ParIS: The Next Destination for Fast Data Series Indexing and Query Answering

Botao Peng
LIPADE, Paris Descartes Univ.
botao.peng@parisdescartes.fr

Panagiota Fatourou
FORTH ICS & Dept. of Comp. Science, Univ. of Crete
faturu@csd.uoc.gr

Themis Palpanas
LIPADE, Paris Descartes Univ.
themis@mi.parisdescartes.fr

***Abstract*—We propose ParIS, the *first* disk-based data series index that inherently takes advantage of modern hardware parallelization, in order to accelerate processing times. Our experimental results demonstrate that ParIS completely removes the CPU latency during index construction for disk-resident data. In terms of exact query answering, ParIS is more than 2 orders of magnitude faster than the current state of the art index scan method, and more than 3 orders of magnitude faster than the optimized serial scan method. ParIS owes its efficiency not only to the effective use of multi-core and multi-socket architectures, in order to distribute and execute in parallel both index construction and query answering, but also to the exploitation of the Single Instruction Multiple Data (SIMD) capabilities of modern CPUs, in order to further parallelize the execution of individual instructions inside each core.**

I. Introduction

[Motivation] An increasing number of applications across many diverse domains continuously produce very large amounts of data series[1] (such as in finance, environmental sciences, astrophysics, neuroscience, engineering, multimedia, and others [1], [2], which makes them one of the most common types of data. When these sequence collections are generated, users need to query and analyze them as soon as they become available, a process that is heavily dependent on data series similarity search (which apart from being a useful query in itself, also lies at the core of several machine learning methods, such as, clustering, classification, motif and outlier detection, etc.). The brute-force approach for evaluating similarity search queries is by performing a sequential pass over the complete dataset. However, as data series collections grow larger, scanning the complete dataset becomes a performance bottleneck, taking hours or more to complete [3]. This is especially problematic in exploratory search scenarios, where every next query depends on the results of previous queries. Consequently, we have witnessed an increased interest in developing indexing techniques and algorithms for similarity search [4], [5], [6], [7], [3], [8], [9], [10], [11], [12], [13].

[Scalability problem] Nevertheless, the continued increase in the rate and volume of data series production with collections that grow to several terabytes in size [1] renders existing data series indexing technologies inadequate. For example, the current state-of-the-art index, ADS+ [3], requires more than 4min to answer every single exact query on a moderately sized 250GB sequence collection. Therefore, traditional solutions and systems are inefficient at, or incapable of managing and processing the voluminous sequence collections that already exist in several domains.

Moreover, we note that, given the evolution of CPU performance, where the processor clock speed is not increasing due to the power wall constraint, algorithmic speedups can now only come by exploiting the parallelism opportunities offered by modern hardware [14], [15], [16].

[Parallel Indexing] In this work, we propose the Parallel Index for Sequences (ParIS), the first data series index that inherently takes advantage of modern hardware parallelization, and incorporates the state-of-the-art techniques in sequence indexing, in order to accelerate processing times. ParIS, which is a disk-based index, can effectively take advantage of multi-core and multi-socket architectures, in order to distribute and execute in parallel the computations needed for both index construction and query answering. Moreover, ParIS exploits the Single Instruction Multiple Data (SIMD) capabilities of modern CPUs, in order to further parallelize the execution of individual instructions inside each core. Overall, ParIS manages to remove the CPU cost during index creation (which is now purely I/O bounded) and make it 2.4x faster than the current state-of-the-art approach [3]. Experiments show also the effectiveness of ParIS in exact query answering: it is up to 1 order of magnitude faster than the state-of-the-art index scan method [3], and up to 3 orders of magnitude faster than the state-of-the-art optimized serial scan [5]. We also note that ParIS has the potential to deliver more benefit as we move to faster storage mediums.

To achieve the ParIS goals, we made careful design choices in the coordination of the compute and I/O tasks, and consequently, developed new algorithms for the construction of the index and for answering similarity search queries on this index. For query answering in particular, we studied alternative solutions, and evaluated the tradeoff between execution speed and the amount of communication among the parallel worker threads, which affects the effectiveness of each individual worker. Our study shows that each one of the alternative algorithms is suitable for different hardware platforms.

We note that even though scaling out to multiple machines is also a valid research direction, in this work, we focus on

[1]A data series, or data sequence, is an ordered sequence of data points. If the ordering dimension is time then we talk about time series, though, series can be ordered over other measures (e.g., angle in astronomical radial profiles, mass in mass spectroscopy, position in genome sequences, etc.).

addressing the problem in the context of a single machine, so as to maximize the benefit we can get out of the hardware. Our results can easily be used as the basis for a scale-out solution.

[Contributions] The contributions we make in this paper, can be summarized as follows.

1. We propose ParIS, the first data series index designed for modern hardware. We describe parallel algorithms for index creation and *exact* query answering, which employ parallelism in reading the data from disk and processing them in the CPU. At the same time, they achieve a balanced workload distribution among the worker threads, and enable these threads to exchange information (while executing) in order to reduce the amount of work to be done.

2. In order to further speedup query answering, we exploit SIMD for complex vectorial computations: we develop a novel vectorized implementation for computing the lower bounding distance between the Piecewise Aggregate Approximation (PAA) and indexable Symbolic Aggregate approXimation (iSAX) representations.

3. Finally, we experimentally evaluate ParIS using a variety of synthetic and real datasets. The results demonstrate the efficiency of the proposed approach, which is orders of magnitude faster for exact query answering than the state-of-the-art methods. Moroever, the results show that ParIS can create the index much faster than the current state-of-the-art, completely masking out the CPU latency, and that it has the potential to deliver more benefit as we move to faster storage mediums.

II. Preliminaries

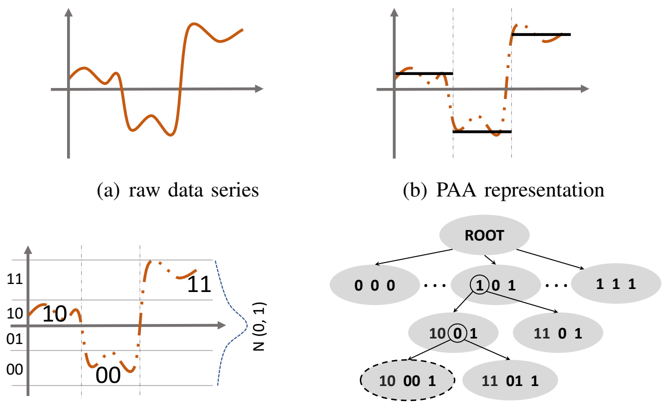
We now provide some necessary definitions, and introduce the related work on state-of-the-art data series indexing.

A. Data Series and Similarity Search

[Data Series] A data series, $S = \{p_1, ..., p_n\}$, is defined as a sequence of points, where each point $p_i = (v_i, t_i)$, $1 \leq i \leq n$, is associated to a real value v_i and a position t_i. The position corresponds to the order of this value in the sequence. We call n the *size*, or *length* of the data series. We note that all the discussions in this paper are applicable to high-dimensional vectors, in general.

