each machine having 64 Gigabytes of main memory, an Intel Xeon CPU with 8 cores and a 256 Gigabytes hard disk.

We implemented the approaches on top of Apache-Spark 1.6.2 [35], using the Java programming language. [4]

Data streams are simulated by distributing the data beforehand and using synchronized sliding windows on each site. This setup allowed us to better evaluate the performance gains of our approach without depending on the specific characteristics or optimization of any dedicated streaming environment (e.g. Spark streaming, Flink, Storm, etc.).

5.1 Comparisons

We compare ParCorr to:

– **Parallel Linear Search**. This is the straightforward comparison that compares each time series to all the other ones, computing a Pearson correlation. In a second step, this approach sorts correlations in decreasing order and keeping the top-correlated ones. The approach is implemented in parallel (each computing node compares the series it contains to all the series of the other nodes).
– **iSAX** [7]. This index allows processing similarity queries using both an exact and an approximate approach. iSAX shows an improvement over Parallel Linear Search: when a computing node receives a time series to be compared to its local time series, rather than applying a linear search it will use a local iSAX index as a filter to identify the most similar time series.

In the data stream context, these algorithms are applied from scratch, after each update (each basic window-sized move of the sliding window). For iSAX, the local indexes have to be built again after each update.

5.2 Datasets

We carried out our experiments on synthetic, seismic and financial datasets.

Synthetic dataset: Each time series in our synthetic dataset consists of 2000 values. At each time point, the generator draws a random number from a Gaussian distribution N(0,1), then adds the value of the last number to the new number. The number of time series varies from one million to 100 million depending on the experiment. This type of random walk generator has been widely used in the past [3,12,4,31,6,7,38].

Seismic dataset: The real world data represents seismic time series collected from the IRIS Seismic Data Access repository [1] at various earthquake zones. After preprocessing, the seismic dataset contains 5 million time series of 2000 values each.

[4] Code and datasets available for free download at `http://parcorr.gforge.inria.fr`

To detect seismic events, there are three main types of algorithms: energy detectors, array detectors and matched filter detectors. The latter is a new kind of detector, where a representative time series is used as a template (*i.e.,* a "matched filter") and correlated against a continuous data stream to detect new occurrences of that same signal. However, such filters require a large number of templates, making indexing an appealing approach. A time series at a given sensor functions like a geophysical fingerprint for earthquakes. A seismic signal that closely matches a previous observation can be used as evidence that the newly observed event must have occurred very close to the event that generated the first observation. Moreover, if the signals are similar we can assume that the characteristics of the earthquakes are similar. There are many examples where almost identical signals produced by different earthquakes have been observed. This is typically the case during seismic crises that can last days or months, while similar signals can be recorded even if years apart. Detecting such correlations is a small variant of our problem, where all time series are compared with a few templates. Here we address the harder problem of finding all correlations among the set of time series. This might be useful in an application in which we want to detect, in a real time fashion, where similar seismic events are occurring.

Financial dataset: The bulk historical finance data was downloaded using the Yahoo Finance API[5] for over 40000 stock symbols for the period from Jan 2010 to Mar 2018. After preprocessing, the financial dataset contains 2000 price returns on end-of-day quotes for each stock symbol.

5.3 Parameters

Table 4 shows the default parameters used for each experiment, unless otherwise specified. A typical application might have a large ratio between the sliding window size and the basic window size, where the basic window indicates the time interval between the recalculation of similarity. ParCorr does better relative to the other algorithms with a smaller basic window size of 10 for example, but a larger number like 50 is more reasonable for high frequency measurements, where recomputing statistics would entail too much overhead. The iSAX word length, leaf capacity, basic cardinality, and maximum cardinality were chosen to be optimal for iSAX [7]. All histograms in the figures have error bars (usually so small as to be invisible) that go from a minimum value to a maximum value with the histogram height representing the mean. These statistics were taken from runs on all the 76 windows (the number of sliding windows of size 500 with a step of 20 over a total of 2000 values).

