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

Fast Data Series Indexing for In-Memory Data

Botao Peng · Panagiota Fatourou · Themis Palpanas

Received: date / Accepted: date

Abstract Data series similarity search is a core operation for several data series analysis applications across many different domains. However, the state-of-the-art techniques fail to deliver the time performance required for interactive exploration, or analysis of large data series collections. In this work, we propose MESSI, the first data series index designed for in-memory operation on modern hardware. Our index takes advantage of the modern hardware parallelization opportunities (i.e., SIMD instructions, multi-socket and multi-core architectures), in order to accelerate both index construction and similarity search processing times. Moreover, it benefits from a careful design in the setup and coordination of the parallel workers and data structures, so that it maximizes its performance for in-memory operations. MESSI supports similarity search using both the Euclidean and Dynamic Time Warping (DTW) distances. Our experiments with synthetic and real datasets demonstrate that overall MESSI is up to 4x faster at index construction, and up to 11x faster at query answering than the state-of-the-art parallel approach. MESSI is the first to answer exact similarity search queries on 100GB datasets in ∼50msec (30-75msec across diverse datasets), which enables real-time, interactive data exploration on very large data series collections.

B. Peng
Institute of Computing Technology, Chinese Academy of Sciences
E-mail: pengbotao@ict.ac.cn

P. Fatourou
FORTH ICS E-mail: faturu@csd.uoc.gr

T. Palpanas
LIPADE, Université de Paris & French University Institute (IUF)
E-mail: themis@mi.parisdescartes.fr

1 Introduction

Several applications across many diverse domains (e.g., finance, astrophysics, etc.), such as in finance, astrophysics, neuroscience, engineering, multimedia, and others [9, 59, 62, 91], continuously produce big collections of data series[1] which need to be processed and analyzed. Often times, this is part of an exploratory process, where users ask a query, review the results, and then decide what their subsequent queries, or analysis tasks should be. The most common type of query that different analysis applications need to answer on these collections of data series is similarity search [27, 28, 59], which is at the core of several data series analysis tasks, such as classification and anomaly detection [13–17, 27, 45, 54, 77].

The continued increase in the rate and volume of data series production with collections that grow to several terabytes in size [2,3,59,63], renders existing data series indexing technologies inadequate. For example, ADS+ [89], the state-of-the-art sequential (i.e., non-parallel) indexing technique, requires more than 2min to answer exactly a single 1-NN (Nearest Neighbor) query on a (moderately sized) 100GB sequence dataset.

Given the evolution of CPU performance, where the processor clock speed is not increasing due to the power wall constraint, algorithmic speedups can now mainly come by exploiting parallelism [7, 12, 31, 34, 55, 60, 69, 71, 78, 83, 87, 88]. This involves (i) parallelism across compute nodes (e.g., using Spark) [48, 85], where the main goal is to scale to datasets that cannot be easily handled by a single node, and (ii) parallelism inside a single compute node (e.g., ex-

[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, frequency in infrared spectroscopy, mass in mass spectroscopy, position in genome sequences, etc.).

ploiting the Multi-Socket and Multi-Core (MSMC) architectures) [64, 65, 67], where the main goal is to minimize latency.

In this study, we focus on parallelization inside a single node. The state-of-the-art approach, ParIS+ [67], is a disk-based data series parallel indexing scheme that exploits the parallelism capabilities provided by MSMC architectures. Experiments showed that ParIS+ answers queries 10x faster than ADS+, and more than 1000x faster than the optimized serial scan method. Still, ParIS+ is designed for disk-resident data and its performance is dominated by the I/O cost. For instance, ParIS+ answers a 1-NN (Nearest Neighbor) exact query on a 100GB dataset in 15sec, which is above the limit for keeping the user's attention (i.e., 10sec), and for supporting interactive analysis (i.e., 100msec) [29].

[Application Scenario] In this work, we focus on designing an efficient parallel indexing and query answering scheme for *in-memory* data series processing. Our work is motivated and inspired by the following real scenario. Airbus[2], currently stores petabytes of data series, describing the behavior over time of various aircraft components (e.g., the vibrations of the bearings in the engines), as well as that of pilots (e.g., the way they maneuver the plane through the fly-by-wire system) [36]. The experts need to access these data in order to run different analytics algorithms. However, these algorithms usually operate on a subset of the data (e.g., only the data relevant to landings from Air France pilots), which fit in memory. In order to perform complex analytics operations (such as searching for similar patterns, or classification) fast, in-memory data series indices must be built. Thus, the time cost of both index creation and query answering become important factors. Apart from engineering, similar needs appear in other domains and applications, as well [9, 62], such as astrophysics and neuroscience, where different, *adhoc* subsets of data need to be analyzed, and for which we need to build indexes and then perform similarity search operations.

[MESSI Approach] We present MESSI, an in-MEmory data SerieS Index that incorporates the state-of-the-art techniques in sequence indexing[3], and inherently takes advantage of modern hardware parallelization in order to accelerate processing times. MESSI supports similarity search queries on both z-normalized and non z-normalized data, using both the Euclidean and the Dynamic Time Warping (DTW) distance measures.

MESSI uses MSMC architectures in order to concurrently perform both index construction and query answering, and it exploits the Single Instruction Multiple Data (SIMD) capabilities of modern CPUs, in order to further parallelize the execution of individual instructions (mainly distance computations) inside each core. More importantly though, MESSI features a novel solution for answering exact 1-NN queries which is 6-11x faster than an in-memory version of ParIS+ across the datasets (of size 100GB) we tested, achieving for the first time interactive exact query answering times, at ∼50msec. It also provides redesigned algorithms that lead to a further ∼4x speedup in index construction time, in comparison to (in-memory) ParIS+.

The design decisions in ParIS+ were heavily influenced by the fact that the cost was mainly I/O bounded. Since MESSI copes with in-memory data series, no CPU cost can be hidden under I/O. Therefore, MESSI required more careful design choices and coordination of the parallel workers. This led to the development of a more subtle design for the index construction and new algorithms for answering similarity search queries on this index.

For query answering in particular, we showed that adaptations of alternative solutions, which have proven to perform the best in other settings (i.e., disk-resident data [67]), are not optimal in our case, so we designed a novel solution that achieves a good balance between the amount of communication among the parallel worker threads, and the effectiveness of each individual worker.

For instance, the new scheme uses concurrent priority queues for storing the data series that cannot be pruned, and for processing these series in order, starting from those whose iSAX representations have the smallest distance to the iSAX representation of the query data series. In this way, the parallel query answering threads achieve better pruning on the data series they process. Moreover, the new scheme uses the index tree to decide which data series to insert into the priority queues for further processing. In this way, the number of distance calculations performed between the iSAX summaries of the query and data series is significantly reduced (ParIS+ performs this calculation for all data series in the collection).