[Similarity Search] Analysts perform a wide range of data mining tasks on data series including clustering [17], classification and deviation detection [18], [19], and frequent pattern mining [20]. Existing algorithms for executing these tasks rely on performing fast similarity search across the different series. Thus, efficiently processing nearest neighbor (NN) queries is crucial for speeding up the above tasks. NN queries are formally defined as follows: given a query series S_q of length n, and a data series collection $\mathcal{S}$ of sequences of the same length, n, we want to identify the series $S_c \in \mathcal{S}$ that has the smallest distance to S_q among all the series in the collection $\mathcal{S}$. (In the case of streaming series, we first create subsequences of length n using a sliding window, and then index those.)

Common distance measures for comparing data series are Euclidean Distance (ED) [21] and dynamic time warping (DTW) [5]. While DTW is better for most data mining tasks,



(a) raw data series (b) PAA representation (c) iSAX representation (d) ADS+ index

Fig. 1. The iSAX representation, and the ADS+ index structure

the error rate using ED converges to that of DTW as the dataset size grows [4]. Therefore, data series indexes for massive datasets use ED as a distance metric [4], [5], [6], [7], [3], though simple modifications can be applied to make them compatible with DTW [4]. Euclidean distance is computed as the sum of distances between the pairs of corresponding points in the two sequences. Note that minimizing ED on z-normalized data (i.e., a series whose values have mean 0 and standard deviation 1) is equivalent to maximizing their Pearson's correlation coefficient [22].

[Distance calculation in SIMD] Single-Instruction-Multiple-Data (SIMD) refers to a parallel architecture that allows the execution of the same operation on multiple data simultaneously [23]. Using SIMD, we can reduce the latency of an operation, because the corresponding instructions are fetched once, and then applied in parallel to multiple data. All modern CPUs support 256-bit wide SIMD vectors, which means that certain floating point (or other 32-bit data) computations can be up to 8 times faster when executed using SIMD. Even though no SIMD solutions have been proposed so far for data series indices, this idea has been exploited for the computation of distance functions [24]. In our study, we take an extra step, and we also use SIMD for operations related to the proposed data series index structure (i.e., for conditional branch calculations during the computation of the lower bound distances; see Section III-B).

B. iSAX Representation and ADS+ Index

[iSAX Representation] The iSAX representation is based on the Piecewise Aggregate Approximation (PAA) representation [25], which divides the data series in segments of equal length, and uses the mean value of the points in each segment in order to summarize a data series. Figure 1(b) depicts an example of PAA representation with three segments (depicted with the black horizontal lines), for the data series depicted in Figure 1(a). Based on PAA, the indexable Symbolic Aggregate approXimation (iSAX) representation was proposed [4]. This method first divides the (y-axis) space in different regions, and assigns a bit-wise symbol to each region. In practice,

the number of symbols is small: iSAX achieves very good approximations with as few as 256 symbols, the maximum alphabet cardinality, $|alphabet|$, which can be represented by eight bits [7]. It then represents each segment w of the series with the symbol of the region the PAA falls into, forming the word $10_2 00_2 11_2$ shown in Figure 1(c) (subscripts denote the number of bits used to represent the symbol of each segment).
[ADS+ Index] Based on this representation, the state-of-the-art ADS+ index was developed [3]. It makes use of variable cardinalities (i.e., variable degrees of precision for the symbol of each segment) in order to build a hierarchical tree index (see Figure 1(d)), consisting of three types of nodes: (i) the root node points to several children nodes, 2^w in the worst case (when the series in the collection cover all possible iSAX representations); (ii) each inner node contains the iSAX representation of all the series below it, and has two children; and (iii) each leaf node contains both the iSAX representation *and* the raw data of all the series inside it (in order to be able to prune false positives and produce exact, correct answers). When the number of series in a leaf node becomes greater than the maximum leaf capacity, the leaf splits: it becomes an inner node and creates two new leaves, by increasing the cardinality of the iSAX representation of one of the segments (the one that will result in the most balanced split of the contents of the node to its two new children [7], [3]). The two refined iSAX representations (new bit set to *0* and *1*) are assigned to the two new leaves. In our example, the series of Figure 1(c) will be placed in the outlined node of the index (Figure 1(d)).

The ParIS index uses the iSAX representation and basic ADS+ index structure, and proposes techniques and algorithms specifically designed for modern hardware.

III. Proposed Solution: ParIS

In this section, we describe our approach, Parallel Indexing of Sequences (ParIS) for parallel index construction.

Figure 2 provides a high level overview of the entire pipeline of how the ParIS index is created and then used for query answering. This pipeline comprises of four concrete stages. In Stage 1, a thread, called the *coordinator worker*, reads raw data series from the disk and transfers them into the *raw data buffer* in main memory. In Stage 2, a number of *IndexBulkLoading workers*, process the data series in the raw data buffer to create their iSAX summarizations. These summarizations determine the root subtree in which each series belongs. The summarizations are then stored in one of the index Receiving Buffers (RecBufs) in memory. There are as many RecBufs as the root subtrees of the index tree, each one storing the iSAX summarizations that belong to a single subtree. This number is usually a few tens of thousands and at most 2^w, where w is the number of segments in the iSAX representation of each time series (w is fixed to 16 in this paper, as in previous studies [3]). The iSAX summarizations are also stored in SAX, the iSAX summarizations array.

When all RecBufs are full, Stage 3 starts. In this stage, a pool of *IndexConstruction workers* process the contents of RecBufs; every such worker has been assigned a distinct RecBuf at each time: it reads the data stored in it and builds the corresponding index subtree. So, root subtrees are built in parallel. The leaves of each subtree is flushed to the disk at the end of the tree construction process. This results in free space in main memory. These 3 stages are repeated until all raw data series are read from the disk, the entire index tree is constructed, and the SAX array is completed. The index tree together with SAX form the ParIS index, which is then used in Stage 4 for answering similarity search queries.

In the following, we elaborate on the stages of this pipeline.

A. ParIS Index Building

The main challenge in devising an algorithm for the creation of our index in parallel is that a significant part of time is required for disk I/O (both reading and writing). In order to address this challenge, we concentrate our efforts in two directions: execute the CPU computations so as to achieve the largest possible overlap with the required disk I/O, and reduce the number of random accesses to disk as much as possible. We achieve these by maintaining the synchronization cost among different threads as low as possible.

1) Index Initialization: In this section, we provide the details of Stages 1 and 2. Figure 3(a) summarizes how the coordinator and IndexBulkLoading workers work.

The raw data buffer is implemented using double buffering. So, it is comprised of two parts, one on which the coordinator works, and another on which the IndexBulkLoading workers work. In this way, the data the coordinator is accessing and the data the IndexBulkLoading workers are handling form two independent sets. So, all these threads work in parallel (as much as possible). Our tuning experiments (omitted for brevity) showed that setting the size of the double buffer size to 2MB results in the best performance (the time cost reduces and then stabilizes once the buffer size gets larger than 2MB).