We calibrate the fraction f (needed for detecting candidate items in the grids) by using a small sample database. We increase f until reaching the desired recall (*e.g.*, 0.95) on the small sample, and then we use the found fraction (0.7 in the case) in our experiments on big datasets.

5.4 Recall and Precision Measures

To understand these concepts in our applications, consider the correlation problem: we want to find all pairs of time series that have at least a correlation of some specified

[5] http://finance.yahoo.com

Table 4: Default parameters

Parameters	Value
Sliding Window Size	500
Basic Window Size	20
iSAX Word Length	8
Leaf Capacity Threshold	1,000
Basic Cardinality	2
Maximum Cardinality	512
Number of Machines	32
Correlation Threshold	0.7
Fraction of Grids f	0.7

threshold during a given window. Call that set S_{true} In that context, the *recall* of a method that finds a set S_{method} is $|(S_{true} \cap S_{method})|/|S_{true}|$ and the *precision* is $|(S_{true} \cap S_{method})|/|S_{method}|$. This would also be true for Euclidean distances. These are completely standard uses of these terms applied to pairs and similarity metrics.

We achieve 100% precision by explicitly computing (step 5 of the algorithm) the actual correlation of each candidate pair extracted from the grids (steps 1-4). For this reason, we tune the parameters of the method to achieve high recall, sometimes at the cost of very low precision of the candidates at step 4. Even a candidate precision of less than 1% is acceptable, because explicitly verifying 100 times the number of actually correlated pairs is still orders of magnitude faster than examining all pairs. As this tuning is done for a particular correlation threshold, the candidate precision varies. In our experiments it ranges from 1.6% (for correlation threshold of 0.7) to 9.8% (for correlation threshold of 0.9).

In our experiments, the default correlation threshold for Pearson is 0.7. We have also tried 0.8 and 0.9. With a Pearson threshold of 0.8, the sketch recall was over 96% and the speedup compared with iSAX was a factor of 17.56. With a Pearson threshold of 0.9, the sketch recall was over 95.7% and the speedup compared with iSAX was a factor of 18. Given any Pearson correlation, the threshold for Euclidean distance is computed by using formula 1.

5.5 Output Examples

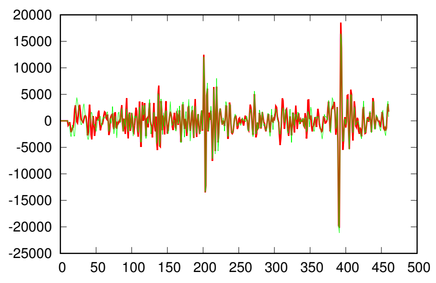

Fig. 5: Correlated pair of seismic sensor signals detected by our method.

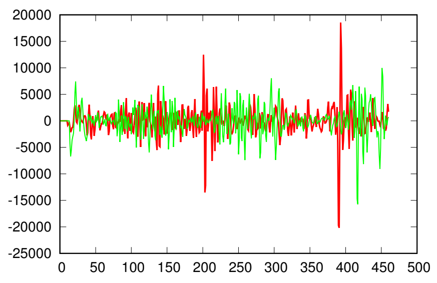

Fig. 6: Example of two uncorrelated seismic sensor signals.

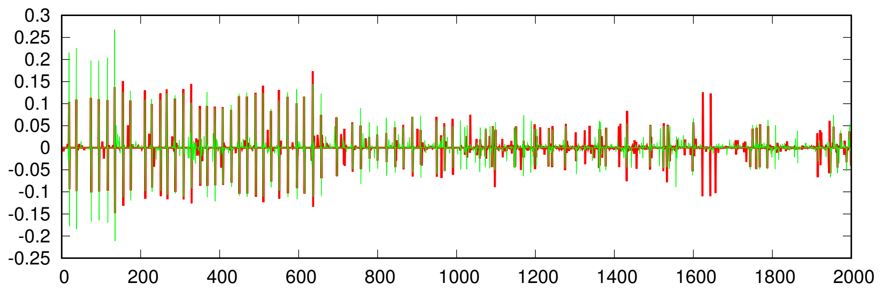

Fig. 7: Return profiles of two data stocks for 2000 dates. In this example, price returns are highly correlated over the first several windows of 500 values, but not thereafter.