To achieve load balancing, we had to come up with a scheme where all priority queues had about the same number of elements. ParIS+ had to perform this calculation for the entire collection of data series. We also experimented with several designs for reducing the synchronization cost among different workers that access the priority queues and for achieving load balancing. We ended up with a scheme where workers use randomization to choose the priority queues they will work on. Consequently, MESSI answers exact 1-NN queries on 100GB datasets within 30-70msec across diverse synthetic and real datasets.

The index construction phase of MESSI differentiates from ParIS+ in several ways. For instance, ParIS+ was using a number of buffers to temporarily store pointers to the iSAX summaries of the raw data series before constructing the tree index [67]. MESSI allocates smaller such buffers per thread and stores in them the iSAX summaries themselves. In this way, it completely eliminates the synchro-

[2] `http://www.airbus.com/`

[3] A preliminary version of this work has appeared elsewhere [66].

nization cost in accessing the iSAX buffers. To achieve load balancing, MESSI splits the array storing the raw data series into small blocks, and assigns blocks to threads in a dynamic fashion. We applied the same technique when assigning to threads the buffers containing the iSAX summary of the data series. Overall, the new design and algorithms of MESSI led to $\sim$4x improvement in index construction time when compared to (in-memory) ParIS+. Still, the main contribution of the paper is our novel query answering scheme, which results in up to 11x better performance than ParIS+. This scheme supports similarity search on both Z-normalized and non Z-normalized data, and can be used with either the Euclidean, or the Dynamic Time Warping (DTW) distance.

[Contributions] Our contributions are summarized below.

- We propose MESSI, the first in-memory data series index designed for modern hardware, which can answer similarity search queries in a highly efficient manner.
- We implement a novel, tree-based exact query answering algorithm for both the Euclidean and Dynamic Time Warping (DTW) distances, which minimizes the number of distance calculations (both lower bound distance calculations for pruning true negatives, and real distance calculations for pruning false positives).
- We also design an index construction algorithm that effectively balances the workload among the index creation workers by using a parallel-friendly index framework with low synchronization cost.
- We provide proofs of correctness for our parallel algorithms. These proofs guarantee that both the index creation and query answering algorithms will always produce correct results.
- We conduct an experimental evaluation with several synthetic and real datasets, which demonstrates the efficiency of the proposed solution. The results show that MESSI is up to 4.2x faster at index construction and up to 11.2x faster at query answering than the state-of-the-art parallel index-based competitor, up to 109x faster at query answering than the state-of-the-art parallel serial scan algorithm, and thus can significantly reduce the execution time of complex analytics algorithms (e.g., *k-NN* classification by more than 1 order of magnitude).

[Paper Structure] The rest of this paper[4] is organized as follows. In Section 2, we provide the necessary background material. The MESSI approach is described in Section 3. Section 4 is the proof of correctness of our index creation and query answering algorithms. Section 5 contains our experimental analysis. We review the related work in Section 6, and conclude in Section 7.

2 Background

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

2.1 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 (in the case of time series, positions are expressed in terms of time, i.e., they are timestamps). We call n the *size*, or *length* of the data series. All discussions in this work are applicable to general high-dimensional vectors, too.

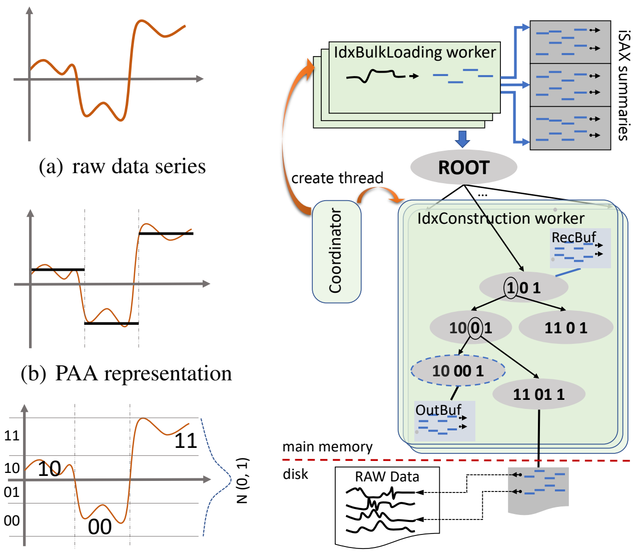
[Similarity Search] Analysts perform a wide range of data mining tasks on data series including clustering [41, 50, 73, 74], classification and deviation detection [19, 76], and frequent pattern mining [35, 57]. Existing algorithms for executing these tasks rely on performing fast similarity search across the different series. Thus, efficiently processing Nearest Neighbor (NN) queries is crucial for speeding up the above tasks. NN queries are defined as follows: given a query series S_q of length n, and a 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) [6] and Dynamic Time Warping (DTW) [72], which performs better for data mining tasks (e.g., classification [10]). 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 [58].

[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 [56]. 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.

In the data series context, SIMD has been employed for the computation of the Euclidean distance functions [78], as well as in the ParIS+ index, for the conditional branch calculations during the computation of the lower bound distances [67].

[4] A preliminary version of this paper has appeared elsewhere [66].



(a) raw data series

(b) PAA representation

(c) iSAX representation

(d) ParIS+ index

Fig. 1 The iSAX representation, and the ParIS+ index

2.2 iSAX Representation and the ParIS+ Index

[iSAX Representation] The iSAX representation (or summary) is based on the Piecewise Aggregate Approximation (PAA) representation [39], which divides the data series in w 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) illustrates an example of PAA representation with three segments (shown 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 [75] (and later used in several different data series indices [42, 52, 65, 76, 89]). 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: previous work has shown that iSAX achieves very good approximations with just 256 symbols, the maximum alphabet cardinality, $|alphabet|$ represented by 8 bits [18]. It then represents each of the w segments of the series not by the real value of the PAA, but 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).

Therefore, iSAX further reduces the size of the data series summarization, and more importantly it leads to a bitwise representation. Note that, even though the iSAX representation was invented several years ago, it remains one of the most popular data series summarization methods. Recent studies have shown that data series indices based on iSAX achieve state-of-the-art performance in various similarity search tasks [27, 28]. For an overview of iSAX-based indices, see [61].

[ParIS+ Index] Based on the iSAX representation, the ParIS+ index was developed [67], which proposed techniques and algorithms specifically designed for modern hardware and disk-based data.

ParIS+ makes use of variable cardinalities for the iSAX summaries (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 summaries); (ii) each inner node contains the iSAX summary of all the series below it, and has two children; and (iii) each leaf node contains the iSAX summaries of all the series inside it, and pointers to the raw data (in order to be able to prune false positives and produce exact, correct answers), which reside on disk. 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 summary 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 [18, 89]). The two refined iSAX summaries (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 distance of a query to a node is the distance between the query (raw values, or iSAX summary) and the node's iSAX summary.