The pseudocode for the coordinator worker is shown in Algorithm 1. We assume that variable $index$ is a data structure containing all buffers, a pointer to the root of the tree index, some arrays of locks that are needed for synchronizing access to RecBufs, and SAX. In this algorithm, B_1 and B_2 are pointers to the two parts of the raw data (double) buffer, i.e., $TS[0]$ and $TS[1]$. Moreover, we denote by n_t the number of IndexBulkLoading workers that are created by the coordinator (see discussion below about the value of n_t). The algorithm works as follows. The coordinator worker first fills in the part of the raw data buffer pointed to by B_1 (line 3). Then, the coordinator worker creates the n_t IndexBulkLoading worker threads (lines 6). These threads create the iSAX summarizations of the data in the raw data buffer part pointed to by B_1 and place them in the appropriate RecBuf and in SAX (see Figure 3(a)); for each data series, we also store in RecBuf its offset in the raw data file. While the IndexBulkLoading workers are performing this task, the coordinator concurrently fills in the other part of the raw data buffer (line 8). This process is repeated until the main memory is exhausted.

The coordinator worker is aware of the current memory usage by monitoring the number of data series that it has

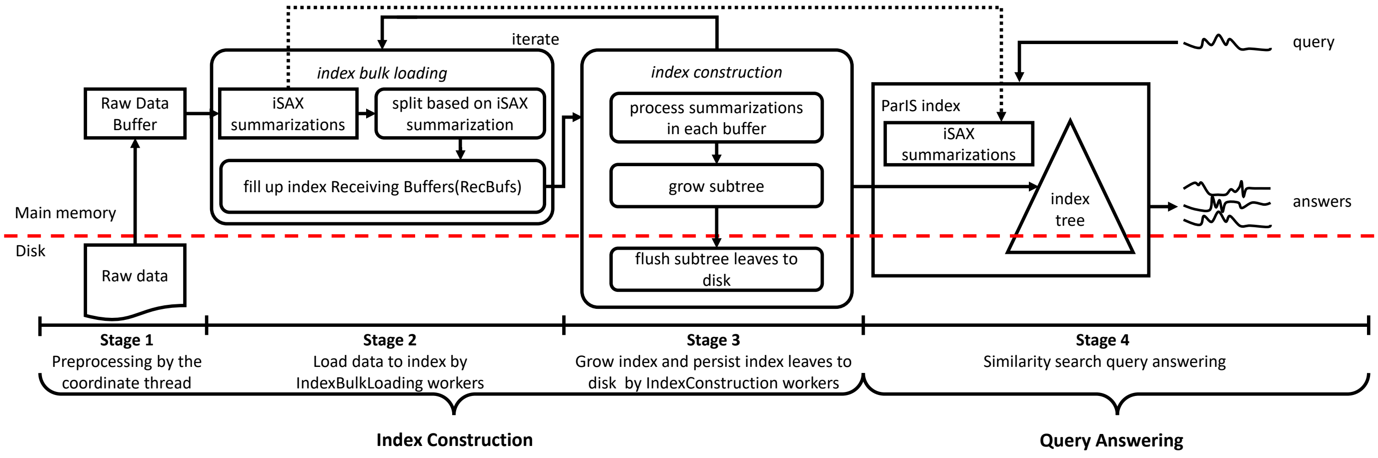

iterate
index bulk loading
Raw Data Buffer
iSAX summarizations
split based on iSAX summarization
fill up index Receiving Buffers(RecBufs)
index construction
process summarizations in each buffer
grow subtree
flush subtree leaves to disk
query
ParIS index
iSAX summarizations
index tree
answers
Main memory
Disk
Raw data
Stage 1
Preprocessing by the coordinate thread
Stage 2
Load data to index by IndexBulkLoading workers
Stage 3
Grow index and persist index leaves to disk by IndexConstruction workers
Stage 4
Similarity search query answering
Index Construction
Query Answering

Fig. 2. Overview of the pipeline for creating the ParIS index, and using the index for query answering.

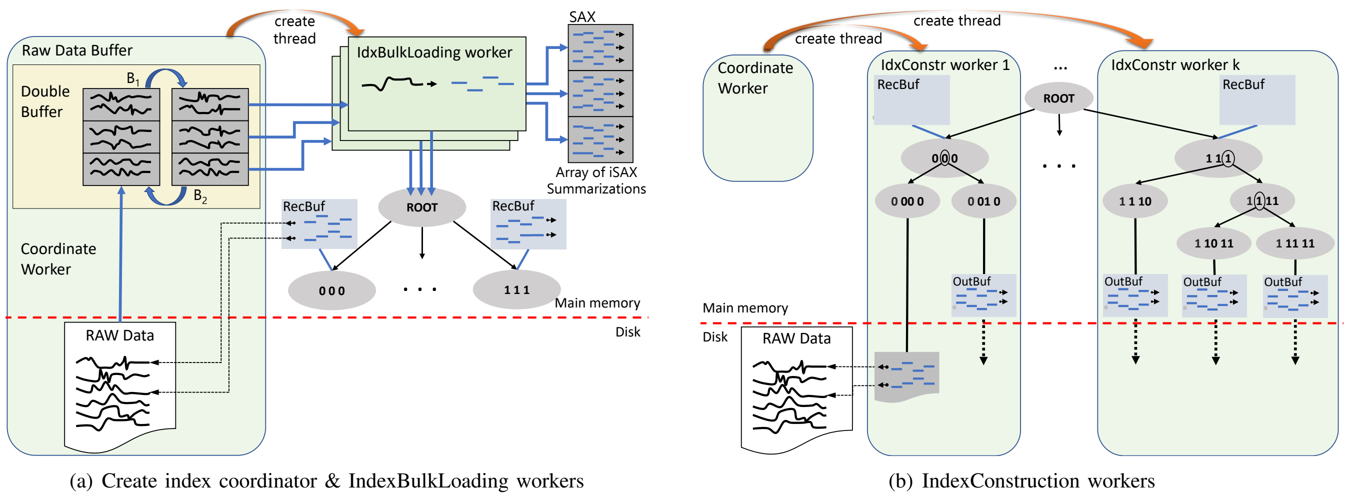


Fig. 3. Workflow and algorithms relevant to index creation.

processed. When the available memory is (nearly) exhausted[2] (line 10), then the coordinator creates the IndexConstruction worker threads (lines 12), which build the part of the index that corresponds to the iSAX summarizations stored in the RecBuf, and flush the leaf nodes of the tree to disk.

The pool of IndexBulkLoading workers could be as big as the number of cores in our machine (minus one which is reserved for the coordinator). IndexBulkLoading workers are assigned each RecBuf one-at-a-time in round-robin fashion, either by using atomic fetch and increment, or a lock. As we will discuss later (in Section IV), 5 IndexBulkLoading workers and 6 IndexConstruction workers are enough to completely mask out the CPU latency at this stage; note that these numbers are orders of magnitude less than the number of the index root subtrees (usually tens of thousands). This is true even if the coordinator creates the IndexBulkLoading workers from scratch each time it fills up a part of the raw data buffer (doing so avoids synchronization and simplifies the implementation). The reason is that the computation is heavily I/O bounded at this stage (and therefore the cost of periodically creating/destroying threads is negligible). For the same reason, techniques like thread pinning does not improve the demonstrated performance. We also note that because of the small number of BulkIndexLoading (and IndexConstruction workers), the use of locks for synchronizing access to RecBufs (or the assignment of subtrees) does not result in any synchronization bottlenecks.