Here we present examples of useful findings of our method on the datasets from the two real use cases. For the seismic use case, our algorithm discovered that the two sensors in Figure 5 are highly correlated, but made no such assertion about the ones in Figure 6. For the finance use case, as we apply our method on sliding windows of length 500 over a longer period, we discover a correlation of price returns of the two stocks on Figure 7 over the first several windows. For subsequent windows, this correlation drops, which can be interpreted by the application as a trading opportunity.

5.6 Communication Strategies

Before presenting the results of our approach in detail, we evaluate here the impact of the communication strategy to detect correlated pairs. This corresponds to the discussion and analysis given in Section 4.4. We conducted this experiment on 5 million time series, with a basic window of 32 and a sliding window of 256. As expected and illustrated by Figure 8, our optimized strategy gives the best performance (response time), but the size of the gain is surprising. Therefore, in the experiments presented below, we use this optimized strategy.

5.7 Results

Figure 9 shows that ParCorr is orders of magnitude faster for parallel correlation than the iSAX methods for the random walk dataset, though its time increases as the basic window size increases. For instance, with a basic window of 20, ParCorr takes at most 160 seconds to process a sliding window, while iSAX Approximate needs 1990 seconds. We attribute this advantage to two factors: the calculation of sketches is incremental and the parallelization of the algorithm is natural. These results also hold for the seismic dataset as can be seen in Figure 10. For the much smaller financial dataset (Figure 11), ParCorr is still faster than iSAX, although less so.

Figure 12 shows that ParCorr scales well to large datasets containing up to 100 million time series. iSAX Approximate is consistently about 50% faster than iSAX exact. Our competitors (Parallel linear search, iSAX Approximate/Exact) do not scale

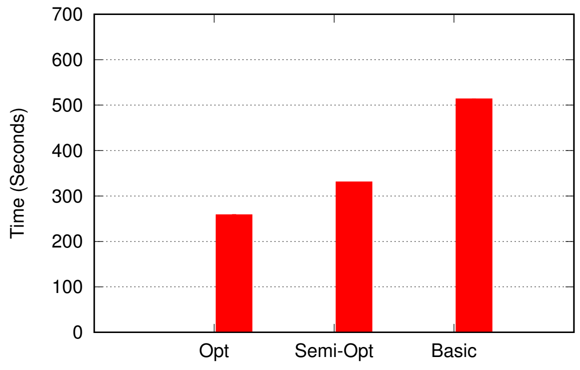

Fig. 8: Execution time of step 4 of the algorithm for each of the communication strategies introduced in Section 4.4. The algorithms are run on a cluster of 32 nodes and 5 million time series (basic window of 32 and sliding window of 256). The optimized strategy gives the best response time.

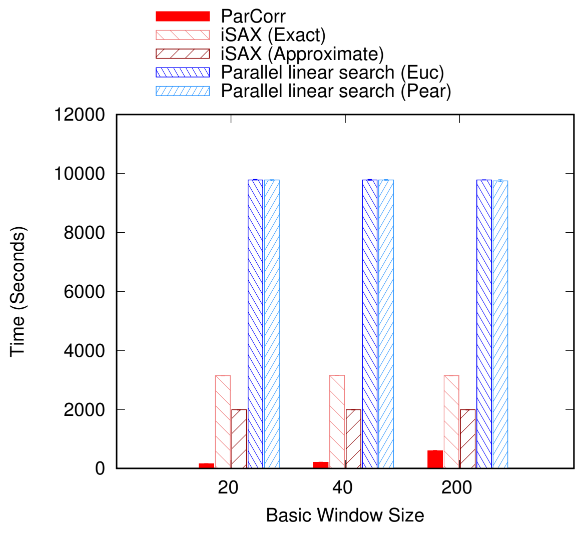

Fig. 9: Execution time for the calculation of the correlations for each sliding window as a function of basic window size for the random walk dataset. The algorithms are run on a cluster of 32 nodes and 5 million time series. The time for ParCorr increases as the basic window size increases, because updating the sketch vector takes slightly longer. All parameters other than basic window size are set to their values from Table 4.