In the index construction phase (see Figure 1(d)), ParIS+ uses a coordinator worker that reads raw data series from disk and transfers them into a raw data buffer in memory. A number of index bulk loading workers compute the iSAX summaries of these series, and insert <iSAX summary, file position> pairs in an array. They also insert a pointer to the appropriate element of this array in the receiving buffer of the corresponding subtree of the index root. When main memory is exhausted, the coordinator worker creates a number of index construction worker threads, each one assigned to one subtree of the root and responsible for further building that subtree (by processing the iSAX summaries stored in the corresponding receiving buffer). This process results in each iSAX summary being moved to the output buffer of the leaf it belongs to. When all iSAX summaries in the receiving buffer of an index construction worker have been processed, the output buffers of all leaves in that subtree are flushed to disk.

For query answering, ParIS+ offers a parallel implementation of the SIMS exact search algorithm [89]. It first computes an approximate answer by calculating the real distance between the query and the best candidate series, which is in the leaf with the smallest lower bound distance to the query. ParIS+ uses the index tree only for computing this approximate answer. Then, a number of lower bound calculation workers compute the lower bound distances between the query and the iSAX summary of each data series in the dataset, which are stored in the SAX array, and prune the series whose lower bound distance is larger than the approximate real distance computed earlier. The data series that are not pruned, are stored in a candidate list for further processing. Subsequently, a number of real distance calculation workers operate on different parts of this array to compute the real distances between the query and the series stored in it (for which the raw values need to be read from disk). For details see [67].

In the in-memory version of ParIS+, the raw data series are stored in an in-memory array. Thus, there is no need for a coordinator worker. The bulk loading workers operate directly on this array (split to as many chunks as the workers). In the rest of the paper, we use ParIS+ to refer to this in-memory version.

3 The MESSI Solution

The parallelism approach we employ in MESSI is governed by two main principles : 1) eliminate synchronization overheads as much as possible, and 2) balance the load of the index workers. These two principles often require contradicting design choices, so the design of MESSI is based on extensive experimentation to find the best compromise whenever needed.

We first outline the main ideas of MESSI. Figure 2 depicts the MESSI index construction and query answering pipeline. MESSI uses an index tree, comprised of several root subtrees. A number of index workers are responsible to construct the index tree. To avoid synchronization overheads and exploit locality, each subtree is built by a distinct worker. To achieve load balancing, workers are assigned subtrees on the fly, with different threads possibly processing different numbers of subtrees (depending on the work necessary on each subtree), so that they are all busy most of the time. To avoid synchronization overheads, workers are assigned to work on disjoint data subsets. This way workers never interfere with one another.

For query answering, workers traverse the tree pruning nodes whenever possible. MESSI uses a number of shared priority queues to store leaf nodes that are not pruned. For reducing the synchronization cost and exploit locality, different workers traverse different subtrees of the index tree. For ensuring load balancing, each worker adds elements in the queues in a round-robin fashion; this way, all queues end up having approximately the same number of elements. After the queues have been populated, the workers process the nodes in the priority queues. The priority of a queue node is its lower bound distance from the query series, so if a DeleteMin operation returns a node whose distance is larger than the current best distance, all nodes in the queue can be pruned (i.e., the worker gives up the entire queue). This scheme allows MESSI to perform additional pruning when processing queue nodes, and results in a reduced number of real-distance computations. In our implementation, more than one threads work on each queue, so that the real distance calculations on a node's series (which is a time-consuming task) overlaps with the deletion of additional nodes from the queue. However, we have chosen the number of threads to work on each queue with care to avoid high synchronization overheads.

3.1 Preliminaries

We proceed with the details of MESSI. The raw data are stored into the $RawData$ array, which is split into a predetermined number of chunks. A number, N_w, of *index worker* threads process the chunks to calculate the iSAX summaries of the raw data series they store. The number of chunks is not necessarily the same as N_w. Chunks are assigned to index workers the one after the other using Fetch and Increment (Fetch&Inc). Based on the iSAX representation, we compute in which subtree of the index an iSAX summary will be stored.

Each index worker stores the iSAX summaries it computes in the appropriate iSAX buffers. Each iSAX buffer is split into N_w parts and each worker works on its own part[5]. The number of iSAX buffers is usually a few tens of thousands and at most 2^w, where w is the number of segments in the iSAX summaries of each data series (w is fixed to 16 in this paper, as in previous studies [65, 89]).

When the iSAX summaries for all data series have been computed, the index workers proceed in the construction of the tree index. Each worker is assigned an iSAX buffer to work on (this is done again using Fetch&Inc). Each worker reads the data stored in (all parts of) its assigned buffer and builds the corresponding index subtree. Therefore, all index workers process distinct subtrees of the index, and work in parallel and independently from one another[6]. When an index worker finishes with the current iSAX buffer it works on, it continues with the next iSAX buffer that has not yet been processed.

When the series in all iSAX buffers have been processed, the tree index has been built and can be used to answer similarity search queries, as depicted in the query answering

[5] We also tried an alternative design, where buffers were not split, so many threads could try to update each element of a buffer concurrently. Therefore, each buffer had to be protected by a lock. This design resulted in worse performance due to the contention in accessing the iSAX buffers.

[6] Parallelizing the processing inside each one of the index root subtrees would require a lot of synchronization due to node splitting. When a node is split, two new leaf nodes are created and the data of the original leaf are moved to the new leaves.

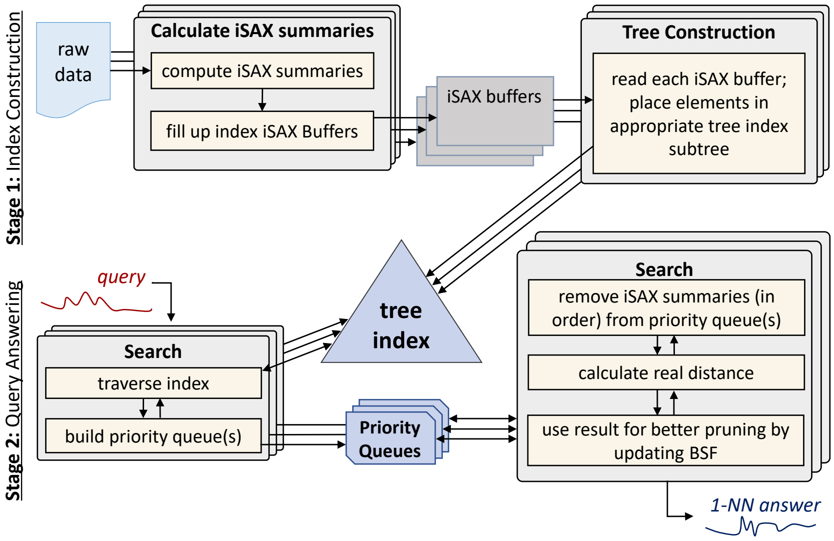

Stage 1: Index Construction
raw data
Calculate iSAX summaries
compute iSAX summaries
fill up index iSAX Buffers
iSAX buffers
Tree Construction
read each iSAX buffer; place elements in appropriate tree index subtree
Stage 2: Query Answering
query
tree index
Search
traverse index
build priority queue(s)
Priority Queues
Search
remove iSAX summaries (in order) from priority queue(s)
calculate real distance
use result for better pruning by updating BSF
1-NN answer