The pseudocode that the IndexBulkLoading workers execute is shown in Algorithm 2. Each such worker has been assigned a chunk, of size $chunksize$, in each part of the raw data buffer (therefore, the size of the raw data buffer is $2 * chunksize * n_t$. Each worker operates only on its chunk. In this way, no synchronization is needed between the IndexBulkLoading workers for accessing the raw data buffer. Each IndexBulk-Loading worker reads the data series in its block one after the other (line 2) and calculates the iSAX summarization for

[2]Note that we only need a small amount of additional memory for creating new index nodes in the subtree of the root currently being processed, which can have a maximum depth of $w(alphabet - 1)$ [3]. Moving data inside the index (e.g., from RecBuf to OutBuf, as we will discuss later) does not require extra memory: we reallocate the same memory addresses between the buffers.

Algorithm 1: *CreateIndex*

```
Input: File* $file$, Index $index$, Integer $n_t$
1  Pointer $B_1 \leftarrow index.TS[0], B_2 \leftarrow index.TS[1]$;
2  Integer p = 0;
3  $B_1 \leftarrow$ read data from $file$;
4  while not reached end of $file$ do
5      for $i \leftarrow 1$ to $n_t$ do
6          create a thread to execute an instance of
             $IndexBulkLoading(index, B_1, p + i * chunksize)$;
7      $B_2 \leftrightarrow B_1$;
8      $B_1 \leftarrow$ read data from $file$ ;
9      Wait for IndexBulkLoading workers to finish;
10     if main memory is full then
11         for $i \leftarrow 1$ to $n_t + 1$ do
12             create a thread to execute an instance of
                 $IndexConstruction(index)$;
13         Wait for IndexConstruction workers to finish;
14     $p \leftarrow p + n_t * chunksize$;
```

Algorithm 2: *IndexBulkLoading*

```
Input: Index $index$, Raw data buffer $TS[]$, Integer $p$
1  for $i \leftarrow 0$ to $chunksize - 1$ do
2      $index.SAX[p + i]$ = $ConvertToSAX$ ($TS[i]$);
3      acquire appropriate lock from $index.RecBufLock[]$;
4      $InsertIntoRecBuf$ ($\langle index.SAX[p + i], p + i \rangle$);
5      release the acquired lock;
```

each of them by calling function ConvertToSAX(). Finally, it stores this iSAX summarization in SAX (line 2) and in the appropriate RecBuf (line 4). Recall that each RecBuf gathers together all data that must be stored into the same root subtree. These data may exist in blocks of the raw data buffer that are associated to different IndexBulkLoading workers. So, more than one such workers may require to concurrently access the same RecBuf. Therefore, synchronization is needed. This synchronization is achieved by using a lock for each such buffer, stored in array $RecBufLock[]$ of $index$.

To eliminate the need for synchronization between the IndexBulkLoading workers in accessing SAX, the iSAX summarization of the data series stored in the p-th position of the raw data file, is stored in the p position of SAX.

2) Subtree Construction and Leaf Materialization: We now describe Stage 3, in which the index is gradually constructed and its leaves are materialized. In addition to the raw data buffer and the RecBufs, ParIS makes use of an additional layer of main memory buffers, called the *Output Buffers* (OutBufs). Each OutBuf is associated to a distinct leaf of the index tree.

When the coordinator worker discovers that the main memory is exhausted, it creates a number of IndexConstruction workers (recall that based on our experiments, this number is 6). These workers process the data in the RecBufs in order to grow the corresponding subtree, until the data end up in the OutBufs of that subtree. Finally, the OutBufs are flushed to disk. This process is illustrated in Figure 3(b).

All IndexConstruction workers process different root subtrees, so they work independently and no synchronization is needed. A worker that finishes its work on one subtree gets assigned to a new RecBuf, until all RecBufs are processed.

Algorithm 3: *IndexConstruction*

```
Input: Index $index$
1  Shared integer $n_b = 0$;
2  while (TRUE) do
3      $i \leftarrow$Atomically fetch and increment $n_b$ ;
4      if ($i \geq 2^w$) then break ;
5      for every $\langle isax, pos \rangle$ pair $\in index.RecBuf[i]$ do
6          $targetLeaf \leftarrow$ Leaf of $index$ tree to insert $\langle isax, pos \rangle$;
7          while $targetLeaf$ is full do
8              SplitNode($targetLeaf$);
9              $targetLeaf \leftarrow$ New leaf to insert $\langle isax, pos \rangle$;
10         Insert $\langle isax, pos \rangle$ in $targetLeaf$'s OutBuf buffer;
11     Flush $targetLeaf$'s OutBuf buffer to disk;
12     Clear this node OutBuf;
```

In order to maintain the scheme simple and efficient, we have chosen not to parallelize processing inside each one of the index root subtrees since that would require a lot of synchronization (due to node splitting). Experiments have shown that this decision does not have any negative impact in the performance of our scheme.

The pseudocode that the IndexConstruction workers execute is shown in Algorithm 3. An IndexConstruction worker first selects one of the RecBufs to process in an atomic way (line 3). This can be done by using either an atomic *fetch_and_add* primitive, or a lock. Then, it moves the data to the appropriate OutBuf in the index (line 10), and if necessary (i.e., if the leaf node is full), it (repeatedly) performs node splitting (line 8). When node splitting is performed, the iSAX summarizations (i.e., the contents of the leaf node to be split) are read from disk and they are placed in the appropriate OutBuf. Then, the leaf node is split by creating two new leaf nodes and the data of the original leaf are moved to the new leaves. After that the OutBufs corresponding to the leaves of the subtree currently processed are flushed to disk (line 11).

B. ParIS Query-Answering

We now describe our method for parallel query-answering. The algorithm first performs an *approximate search* to obtain the first Best-So-Far (BSF) answer, and then proceeds with a sequential scan of the raw data that could not be pruned using the BSF, in order to produce the exact, final answer to the query. The approximate search is really fast, requiring only a negligible percentage (a few msec) of the (mostly) on-disk sequential scan cost. It is a simple, in-memory path traversal from the index root to the leaf with the iSAX representation that is the most similar to that of the query. Once a leaf is reached, the distance between the query and each of the leaf's data series is calculated. The minimum distance found is used as the first BSF answer (see left part of Figure 5). This BSF is used to prune the candidate series by computing lower bound distances to their summarizations. The series that are not pruned will be visited in the raw file, and the true distance will be computed (the BSF may be updated during this phase).

In the following, we concentrate on our algorithm for parallelizing the scan phase. We first describe how we exploit SIMD p the lower bound distance calculations.

1) Lower-Bound Distance Calculation: The algorithm starts by calculating the lower bound distance between the query series and the iSAX summarizations of all series in the index. This operation takes place entirely in main memory, since the iSAX summarizations are small enough to fit in the memory of modern servers[3]. This is a procedure that we can execute using SIMD, since both the queries and the index series are vectors, on which we need to perform the same operation (i.e., a distance calculation).