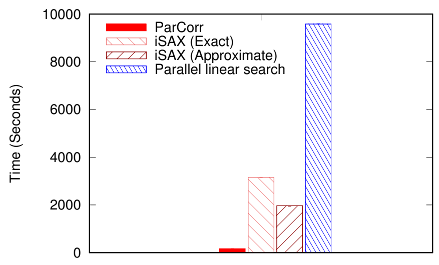

Fig. 10: Execution time for the calculation of the correlations for each sliding window for the seismic dataset. The algorithms are run on a cluster of 32 nodes and 5 million time series. All parameters are set to their values from Table 4.

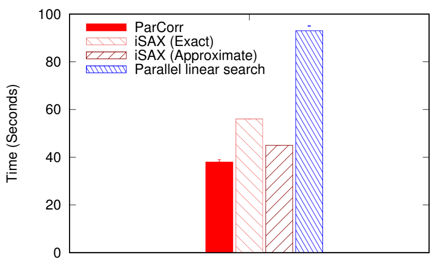

Fig. 11: Execution time for the calculation of the correlations for each sliding window for the financial dataset. The algorithms are run on a cluster of 32 nodes and 40K time series. All parameters are set to their values from Table 4.

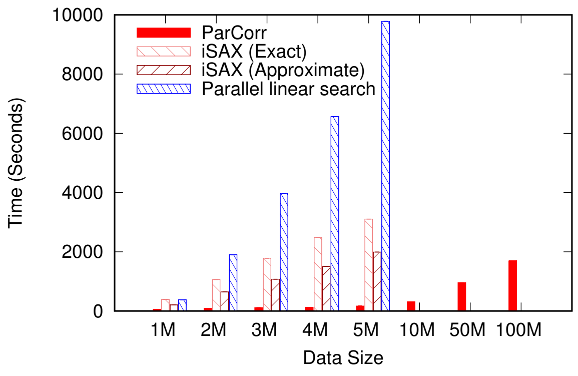

Fig. 12: Execution time for the calculation of the correlations for each sliding window as a function of dataset size for the random walk dataset. The algorithms are run on a cluster of 32 nodes. All parameters are set to their values from Table 4. Note that ParCorr scales to larger datasets nearly linearly and the times remain practical. The other methods exceeded the measurement window.

since they cannot handle more than 5 million time series due to the fact that both memory usage and communication costs grow too fast.

Figure 13 shows that both iSAX and ParCorr enjoy a roughly linear speedup, whereas Figure 14 shows that ParCorr is orders of magnitude faster in absolute time at all degrees of parallelization. ParCorr needs at most 598 seconds on 8 nodes (169 seconds on 32 nodes) while iSAX Approximate needs at most 13460 seconds (9784 seconds on 32 nodes).

Figure 15 shows that ParCorr's performance (using Spark) is comparable to iSAX (running natively without Spark) on a single node. ParCorr shows a small advantage but not as much as in a parallel setting, because Spark entails some overhead that

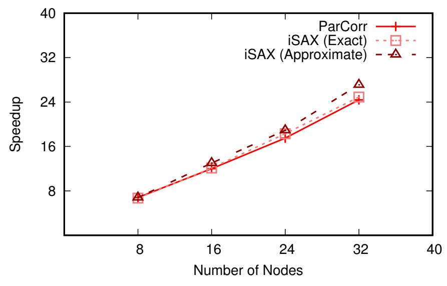

Fig. 13: SpeedUp: All algorithms enjoy linear speedup with roughly the same slope as the number of processing nodes increase. All parameters are set to their values from Table 4.