Fig. 2 MESSI index construction and query answering

phase of Figure 2. To answer a query, we first perform a search for the query iSAX summary in the tree index. This returns a leaf whose iSAX summary has the closest distance to the iSAX summary of the query. We calculate the real distance of the (raw) data series pointed to by the elements of this leaf to the query series, and store the minimum of these distances into a shared variable, called BSF (Best-So-Far). Then, the index workers start traversing the index subtrees (the one after the other) using BSF to decide which subtrees will be pruned. The leaves of the subtrees that cannot be pruned are placed into (a fixed number of) minimum priority queues, using the lower bound distance between the raw values of the query series and the iSAX summary of the leaf node, in order to be further examined. Each thread inserts elements in the priority queues in a round-robin fashion so that load balancing is achieved (i.e., all queues contain about the same number of elements).

As soon as the necessary elements have been placed in the priority queues, each index worker chooses a priority queue to work on, and repeatedly calls DeleteMin() on it to get a leaf node, on which it performs the following operations. It first checks whether the lower bound distance stored in the priority queue is larger than the current BSF: if it is then we are certain that the leaf node does not contain any series that can be part of the answer, and we can prune it; otherwise, the worker needs to examine the series contained in the leaf node, by first computing lower bound distances using the iSAX summaries, and if necessary also the real distances using the raw values. During this process, we may discover a series with a smaller distance to the query, in which case we also update the BSF. When a worker reaches a node whose distance is bigger than the BSF, it gives up this priority queue and starts working on another, since all other elements in the abandoned queue have a higher distance to the query. This process is repeated until all priority queues have been processed, and the BSF is updated along the way. At the end of the calculation, the value of BSF is returned as the query answer.

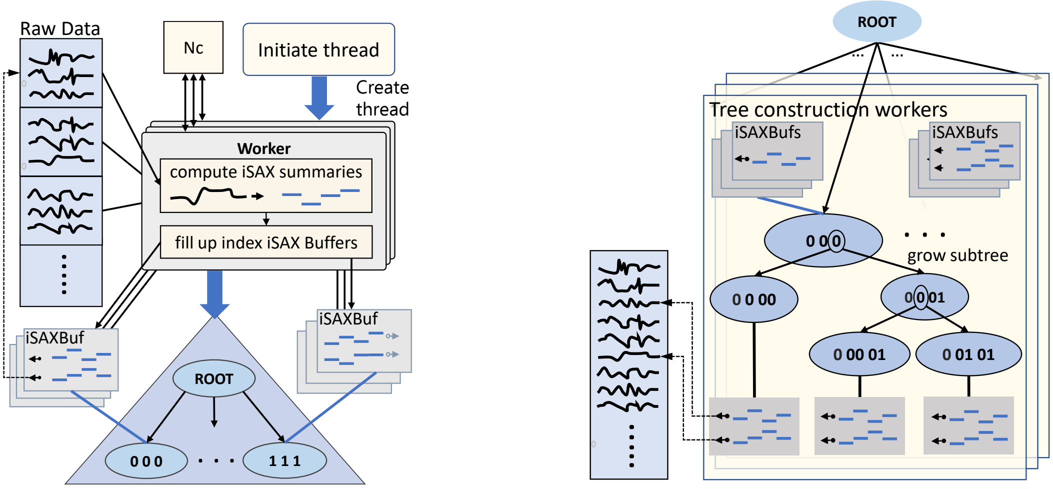
Algorithm 1: $CreateIndex$

```
Input: Index $index$, Integer $N_w$, Integer $chunk\_size$
1 for $i \leftarrow 0$ to $N_w - 1$ do
2     create a thread to execute an instance of
        IndexWorker($index$, $chunk\_size$,$i$, $N_w$);
3 end
4 wait for all these threads to finish their execution;
```

Note that, similarly to ParIS+, MESSI uses SIMD (Single-Instruction Multiple-Data) for calculating the distances of both the index iSAX summaries from the query iSAX summary (*lower bound distance calculations*), and the raw data series from the query data series (*real distance calculations*) [67].

3.2 Index Construction

Algorithm 1 presents the pseudocode for the *initiator* thread. The initiator creates N_w index worker threads to execute the index construction phase (line 2). As soon as these workers finish their execution, the initiator returns (line 4). We fix N_w to be 24 threads (Figure 11 in § 5 justifies this choice). We assume that the $index$ variable is a structure (struct) containing the $RawData$ array, all iSAX buffers, and a pointer to the root of the tree index. Recall that MESSI splits $RawData$ into chunks of size $chunk\_size$. We assume that the size of $RawData$ is a multiple of $chunk\_size$ (if not, standard padding techniques can be applied).



(a) CalculateiSAXSummaries (b) TreeConstruction

Fig. 3 Workflow and algorithms for MESSI index creation

Algorithm 2: $IndexWorker$

```
Input: Index $index$, Integer $chunk\_size$, Integer $pid$,
       Integer $N_w$
1 CalculateiSAXSummaries($index$, $chunk\_size$,$pid$);
2 barrier to synchronize the IndexWorkers with one another;
3 TreeConstruction($index$, $N_w$);
4 exit();
```

Algorithm 3: $CalculateiSAXSummaries$

```
   Input: Index $index$, Integer $chunk\_size$, Integer $pid$
1  Shared integer $F_c = 0$;
2  while (TRUE) do
3      $b \leftarrow$ Atomically fetch and increment $F_c$;
4      $b = b * chunk\_size$;
5      if ($b \geq$ size of the $index.RawData$ array) then break ;
6      for $j \leftarrow b$ to $b + chunk\_size$ do
7          $isax = ConvertToiSAX(index.RawData[j])$;
8          $\ell$ = find appropriate root subtree where $isax$ must be
             stored;
9          $index.iSAXbuf[\ell][pid] = \langle isax, j \rangle$;
10     end
11 end
```

The pseudocode for the index workers is in Algorithm 2. The workers first call the $CalculateiSAXSummaries$ function (line 1) to calculate the iSAX summaries of the raw data series and store them in the appropriate iSAX buffers. As soon as the iSAX summaries of all the raw data series have been computed (line 2), the workers call $TreeConstruction$ to construct the index tree.