Using SIMD, we can perform eight calculations in parallel, using a single instruction (we assume a 256-bit SIMD vector which contains eight 32-bit float elements). We need to implement a conditional branch in SIMD, but contrary to previous solutions [24], this is a complex branch: not only do we have to use different conditional branches for different positions in the SIMD vector, but we also need to make different assignments for different branches. In our case, the calculation of the lower bound distance between the PAA of the query series and an iSAX summarization has three conditions, checking whether the PAA lies (i) above, (ii) below, or (iii) within the iSAX interval. Therefore, we need to choose different values from different dictionaries in order to perform the distance computation in SIMD.

To resolve this problem, we calculate the result of all branches, and use a conditional mask to extract the results in the correct branch (see Figure 4). We generate 3 branch masks and calculate the distance in those 3 conditions for every point in the vector. Using the appropriate SIMD instruction [26], we can calculate the value of 3 branch masks as the result of a condition judgment. Next we apply a logical "AND" between the 3 branch results and their masks. After that, all bits of the branch result in the wrong branch will be zero. Now there is only one value at the same position in those 3 branch results. Finally, we merge all possible branches in one vector, which is the correct final result.

In this way, we have a SIMD version of the distance computation function, which is a frequent and (CPU) time-consuming operation. Our solution renders all computations vectorial, which can not only accelerate the calculations, but also reduce the time spent for changing register types (the registers used for vector and normal values are different).

In order to evaluate the effect on performance of our SIMD lower bound distance calculation function, we measured the execution time of exact similarity search when all data are loaded in main memory (thus, factoring out the disk I/O cost). We compared our solution to the case where all computations are performed using Single Instruction Single Data (SISD). The results showed that the average time cost per lower-bounding calculation when using SIMD is 2.6x faster than the SISD solution. This is a non-negligible speedup, which is attributed to the large number of vectorial computations that need to be executed in the context of data series similarity search. (We omit the results due to lack of space.)

[3]This is true even for very large datasets: e.g., the highest granularity iSAX summarizations for 1 billion data series (occupying 1TB on disk) only need about 10GB of space in main memory.

2) Exact Search: The exact search algorithm employs approximate search as a first step and uses the approximate answer as the initial value of the BSF variable (see Algorithm 4). If BSF is not 0, exact search accesses in a sequential manner (on disk), all the raw data that could not be pruned. This is the step that we described how to implement in SIMD in the previous section. BSF is used for pruning and it is always updated to store the minimum distance calculated so far.

In order to benefit by parallel I/O and skip sequential reading, the ExactSearch separates the phase of the lower bound calculation from that of the real distance calculation and has different types of worker threads, namely the Lower Bound Computation (*LBC*) and the Real Distance Computation (*RDC*) workers, respectively, executing each type of calculation (see right part of Figure 5).

When a thread t executes an ExactSearch (Algorithm 4), it first performs an approximate search to get the initial BSF answer (line 3), and then it initiates a number of LBC workers (line 5). Different LBC workers work on different parts of SAX. Each such worker computes the lower bound distance between the query PAA and each iSAX summarization in its SAX part and records the data series for which this distance is less than the current BSF in a local candidate list, which it eventually returns to t (see Algorithm 5). This list contains the position and the lower-bound distance, needed to read the raw data and to calculate the real distance for the data series.

Once, all LBC workers have finished, t merges the candidate lists they have created (Algorithm 4, line 6) and initiates the RDC Worker (line 7).

Each RDC Worker (Algorithm 6) repeatedly retrieves a (minDistance, position) pair from the merged candidate list (C_l) in an atomic way (line 2). Atomicity is achieved with the use of a lock which all RDC workers share. The worker then reads the required data from disk, calculates the real distance (line 6), and if necessary, updates the shared BSF variable (line 8). A thread lock ensures that the BSF modification is done atomically. Storing BSF in shared memory and updating it during the course of the execution contributes towards reducing the number of calculations that RDC workers perform.

In this study, we use 1 LBC Worker thread per core, and 5 RDC Worker threads per core. Oversubscribing the RDC Workers (that are involved in expensive I/O operations) makes sure that we saturate the disk I/O bandwidth and the CPU remains busy. Our experiments showed that time performance remains relatively stable as we vary the number of RDC Worker threads per core (especially between 3-5 threads for the HDD server, and 4-10 threads for the SSD server), while 1 LBC Worker thread was enough to achieve the best performance. (For brevity, we omit these experiments.)

IV. Experimental Evaluation

[Setup] We ran the experiments on two servers, whose physical memory was limited to 75GB[4]. The first server (default)

[4]We used GRUB to limit the amount of RAM, so that all methods are forced to use the disk. Note that GRUB prevents the operating system from using the rest of the RAM as a file cache.

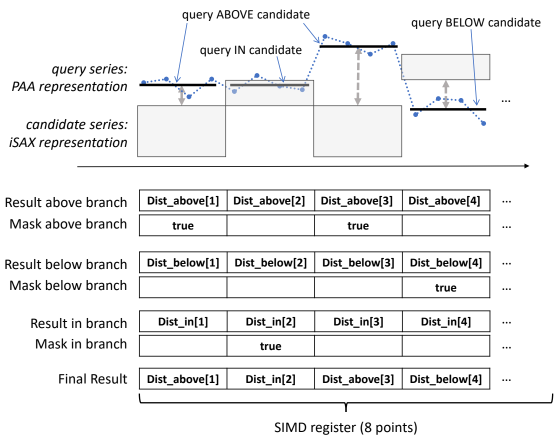


Fig. 4. SIMD conditional branch calculation.

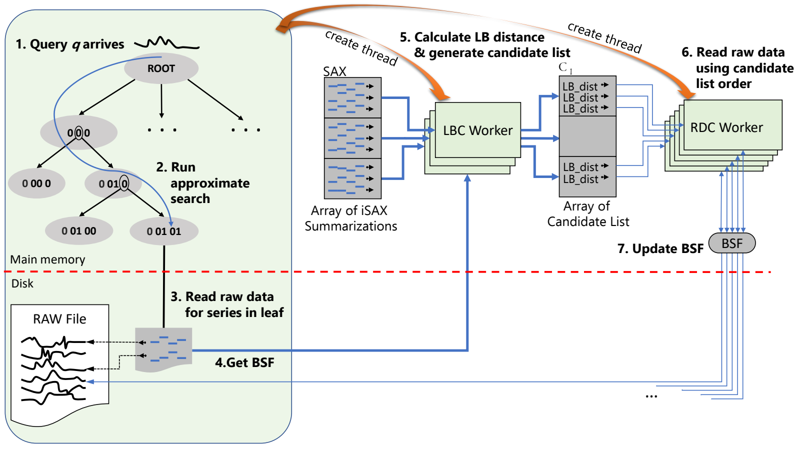

1. Query q arrives
ROOT
2. Run approximate search
Main memory
Disk
3. Read raw data for series in leaf
RAW File
4.Get BSF
create thread
5. Calculate LB distance & generate candidate list
SAX
Array of iSAX Summarizations
LBC Worker
C_l
LB_dist
Array of Candidate List
create thread
6. Read raw data using candidate list order
RDC Worker
7. Update BSF
BSF

Fig. 5. Workflow and algorithms relevant to query answering (balanced: ParIS)