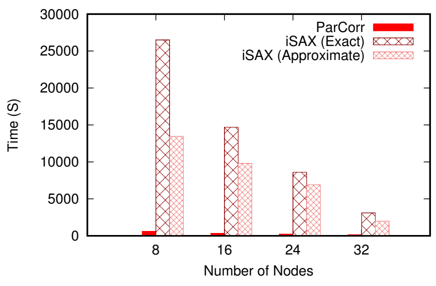

Fig. 14: Execution time for the calculation of the correlations for each sliding window as a function of the number of processing nodes for the random walk dataset. The algorithms are run on 5 million time series. All parameters are set to their values from Table 4.

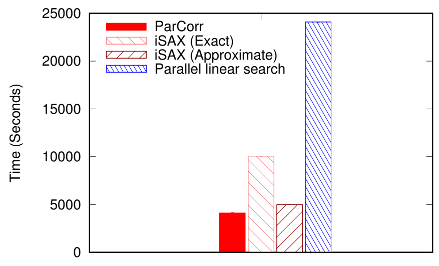

Fig. 15: Execution time for each sliding window on a single node for the random walk dataset. The dataset is 1 million time series. The SPARK overhead applies to ParCorr but not to iSAX. All parameters are set to their values from Table 4.

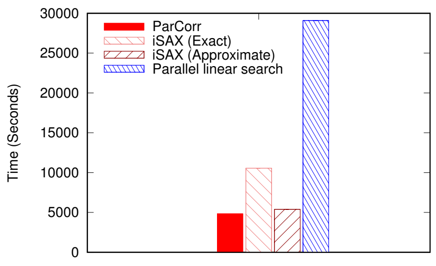

Fig. 16: Execution time for each sliding window on a single node for the seismic dataset. The dataset is 1 million time series. The SPARK overhead applies to ParCorr but not to iSAX. All parameters are set to their values from Table 4.

iSAX does not incur. These results are consistent with the results for the seismic and financial data as shown in Figures 16 and 17.

Figure 18 shows the high precision of ParCorr and iSAX. ParCorr verifies all the candidate pairs that the sketch filter produces. iSAX Exact incorporates a verification step as well. These results hold also for seismic and financial data as seen in Figures 20 and 22.

Figure 19 shows that iSAX Exact gives perfect recall because of its bounding box guarantee. ParCorr gives no such guarantee, so for applications that require 100% recall, iSAX Exact should be used. Empirically, ParCorr yields a recall of over 90%, as shown in Figures 21 and 23.

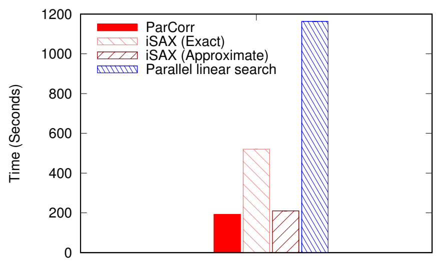

Fig. 17: Execution time for each sliding window on a single node for the financial dataset. The dataset is 40K time series. The SPARK overhead applies to ParCorr but not to iSAX. All parameters are set to their values from Table 4.

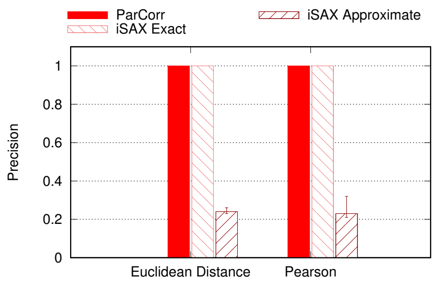

Fig. 18: Let Peuc and Pcorr be the sets of pairs of time series whose final w values fall within the threshold in the case of Euclidean distance and Pearson respectively. The precision is the fraction of the set of pairs found by each algorithm that belong to Peuc or Pcorr, respectively. ParCorr has 100% precision because it checks candidate pairs that are produced by the sketch algorithm.

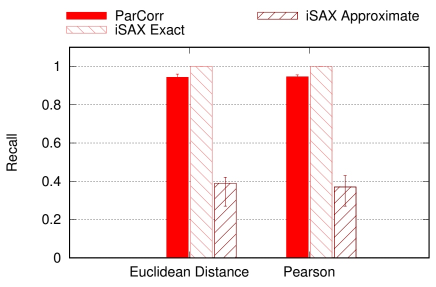

Fig. 19: Let Peuc and Pcorr be defined as in the caption of Figure 18. The recall is the fraction of Peuc or Pcorr, respectively, that is found by each algorithm. Note that iSAX Exact gives higher recall than ParCorr.