The pseudocode of $CalculateiSAXSummaries$ is shown in Algorithm 3 and is schematically illustrated in Figure 3(a). Each index worker repeatedly does the following. It first performs a Fetch&Inc to get assigned a chunk of raw data series to work on (line 3). Then, it calculates the offset in the $RawData$ array that this chunk resides (line 4) and starts processing the relevant data series (line 6). For each of them, it computes its iSAX summary by calling the ConvertToiSAX function (line 7), and stores the result in the appropriate iSAX buffer of $index$ (lines 8-9). Recall that each iSAX buffer is split into N_w parts, one for each thread; therefore, $index.iSAXbuffer$ is a two dimensional array.

Each part of an iSAX buffer is allocated dynamically when the first element to be stored in it is produced. The size of each part has an initial small value (5 series in this work, as we discuss in the experimental evaluation) and it is adjusted dynamically based on how many elements are inserted in it (by doubling its size each time). We note that we also tried a design of MESSI with no iSAX buffers, but this led to slower performance (due to the worse cache locality). Thus, we do not discuss this alternative further.

As soon as the computation of the iSAX summaries is over, each index worker starts executing the $TreeConstruction$ function. Algorithm 4 shows the pseudocode for this function and Figure 3(b) schematically describes how it works. In $TreeConstruction$, a worker repeatedly executes the following actions. It accesses F_b (using Fetch&Inc) to get assigned an iSAX buffer to work on (line 3). Then, it traverses all parts of the assigned buffer (lines 5-6) and inserts every pair $\langle$iSAX summary, pointer to relevant data series$\rangle$ stored there in the index tree (line 7-12). Recall that the iSAX summaries contained in the same iSAX buffer will be stored in the same subtree of the index tree. So, no synchronization is needed among the index workers during this process. If a tree worker finishes its work on a subtree, a new iSAX buffer is (repeatedly) assigned to it, until all iSAX buffers have been processed.

Algorithm 4: $TreeConstruction$

```
Input: Index $index$, Integer $N_w$
Shared integer $F_b = 0$;
while (TRUE) do
    $b$ ←Atomically fetch and increment $F_b$;
    if ($b \geq 2^w$) then break ;        // root has <= $2^w$ children
    for $j \leftarrow 0$ to $N_w$ do
        for every $\langle isax, pos \rangle$ pair $\in index.iSAXbuf[b][j]$ do
            $targetLeaf$ ← Leaf of $index$ tree to insert $\langle isax, pos \rangle$;
            while $targetLeaf$ is full do
                SplitNode($targetLeaf$);
                $targetLeaf$ ← New leaf to insert $\langle isax, pos \rangle$;
            end
            Insert $\langle isax, pos \rangle$ in $targetLeaf$;
        end
    end
end
```

Algorithm 5: $ExactSearch$

```
1  Shared float $BSF$;
   Input: QuerySeries $QDS$, Index $index$, Integer $N_q$
2  QDS_iSAX = calculate iSAX summary for QDS;
3  BSF = approxSearch($QDS\_iSAX$, $index$);
4  for $i \leftarrow 0$ to $N_q - 1$ do
5      $queue[i]$ = Initialize the $i$th priority queue;
6  end
7  for $i \leftarrow 0$ to $N_s - 1$ do
8      create a thread to execute an instance of
         SearchWorker($QDS$, $index$, $queue[]$, $i$, $N_q$);
9  end
10 Wait for all threads to finish;
11 return ($BSF$);
```

3.3 Query Answering with Euclidean Distance

The pseudocode for executing an exact search query with Euclidean distance is shown in Algorithm 5. We first calculate the iSAX summary of the query (line 2), and execute an approximate search (line 3) to find the initial value of BSF, i.e., a first upper bound on the actual distance between the query and the series indexed by the tree. This process is illustrated in Figure 4(a).

During a search query, the index tree is traversed and the distance of the iSAX summary of each of the visited nodes to the iSAX summary of the query is calculated. If the distance of the iSAX summary of a node, nd, to the query iSAX summary is higher than BSF, then we are certain that the distances of all data series indexed by the subtree rooted at nd are higher than BSF. So, the entire subtree can be pruned. Otherwise, we go down the subtree, and the leaves with a distance to the query smaller than the BSF, are inserted in the priority queue.

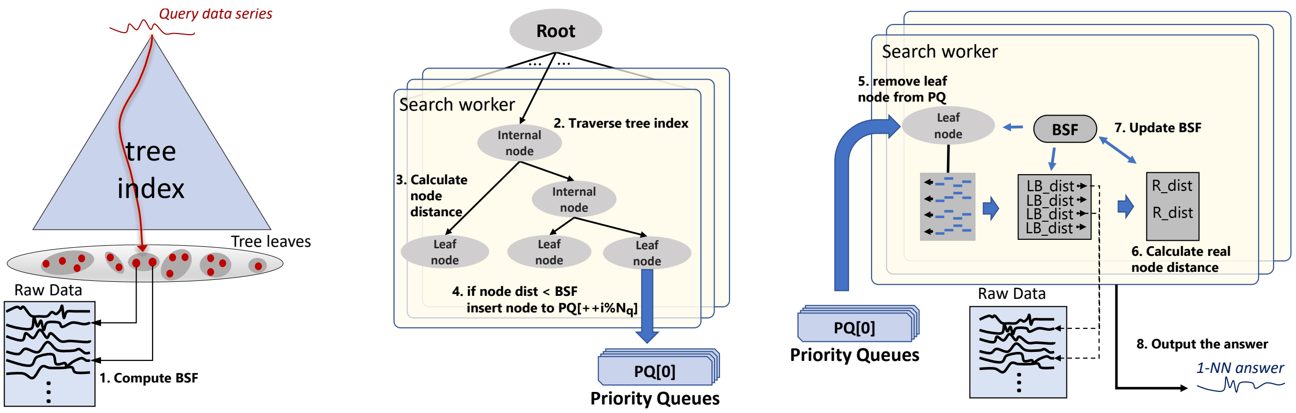
The technique of using priority queues maximizes the pruning degree, thus resulting in a relatively small number of raw data series whose real distance to the query series must be calculated. As a side effect, BSF converges fast to the correct value. Thus, the number of iSAX summaries that are tested against the iSAX summary of the query series is also reduced.

Algorithm 5 creates $N_s = 48$ threads, called the *search workers* (lines 7-9), which perform the computation described above by calling $SearchWorker$. It also creates $N_q \geq 1$ priority queues (lines 4-6), where the search workers place those data series that are potential candidates for real distance calculation. After all search workers have finished (line 10), $ExactSearch$ returns the current value of BSF (line 11).