Algorithm 4: $ExactSearch$

```
Input: querySeries $QTS$, query iSAX $isax$, Index $index$, File* $file$
1 candidate list $C_l$, $subC_l[]$;
2 float $BSF$;
3 BSF = approxSearch($QTS$, $isax$, $index$);
4 create a number of threads, each executing
    $subC_l \leftarrow LBCWorker(QTS$, proper part of $index.SAX$, BSF);
5 Wait for all threads to finish;
6 $C_l \leftarrow$ merge all sublists ($subC_l$) returned by the LBCWorker threads;
7 create a number of threads, each executing an instance of
    $RDCWorker$ ($QTS$, $C_l$, BSF, $file$);
8 Wait for all threads to finish;
9 return (BSF);
```

Algorithm 5: $LBCWorker$

```
Input: querySeries $QTS$, iSAX summarizations $SAX\_part[]$, float
       BSF
1 local candidate list $subC_l$;
2 for $i \leftarrow 1$ to size of $SAX\_part$ do
3     $minDist \leftarrow LowerBound\_SIMD$ ($QTS$, $SAX\_part[i]$);
4     if $minDist < BSF$ then
5         add ($minDist$, Raw Data file position of $SAX\_part[i]$) pair
          in $subC_l$;
6 return ($subC_l$)
```

with two Intel Xeon E5-2650 v4 2.2Ghz processors with 12 cores each, 10.8TB (6 x 1.8TB) 10K RPM SAS HDD drives in RAID0, with measured throughput of the RAID0 array 1200MB/sec. The second server, with the same setup for CPUs and memory, had 3.2TB (2 x 1.6TB) SATA SSD drives in

Algorithm 6: $RDCWorker$

```
Input: querySeries $QTS$, candidate list $C_l$, float BSF, File* $file$
1 while not reached end of $C_l$ do
2     Atomically read the next ($minDist$,$position$) pair from $C_l$;
3     if $minDist$ <$BSF$ then
4         Move file pointer to the proper position in $file$;
5         $rawData \leftarrow$ read raw data series from file;
6         $realDist \leftarrow Dist$ ($rawData$, $QTS$);
7         if $realDist < BSF$ then
8             Atomically update BSF to the value of $realDist$;
```

RAID0, measured throughput of the RAID0 array 500MB/sec. All algorithms were implemented in C, and compiled using the GCC6.2.0 on Ubuntu Linux 16.04.

[Datasets] In order to evaluate the performance of the proposed approach, we use several synthetic datasets for a fine grained analysis, and two real datasets from diverse domains. Unless otherwise noted, the series have a size of 256 points, which is a standard length used in the literature, and allows us to compare our results to previous work. We used synthetic datasets of sizes 50GB-250GB (with a default size of 100GB), and a random walk data series generator that works as follows: a random number is first drawn from a Gaussian distribution N(0,1), and then at each time point a new number is drawn from this distribution and added to the value of the last number. This kind of data generation has been extensively used in the past (and has been shown to model real-world financial data) [27], [4], [6], [7], [3]. We used the same process to generate 100 query series.

For our first real dataset, *Seismic*, we used the IRIS Seismic Data Access repository [28] to gather 100M series representing seismic waves from various locations, for a total size of 110GB. The second real dataset, *SALD*, includes neuroscience MRI data series [29], for a total of 200M series of size 128, of size 100 GB. In both cases, we used as queries 100 series out of the datasets (produced using our synthetic series generator).

In all cases, we repeated the experiments 5 times and we report the average values. We omit reporting the error bars, since all runs gave results that were very similar (less than 3% difference). Queries were always run in a sequential fashion, one after the other, in order to simulate an exploratory analysis scenario, where users formulate new queries after having seen the results of the previous one.

[Algorithms] We experiment with our ParIS algorithms, and compare those to the state of the art data series index, ADS+ [3]. We also compare to (i) the UCR Suite [5], the state of the art, optimized serial scan technique for exact similarity search, and (ii) the DS-Tree index [6] that stores the data in the leaves. All algorithms are available online [30]. We note that for the disk-resident experiments, we never loaded the datasets in main memory. In order to mitigate the effects of caching, we cleared the caches before each experiment (i.e., before running index creation and before executing each query).

A. Results

We present the performance results for the index creation and query answering in ParIS, and compare them to the results of the current state-of-the-art algorithm, ADS+, as well as the DS-Tree data series index.

1) Index Creation Performance Evaluation: In our first experiment, we evaluate the time it takes to create the data series for a synthetic dataset of 100M series (Figure 7). The results show that the proposed solution completely masks out the CPU latency and results in performance which is up to 2.4x faster than ADS+.

We observe that the performance of ParIS improves as the number of cores grows from 2 to 6 (note that a single thread runs on each core); after 6 cores the improvement is rather small. The reason for this behavior is illustrated in Figure 6. Note that there are 5 types of time cost: (i) read raw data from disk; (ii) write raw data on disk; (iii) write iSAX representations on disk (for exact search); (iv) manage data, which involves processing the raw data and inserting the iSAX representation in the correct RecBuf; and (v) create node, which takes place when we grow a subtree during the flushing of a RecBuf into the OutBufs. When we use 2 cores, the computation of read data from disk and the management of data series are divided between the 2 cores, and the index creation proceeds in parallel. Similarly, the time cost for the management of data series decreases with the number of cores, since the data that each core needs to process gets reduced. However, the time cost to read data is always the same, given that we have to access the same disk. Our I/O experiments showed that the total CPU cost becomes less than the I/O cost when we use more than 6 cores. Then, the management time cost becomes less than the time cost of reading.

Overall, these results demonstrate that not only does the proposed solution completely masks out the CPU latency (using only 6 threads), but it will continue to do the same when the storage medium of the dataset becomes much faster, e.g., with the use of NVRAMs. In the rest of this study, we use 6 cores as the default value.

We now turn our attention to datasets of increasing size, and additionally compare ParIS to another competitive data series index, DS-Tree. Figures 9 and 10 depict the results for both HDD and SSD. The results show that the ParIS algorithm is up to 2.4x faster than ADS+. Note that the DS-Tree is always one order of magnitude slower than the other approaches, so we do not consider the DS-Tree in the rest of our experiments.

2) Query Answering Performance Evaluation: We present results that demonstrate ParIS's efficiency in query answering: ParIS is more than 1 order of magnitude faster than ADS+, and up to 3 orders of magnitude faster than UCR Suite.

Figure 8 shows the exact query answering time for ParIS and ADS+ as we vary the number of cores. We observe that time performance improves as we increase the number of cores, though, the returns are negligible when we go beyond 6 cores.

We present in Figure 11 (log-scale y-axis) the results of the similarity search evaluation as the dataset size increases, for UCR Suite, ADS+ and ParIS. We observe that ParIS is one order of magnitude faster than ADS+, and more than two orders of magnitude faster than UCR Suite. We also note that the performance improvement of ParIS gets larger with increasing dataset sizes, as ParIS is able to scale better than UCR Suite. This is because ParIS can effectively prune the search space, while UCR Suite always has to read all the data from disk. Figure 12 (log-scale y-axis) shows the performance of exact query answering for the SSD server. Both ADS+ and ParIS benefit from the SSD low random access latency. The performance improvement of ParIS is increasing with the size of the dataset (since the number of random disk accesses increases, too), achieving in our experiments performance up to 15x faster than ADS+, and 2000x faster than UCR Suite.