The experiments on real and synthetic data show that ParCorr is fast, scales well, guarantees 100% precision, and achieves very high recall. This reduction in recall is acceptable for many applications, especially given the high gain in response time.

6 Conclusion

Finding similar pairs of time series on sliding windows is useful for many applications. Methods to do so for hundreds of millions of time series in a highly efficient and scalable fashion is the contribution of this paper. Compared with the previous state-of-the-art iSAX, our solution is far faster and at least as scalable while showing only very little loss in recall. For many applications, where scalability is mandatory, this is highly beneficial.

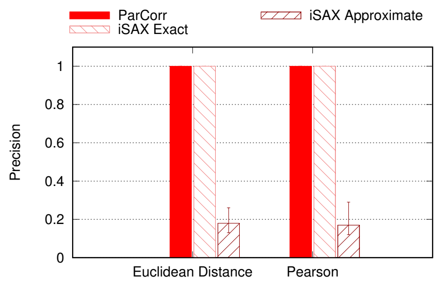

Fig. 20: For the seismic dataset, iSAX Exact and ParCorr both achieve 100% precision for Euclidean. ParCorr also achieves 100% precision for Pearson correlation.

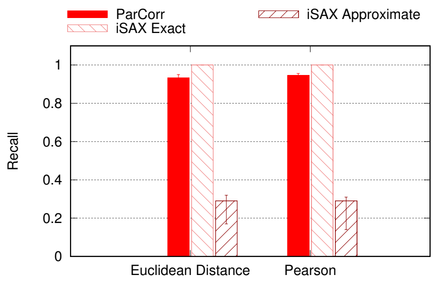

Fig. 21: For the seismic dataset, iSAX Exact achieves perfect recall. ParCorr achieves over 90% for both Euclidean and Pearson correlation.

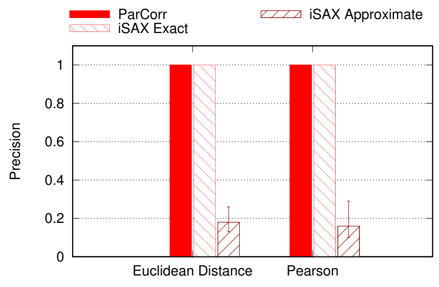

Fig. 22: For the financial dataset, iSAX Exact and ParCorr both achieve 100% precision for Euclidean. ParCorr also achieves 100% precision for Pearson correlation.

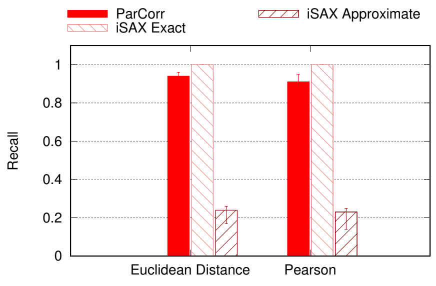

Fig. 23: For the financial dataset, iSAX Exact achieves perfect recall. ParCorr achieves over 90% for both Euclidean and Pearson correlation.

Future work includes:

1. Improving the recall perhaps by expanding the grid search to neighboring cells.
2. Extending this work to cases where we want to discover delayed correlations (where a window between $t1$ and $t1+w$ may be similar to a window between $t2$ and $t2+w$) in addition to co-temporous correlations. This is operationally straightforward to do by keeping the points representing previously treated time series in the grid structures, but it does require adjustments because the normalization transformation may change.
3. Allowing the window size to adapt to changes in data to find particularly highly correlated windows of different sizes; here there may be helpful analogies from tiling and from MASS.
4. Computing the statistical significance of correlations, taking into account multiple testing and the overlap of the windows.

7 Acknowledgement

The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732051.