We have experimented with two different settings regarding the number of priority queues, N_q, that the search workers use. The first, called *Single Queue* (SQ), refers to $N_q = 1$, whereas the second focuses in the Multiple-Queue (MQ) case where $N_q > 1$. Using a single shared queue imposes a high synchronization overhead, whereas using a local queue per thread results in load imbalance, since, depending on the workload, the size of the different queues may vary significantly. Thus, we choose to use N_q shared queues, where N_q is a fixed number (in our analysis $N_q = 24$, as experiments showed that this is the best choice).

The pseudocode of search workers is shown in Algorithm 6, and the work they perform in Figures 4(b) and 4(c). At each point in time, each thread works on a single queue. Initially, each queue is shared by two threads. Each search worker first identifies the queue where it will perform its first insertion (line 2). Then, it repeatedly chooses (using Fetch&Inc) a root subtree of the index tree to work on by calling $TraverseRootSubtree$ (line 6). After all root subtrees have been processed (line 8), it repeatedly chooses a priority queue (lines 10, 16) and works on it by calling $ProcessQueue$ (line 11). Each element of the $queue$ array has a $finished$ field indicating if the processing of the corresponding priority queue has finished. As soon as a search worker determines that all priority queues have been processed (line 13), it terminates.

We continue to describe the pseudocode for $TraverseRootSubtree$, which is presented in Algorithm 7 and illustrated in Figure 4(b). $TraverseRootSubtree$ is recursive. On each internal node, nd, it checks whether the (lower bound) distance of the iSAX summary of nd to the raw values of the query (line 1) is smaller than the current BSF, and if it is, it examines the two subtrees of the node using recursion (lines 11-12). If the traversed node is a leaf node and its distance to the iSAX summary of the query series is smaller than the current BSF (lines 4-9), it places it in the appropriate priority queue (line 6). Recall that the priority queues are accessed in a round-robin fashion (line 9). This strategy maintains the size of the queues balanced, and



(a) Approximate search and first BSF

(b) Tree traversal and priority queues

(c) Node distance calculation from priority queues

Fig. 4 Workflow and algorithms for MESSI query answering

Algorithm 6: $SearchWorker$

```
   Input: QuerySeries $QDS$, Index $index$, Queue $queue[]$,
          Integer $pid$, Integer $N_q$
1  Shared integer $N_b = 0$;
2  $q = pid \mod N_q$;
3  while (TRUE) do
4      $i \leftarrow$Atomically fetch and increment $N_b$;
5      if ($i \geq 2^w$) then break;
6      $TraverseRootSubtree(QDS, index.rootnode[i],$
         $queue[], \&q, N_q)$;
7  end
8  Barrier to synchronize the search workers with one another;
9  $q = pid \mod N_q$;
10 while (true) do
11     $ProcessQueue(QDS, index, queue[q])$;
12     if all queue[].finished=true then
13         break;
14     end
15     $q \leftarrow$ index such that $queue[q]$ has not been processed yet;
16 end
```

Algorithm 7: $TraverseRootSubtree$

```
   Input: QuerySeries $QDS$, Node $node$, queue $queue[]$,
          Integer $*pq$, Integer $N_q$
1  $nodedist$ = FindDist($QDS$, $node$);
2  if $nodedist > BSF$ then
3      break;
4  else if node is a leaf then
5      acquire $queue[*pq]$ lock;
6      Put node in $queue[*pq]$ with priority $nodedist$;
7      release $queue[*pq]$ lock;
8      // next time, insert in the subsequent queue
9      $*pq \leftarrow (*pq + 1) \mod N_q$;
10 else
11     TraverseRootSubtree($node.leftChild, queue[], pq, N_q$);
12     TraverseRootSubtree($node.rightChild, queue[], pq, N_q$)
13 end
```

reduces the synchronization cost of node insertions to the queues. We implement this strategy by (1) passing a pointer to the local variable q of $SearchWorker$ as an argument to $TraverseRootSubtree$, (2) using the current value of q for choosing the next queue to perform an insertion (line 6), and (3) updating the value of q (line 9). Each queue may be accessed by more than one threads, so a lock per queue is used to protect its concurrent access by multiple threads.

We next describe how $ProcessQueue$ works (see Algorithm 8 and Figure 4(c)). The search worker repeatedly removes the (leaf) node, nd, with the highest priority from the priority queue, and checks whether the corresponding distance stored in the queue is still less than the BSF. We do so, because the BSF may have changed since the time that the leaf node was inserted in the priority queue. If the distance is less than the BSF, then $CalculateRealDistance$ (line 9) is called to identify if any series in the leaf node (pointed to by nd) has a real distance to the query that is smaller than the current BSF. If we discover such a series (line 10), BSF is updated to the new value (line 13). We use a lock to protect BSF from concurrent update efforts (lines 11, 15). Previous experiments showed that the initial value of BSF is very close to its final value [32, 33]. Indeed, in our experiments, the BSF is updated only 10-12 times (on average) per query. So, the synchronization cost for updating the BSF is negligible.

$CalculateRealDistance$ is shown in Algorithm 9. Note that both the lower bounding (line 2) and the real (line 3) distance calculations use SIMD [65]. However, this does not lead to the same significant impact in performance as in ParIS+. This is because pruning is much more effective in MESSI for two reasons: (i) MESSI performs much less lower bounding distance calculations since many of them are pruned during the traversal of the tree (see Algorithm 7); (ii) MESSI also performs a smaller number of real distance calculations since examining the raw data series in the order

Algorithm 8: $ProcessQueue$

```
   Input: QuerySeries $QDS$, Index $index$, Queue $Q$
1  while TRUE) do
2      acquire Q's lock;
3      $node$ = DeleteMin($Q$);
4      release Q's lock;
5      if $node$ == $NULL$ then
6          return;
7      end
8      if $node.dist < BSF$ then
9          $realDist$ = CalculateRealDistance($QDS$, $index$, $node$);
10         if $realDist < BSF$ then
11             acquire $BSFLock$;
12             if $realDist < BSF$ then
13                 $BSF$ = $realDist$;
14             end
15             release $BSFLock$;
16         end
17     else
18         $Q.finished$ = true;
19         return;
20     end
21 end
```

Algorithm 9: $CalculateRealDistance$

```
Input: QuerySeries $QDS$, Index $index$, node $node$, float $BSF$
for every (isax, pos) pair $\in$ node do
    if $LowerBound\_SIMD(QDS, isax) < BSF$ then
        $dist = RealDist\_SIMD(index.RawData[pos], QDS)$;
        if $dist < BSF$ then
            $BSF = dist$;
        end
    end
end
return $(BSF)$
```

defined by the priority queue (see Algorithm 8), rather than in the order of the raw file that ParIS+ uses, means that the *BSF* gets updated earlier and converges earlier to the value of the nearest neighbor, leading to better pruning.