3) Real Datasets: In this set of experiments, we test the different algorithms using real datasets. Figure 13(a) shows the result of index creation time cost on the SALD and Seismic real datasets, while Figure 13(b) (log scale y-axis) reports the exact similarity search time cost for UCR Suite, ADS+, and ParIS. Similar to our previous results, ParIS is faster than ADS+ during index creation: ParIS is up to 2x faster for SALD, and 1.7x faster for Seismic. In terms of query answering, the performance results differ for the two real datasets. For SALD, ParIS is 140x faster than UCR Suite and 4x faster than ADS+, while for Seismic, ParIS is 130x faster than UCR Suite and 5x faster than ADS+.

The SSD experiments show similar, yet more pronounced trends (Figure 13(c), log scale y-axis): ParIS is almost 1 order of magnitude faster than ADS+, and 3 orders of magnitude faster than UCR Suite.

4) Classification Task: In the final set of experiments, we tested ParIS on a complex analytics task. In particular, we evaluated its performance in a classification task, and measured the benefit it would bring to a k-NN Classifier, which is a classifier that assigns to a new object the majority class of the k nearest neighbors of that object (a data series, in our case). Figures 14 and 15 show the performance of ParIS and ADS+ for different values of k on a 100GB dataset. The results show that a k-NN Classifier using ParIS can finish a classification task up to 8x and 18x faster than when using ADS+ on HDD and SSD, respectively, which can reduce the total processing time for classifying 100K objects from several days down to a few hours.

Overall, ParIS exhibits a robust performance across different datasets and settings, and enables for the first time fast analysis of very large data series collections.

V. Related Work

[Data series summarization and indexing] Various dimensionality reduction techniques exist for data series, which can then be scanned and filtered [31], [32] or indexed and pruned [33], [34], [6], [4], [18], [3] during query answering.

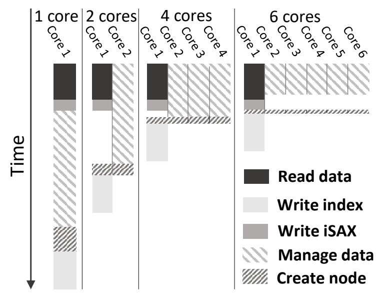


Fig. 6. Time cost illustration of index creation process

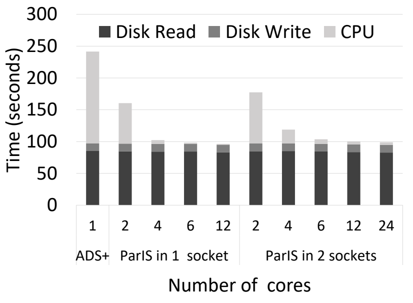


Fig. 7. Index creation time (HDD), varying the number of cores

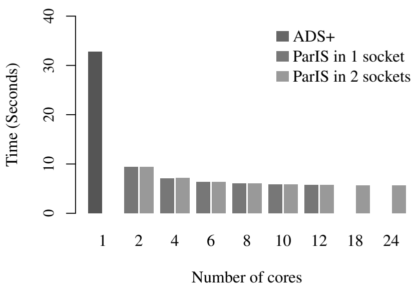


Fig. 8. Exact query answering time (HDD), varying the number of cores

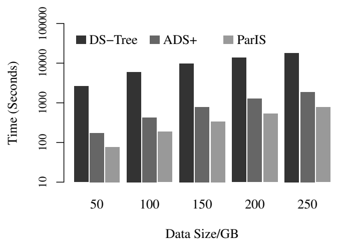


Fig. 9. Index creation time (HDD), varying the dataset size

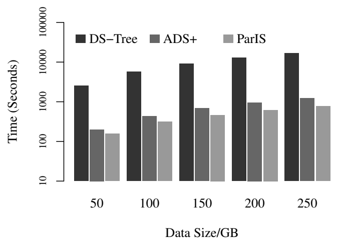


Fig. 10. Index creation time (SSD), varying the dataset size

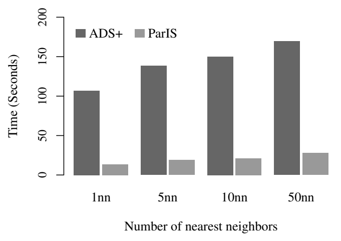


Fig. 14. Time for a k-NN Classifier that uses ADS+/ParIS to classify one object (100GB dataset, HDD)

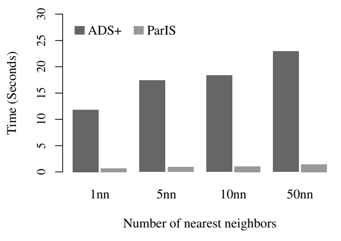


Fig. 15. Time for a k-NN Classifier that uses ADS+/ParIS to classify one object (100GB dataset, SSD)

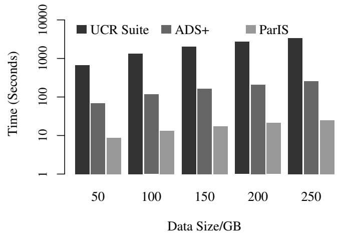


Fig. 11. Exact query answering time (HDD), varying the dataset size

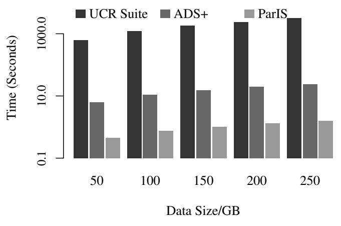
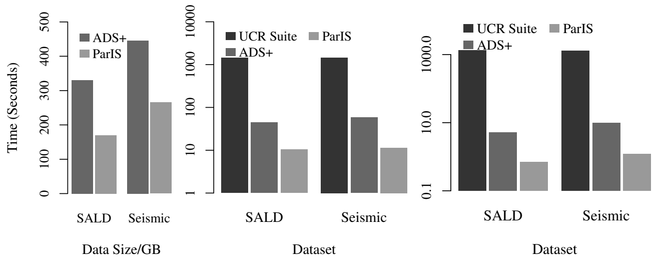


Fig. 12. Exact query answering time (SSD), varying the dataset size



(a) Index creation time (b) Exact query answering time (HDD) (c) Exact query answering time (SSD)

Fig. 13. Time cost for index creation and similarity search for real data

We follow the same approach of indexing the series based on their summaries, though our work is the first to exploit the parallelization opportunities offered by modern hardware, in order to accelerate index construction and similarity search. FastQuery is an approach used to accelerate search operations in scientific data [35], based on the construction of bitmap indices. In essence, the iSAX summarization used in our approach is an equivalent solution, though, specifically designed for sequences (which have high dimensionalities).

[Data structures for SIMD] While the interest in using SIMD for improving the performance of data management solutions is not new [36], there are still many algorithms that do not take advantage of this hardware characteristic. The problem of developing a SIMD-friendly B+-Tree index was recently studied [37], with a focus on a basic B+-Tree method, the k-ary search algorithm. For data series in particular, previous work has used SIMD for Euclidean distance computations [24]. In our work, we go beyond this straightforward use of SIMD, and we propose an algorithm that uses SIMD for the computation of lower bounds, which involve branching operations.