References

1. Incorporated research institutions for seismology – seismic data access. http://ds.iris.edu/data/access/.
2. D. Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003.
3. R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 69–84. Springer-Verlag, 1993.
4. I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: efficient time series search and retrieval. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 252–263, 2008.
5. Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 599–610. ACM, 2004.
6. A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. iSAX 2.0: Indexing and mining one billion time series. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 58–67, 2010.
7. A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *Knowledge and Information Systems (KAIS)*, 39(1):123–151, 2014.
8. K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Transactions on Database Systems (TODS)*, 27(2):188–228, June 2002.
9. K. Chan and A. W. Fu. Efficient time series matching by wavelets. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 126–133. IEEE Computer Society, 1999.
10. M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 380–388, 2002.
11. R. Cole, D. Shasha, and X. Zhao. Fast window correlations over uncooperative time series. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 743–749. ACM, 2005.
12. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 419–429, 1994.
13. F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases. In *International Conference on Discovery Science*, pages 278–289, 2004.
14. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 518–529, 1999.
15. A. Gionis, H. Mannila, and J. Seppänen. Geometric and combinatorial tiles in 0–1 data. In *Knowledge Discovery in Databases: PKDD*, pages 173–184, 2004.
16. T. Guo, S. Sathe, and K. Aberer. Fast distributed correlation discovery over streaming time-series data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1161–1170, 2015.
17. D. Hallac, S. Vare, S. P. Boyd, and J. Leskovec. Toeplitz inverse covariance-based clustering of multivariate time series data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 215–223, 2017.
18. A. Henelius, I. Karlsson, P. Papapetrou, A. Ukkonen, and K. Puolamäki. Semigeometric tiling of event sequences. In *Machine Learning and Knowledge Discovery in Databases. ECML PKDD*, pages 329–344, 2016.

19. P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 189–197, 2000.
20. W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in Modern Analysis and Probability*, volume 26 of *Contemporary Mathematics*, pages 189–206, 1984.
21. E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems (KAIS)*, 3(3):263–286, 2001.
22. E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC)*, pages 614–623, 1998.
23. Y. Matsubara and Y. Sakurai. Regime shifts in streams: Real-time forecasting of co-evolving time sequences. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1045–1054, 2016.
24. A. Mueen, S. Nath, and J. Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 171–182, 2010.
25. A. Mueen, Y. Zhu, M. Yeh, K. Kamgar, K. Viswanathan, C. Gupta, and E. Keogh. The fastest similarity search algorithm for time series subsequences under euclidean distance, August 2017. `http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html`.
26. S. Papadimitriou, J. Sun, and C. Faloutsos. Streaming pattern discovery in multiple time-series. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 697–708, 2005.
27. S. Papadimitriou and P. S. Yu. Optimal multi-scale patterns in time series streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–658, 2006.
28. C. Perng, H. Wang, and S. Ma. Fast relevance discovery in time series. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 1016–1020, 2006.
29. Y. Sakurai, C. Faloutsos, and M. Yamamuro. Stream monitoring under the time warping distance. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1046–1055, 2007.
30. D. Shasha and Y. Zhu. *High Performance Discovery in Time series, Techniques and Case Studies*. Springer, 2004.
31. J. Shieh and E. Keogh. iSAX: Indexing and mining terabyte sized time series. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 623–631, 2008.
32. Q. Xie, S. Shang, B. Yuan, C. Pang, and X. Zhang. Local correlation detection with linearity enhancement in streaming data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 309–318, 2013.
33. C. M. Yeh, H. V. Herle, and E. J. Keogh. Matrix profile III: the matrix profile allows visualization of salient subsequences in massive time series. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 579–588, 2016.
34. C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. J. Keogh. Matrix profile I: all pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 1317–1322, 2016.
35. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.
36. Y. Zhu, N. Imamura, D. N. Nikovski, and E. J. Keogh. Matrix profile VII: time series chains: A new primitive for time series data mining. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2017.
37. Y. Zhu, Z. Zimmerman, N. S. Senobari, C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. J. Keogh. Matrix profile II: exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 739–748, 2016.
38. K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1555–1566, 2014.