3.4 Query Answering with Dynamic Time Warping

Not only can MESSI accelerate similarity search based on Euclidean distance, but it also can be used to perform similarity search using the Dynamic Time Warping (DTW) distance. No changes are required in the index structure for this: the index we build can answer both Euclidean and DTW similarity search queries. Supporting DTW queries requires modifying the query answering algorithm only, and using LB_Keogh [40], which is a tight lower bound of the DTW distance[7]. Recall that a lower bound for the DTW distance between the query and a candidate series can be computed by considering the distances between the corresponding points of the candidate series and the points of the LB_Keogh envelope of the query (see Figure 5; if some points of the candidate fall inside the query envelope, then their distance is zero).

Assuming the reach, or allowed range of (the constrained) warping, is r, we define two new sequences, U and L, corresponding to the upper and lower parts of the LB_Keogh envelope:

$$U_i = \max\left(q_{i-r} : q_{i+r}\right)$$
$$L_i = \min\left(q_{i-r} : q_{i+r}\right)$$

Having defined U and L, we now use them to define a lower bounding measure for DTW between a query sequence Q and a candidate answer C [40]:

$$LB\_Keogh(Q, C) = \sqrt{\sum_{i=1}^{n} \begin{cases} (c_i - U_i)^2 & if\ c_i > U_i \\ (c_i - L_i)^2 & if\ c_i < L_i \\ 0 & otherwise \end{cases}}$$

Intuitively, when the query series arrives, we compute the LB_Keogh envelope of this series, as shown in Figure 5. We then probe the index using the envelope as the query (instead of the series itself). The distances we compute using the LB_Keogh envelope are guaranteed to be lower bounds of the true DTW distances [40]. Therefore, this operation correctly prunes the search space, and (as in the case of Euclidean distance) we then simply need to remove the false positives by computing the DTW distance on the raw data values of the (small set of) candidate answers. Overall, the process of query answering using DTW follows the same steps as those described in Algorithms 5-8, except that in line 2 of Algorithm 5 we compute the PAA of the LB_Keogh envelope of the query, in line 1 of Algorithm 7 we compute the (lower bounding) distance between the query envelope PAA and the node iSAX summarization, and in line 9 of Algorithm 8 we call the function that computes the DTW real distance (Algorithm 10).

More specifically, we first perform an approximate search in order to get an initial solution that is close to the actual answer, which will serve as our BSF. In order to prune the index tree, we then calculate the lower bound distance between the query envelope PAA representation and the iSAX summarization of the leaf nodes (see Figure 6 *top*). We insert the leaves that we cannot prune (i.e., the DTW lower bound is less than the BSF) in the priority queue. When MESSI pops a leaf from the priority queue, it calculates the DTW lower bound distance between the query envelope PAA representation and the iSAX summarization of each series in the leaf (Algorithm 10, line 2). For the series that survive this second filter, we have to access the raw data. We start by computing the DTW lower bound distance between the

[7] We note that other lower bounds for DTW can be used as well, such as LB_Improved [47]. Even though LB_Improved can produce tighter bounds, in our experiments it also resulted in higher query answering times due to the additional computations it involves.

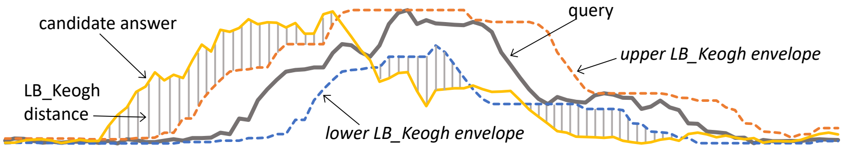
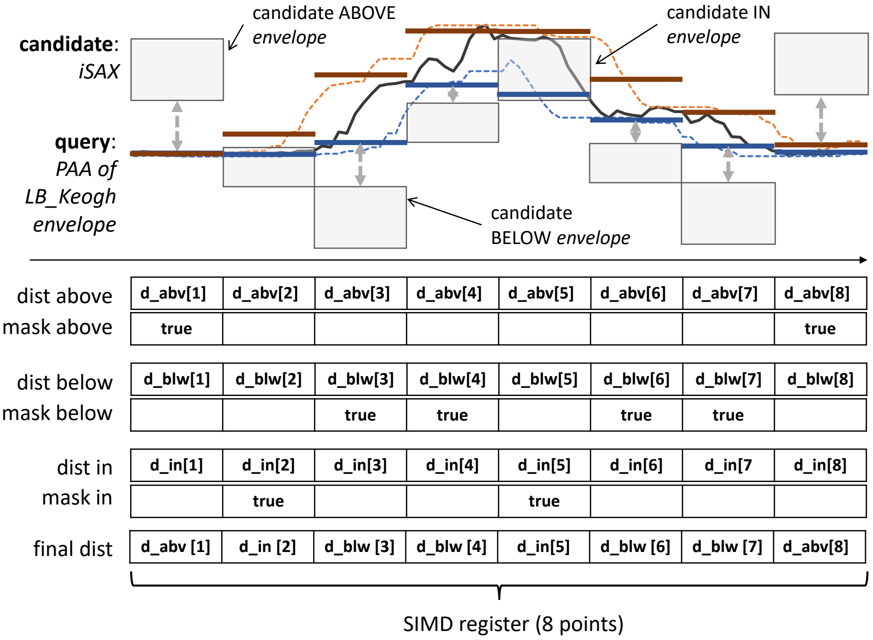


Fig. 5 Envelope (dotted lines representing the upper, U, and lower, L, series that define the envelope) of query Q (thick solid black line) with warping size 10%. Vertical lines represent LB_Keogh lower bound between query envelope and candidate answer C (thin solid yellow line).



dist above	d_abv[1]	d_abv[2]	d_abv[3]	d_abv[4]	d_abv[5]	d_abv[6]	d_abv[7]	d_abv[8]
mask above	true							true
dist below	d_blw[1]	d_blw[2]	d_blw[3]	d_blw[4]	d_blw[5]	d_blw[6]	d_blw[7]	d_blw[8]
mask below			true	true		true	true	
dist in	d_in[1]	d_in[2]	d_in[3]	d_in[4]	d_in[5]	d_in[6]	d_in[7	d_in[8]
mask in		true			true			
final dist	d_abv [1]	d_in [2]	d_blw [3]	d_blw [4]	d_in[5]	d_blw [6]	d_blw [7]	d_abv[8]

Fig. 6 SIMD conditional branch DTW lower bound distance calculation.

```
Algorithm 10: CalculateRealDistanceDTW
    Input: QuerySeries $QDS$, Envelope of QuerySeries
           $EQDS$, Index $index$, node $node$, float $BSF$
1   for every (isax, pos) pair $\in$ node do
2       if $LowerBound\_DTW\_SIMD(EQDS, isax) < BSF$ then
3           if $LB\_Keogh(EQDS, index.RawData[pos]) < BSF$ then
4               $dist = RealDist\_SIMD(index.RawData[pos], QDS)$;
5               if $dist < BSF$ then
6                   $BSF = dist$;
7               end
8           end
9       end
10  end
11  return $(BSF)$
```

query envelope raw values and the raw values of the series (Algorithm 10, line 3). If this step cannot prune the series, either, then we finally compute the true DTW distance between the raw values of the query and the series (Algorithm 10, line 4).