[Modern Hardware] Multi-core CPUs offer thread parallelism through multiple cores and simultaneous multi-threading (SMT). Thread-Level Parallelism (TLP) methods, like multiple independent cores and hyper-threads are commonly used to increase algorithm efficiency [38]. A recent study proposed a high performance temporal index similar to time-split B-tree (TSB-tree), called TSBw-tree, which focuses on transaction time databases [39]. However, this is designed for temporal data, which are 2-dimensional, while in our case, data series can have thousands of dimensions (i.e., the length of the sequence). Graphics Processing Units (GPUs) are another modern hardware option, which allows for massively parallel computations. A recent study described the use of a GPU in order to accelerate similarity search in a Trajectory Indexing system [40]. In our work, we do not use GPUs; though, it is a very interesting research direction, and deserves to be studied in its own right.

[Scans vs indexing] Even though recent works have shown that sequential scans can be performed efficiently [5], [41], such techniques are applicable when the dataset consists of a single, very long data series, and queries are looking for potential matches in small subsequences of this long series. Such approaches, in general, do not provide any benefit when the dataset is composed of a large number of small data series, like in our case. Therefore, indexing is required in order to efficiently support data exploration tasks, where the query workload is not known in advance.

VI. Conclusions

We presented ParIS, the first data series index that exploits the parallelism opportunities of modern hardware. The experimental evaluation with several synthetic and real datasets demonstrates the efficiency of ParIS, which is 2-3 orders of magnitude faster than previous approaches. Part of our future work is to study in more depth parallel I/O techniques [42], to combine our approach with solutions developed for distributed systems [12], and extend it to support other distance measures, such as DTW.

Acknowledgments This work was partially supported by the Chinese Scholarship Council, FMJH Program PGMO, EDF, Thales and HIPEAC 4. Part of the work was performed while P. Fatourou was visiting LIPADE, Paris Descartes University, and while B. Peng was visiting CARV, FORTH ICS.

References

[1] T. Palpanas, "Data series management: The road to big sequence analytics," *SIGMOD Record*, 2015.
[2] K. Zoumpatianos and T. Palpanas, "Data series management: Fulfilling the need for big sequence analytics," in *ICDE*, 2018.
[3] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Ads: the adaptive data series index," *VLDB J.*, vol. 25, no. 6, 2016.
[4] J. Shieh and E. Keogh, "i sax: indexing and mining terabyte sized time series," in *SIGKDD*, 2008.
[5] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh, "Searching and mining trillions of time series subsequences under dynamic time warping," in *SIGKDD*, 2012.
[6] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang, "A data-adaptive and dynamic segmentation index for whole matching on time series," *VLDB*, vol. 6, no. 10, pp. 793–804, 2013.
[7] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh, "Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+," *KAIS*, vol. 39, no. 1, pp. 123–151, 2014.
[8] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "Dpisax: Massively distributed partitioned isax," in *ICDM*, 2017.
[9] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas, "Coconut: A scalable bottom-up approach for building data series indexes," *PVLDB*, vol. 11, no. 6, pp. 677–690, 2018.
[10] M. Linardi and T. Palpanas, "Ulisse: Ultra compact index for variable-length similarity search in data series," in *ICDE*, 2018.
[11] ——, "Scalable, variable-length similarity search in data series: The ulisse approach," *PVLDB*, 2019.
[12] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "Massively distributed time series indexing and querying," *TKDE (to appear)*, 2018.
[13] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim, "The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art," *PVLDB*, 2019.
[14] L. Xiao, Y. Zheng, W. Tang, G. Yao, and L. Ruan, "Parallelizing dynamic time warping algorithm using prefix computations on gpu," in *(HPCC_EUC)*. IEEE, 2013, pp. 294–299.
[15] A. Ailamaki, "Databases and hardware: The beginning and sequel of a beautiful friendship," *VLDB*, 2015.
[16] T. Palpanas, "The parallel and distributed future of data series mining," in *HPCS*, 2017.
[17] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans, "Time series epenthesis: Clustering time series streams requires ignoring some data," in *ICDM*, 2011, pp. 547–556.
[18] J. Shieh and E. Keogh, "iSAX: disk-aware mining and indexing of massive time series datasets," *DMKD*, no. 1, 2009.
[19] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *CSUR*, 2009.
[20] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, M. B. Westover, and N. B. Shamlo, "A disk-aware algorithm for time series motif discovery," *DAMI*, vol. 22, no. 1-2, pp. 73–105, 2011.
[21] R. Agrawal, C. Faloutsos, and A. N. Swami, "Efficient similarity search in sequence databases," in *FODO*, 1993.
[22] A. Mueen, S. Nath, and J. Liu, "Fast approximate correlation for massive time-series data," in *SIGMOD*, 2010.
[23] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, 2011.
[24] B. Tang, M. L. Yiu, Y. Li *et al.*, "Exploit every cycle: Vectorized time series algorithms on modern commodity cpus," in *IMDM*, 2016.
[25] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *KIS*, 2001.
[26] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual," 2016.
[27] B.-K. Yi and C. Faloutsos, "Fast time sequence indexing for arbitrary lp norms." VLDB, 2000.
[28] "Incorporated Research Institutions for Seismology – Seismic Data Access," http://ds.iris.edu/data/access/, 2016.
[29] "Southwest university adult lifespan dataset (sald)," http://fcon_1000.projects.nitrc.org/indi/retro/sald.html, 2018.
[30] "Source code and datasets used in this paper," http://www.mi.parisdescartes.fr/~themisp/paris/, 2018.
[31] S. Kashyap and P. Karras, "Scalable knn search on vertically stored time series," in *SIGKDD*, 2011, pp. 1334–1342.
[32] C. Li, P. S. Yu, and V. Castelli, "Hierarchyscan: A hierarchical similarity search algorithm for databases of long sequences," in *ICDE*, 1996, pp. 546–553.
[33] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.
[34] I. Assent, R. Krieger, F. Afschari, and T. Seidl, "The ts-tree: efficient time series search and retrieval," in *EDBT*, 2008.
[35] J. Chou, K. Wu *et al.*, "Fastquery: A parallel indexing system for scientific data," in *CLUSTER*. IEEE, 2011, pp. 455–464.
[36] J. Zhou and K. A. Ross, "Implementing database operations using simd instructions," in *SIGMOD*. ACM, 2002.
[37] S. Zeuch, J. Freytag, and F. Huber, "Adapting tree structures for processing with SIMD instructions," in *EDBT*, 2014.
[38] P. Gepner and M. F. Kowalik, "Multi-core processors: New way to achieve high system performance," in *PAR ELEC*. IEEE, 2006, pp. 9–13.
[39] D. B. Lomet and F. Nawab, "High performance temporal indexing on modern hardware," in *ICDE*, 2015.
[40] M. G. Gowanlock and H. Casanova, "Distance threshold similarity searches: Efficient trajectory indexing on the GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, 2016.
[41] A. Mueen, H. Hamooni, and T. Estrada, "Time series join on subsequence correlation," in *ICDM*, 2014, pp. 450–459.
[42] P. Ghodsnia, I. T. Bowman, and A. Nica, "Parallel i/o aware query optimization," in *SIGMOD*. ACM, 2014, pp. 349–360.