In order to speed up the execution of the DTW lower bound distance calculations, we develop a SIMD solution. Note that, in contrast to the simple case of a Euclidean distance calculation on the real data, developing a SIMD solution for the LB_Keogh lower bound is not straight-forward. The Euclidean distance calculated on the real values of two series involves exactly the same operations (i.e., first a subtraction and then a power of two operation) for all the points of the series. This leads to a simple SIMD solution, where the entire SIMD register performs the same operations, all useful and necessary for the final result. On the other hand, the algorithm for computing the DTW lower bound involves branching. As we discuss below, we need to perform different operations for the candidate series segments, depending on whether their values are larger, smaller, or lie within the the LB_Keogh envelope values. We therefore need to translate these branches of the operation into an efficient SIMD implementation.

Our DTW lower bound SIMD solution is illustrated in Figure 6. In the top part of this figure, boxes represent the iSAX summary of each segment of the candidate series, and red and blue horizontal lines represent the PAA representations of each segment of the upper and the lower LB_Keogh envelope of the query, respectively. The bottom part of Figure 6 illustrates how the lower bound DTW distance between the iSAX summary of the candidate series and the PAA representations of the query LB_Keogh envelopes, is calculated using SIMD. Our architecture supports the computation of the lower bound DTW distances for eight of these segments concurrently.

This per-segment computation needs to capture three different cases, namely, the PAA representation of the candidate being *ABOVE*, *BELOW*, or *IN* the query envelope PAA representation. The code to run is different in each of these cases. The ABOVE distance is calculated between the lower edge of the candidate series segment's iSAX box and the corresponding segment of the PAA representation of the *Upper* LB_Keogh envelope (shown as a red line in Figure 6). The BELOW distance is calculated between the upper edge of the iSAX box of the candidate series and the the PAA representation of the *Lower* LB_Keogh envelope (shown as a blue line in Figure 6). Finally, the IN distance is simply zero.

Since we cannot know beforehand which of the above three cases is true in each case, we use SIMD to calculate all three distances for each segment. These are depicted in Figure 6 as *dist above*, *dist below*, and *dist in*. We then choose the correct distance among these three distances. To do so, we use SIMD to compute three masks, setting the appropriate positions to *true* depending on the observed situation, that is, depending on the position (*ABOVE*, *BELOW*, or *IN*) of a candidate series' segment with respect to the corresponding segments of the LB_Keogh envelope. In Figure 6 for example, the first candidate iSAX representation is above the corresponding query envelope PAA, which means that only the *ABOVE* mask will be *true* for this position; consequently we will choose the ABOVE distance value for this position of the SIMD vector.

The final result is then computed using SIMD by summing up the right distances for each segment, i.e., those for which the corresponding mask position was set to *true* (shown as *final dist* in Figure 6 bottom). This operation is efficiently executed by using the appropriate SIMD instructions (AVX, AVX2 and SSE3) [24].

3.5 Complexity Analysis

We now provide a best- and worst-case time analysis, which contrasts the time needed in a concurrent setting with that required in a single-thread environment. That is, we compare the performance of MESSI when multiple workers are active with its performance when just a single thread is active. Note that the best- and worst-case scenaria are mainly (but not exclusively) driven by the data characteristics: as we detail below, different datasets may lead to index trees with significantly different properties.

[Index Construction] Index construction in MESSI is comprised of two phases. During the first phase (Algorithm 3, Figure 3(a)), the index workers calculate the iSAX summaries of the raw data series and store them into the iSAX buffers. During the second phase (Algorithm 4, Figure 3(b)), the index workers process the iSAX buffers and build the index tree. We analyze each of these phases separately. Assume that the time needed by a single thread to execute phase 1 is T_1, the time needed to execute phase 2 is T_2, and the total sequential time for index creation is $T = T_1 + T_2$. In MESSI, every index worker processes about the same number of data series. Note that processing data series of the same length takes the same amount of time. Given that each index worker works on its own part of the iSAX buffers, the amount of time spent to allocate the buffers in the concurrent setting is at most a factor of 2 larger than that needed in the sequential case. Therefore, if we exclude the contention that a worker experiences when accessing the Fetch&Increment object, all workers require about the same amount of time to finish the first phase.

Best Case: The best case scenario occurs when threads experience no contention in accessing the Fetch&Inc objects (i.e., no two threads access a Fetch&Inc object at exactly the same time). Since the threads do not experience any contention when storing elements in the iSAX buffers either, MESSI then requires $O(T_1/N_w)$ time for executing the first phase, which is optimal. We now focus on the second phase. In the best case scenario, it additionally holds that the iSAX buffers are assigned to threads in a way that all threads finish the second stage at about the same time. Note that this does not necessarily require that all subtrees contain the same number of nodes (e.g., if bigger subtrees are processed earlier than smaller ones, then the processing of big subtrees overlaps with that of smaller ones). Then, MESSI requires $O(T_2/N_w)$ time to execute phase 2, which is also optimal. Therefore, in the best case, MESSI exhibits optimal speedup for index creation.

Worst Case: In the worst case, all N_w workers access the Fetch&Inc objects at the same time. We assume that the system serializes these accesses and the ith worker in this serialization order will get a response from the object after i time units. Therefore, each block of N_w concurrent accesses to the Fetch&Inc object (by all threads) adds a total of N_w time units due to contention. We will have $(N_c + N_b)/N_w$ such blocks of accesses, where N_c is the number of chunks and $N_b \leq 2^w$ is the number of root subtrees in the index tree. In the worst case, MESSI will also experience the following: while a thread will be processing the last chunk of the raw data array, all other threads will be sitting idle. The same will occur with the last subtree. So, if T_c is the sequential time for processing a chunk and T_s is the sequential time for processing the biggest subtree of the root, the worst case execution time of MESSI will be $O((T - T_c - T_s)/N_w +$

$T_c + T_s + N_c + N_b)$. In the extreme scenario, where all data series are stored in only one of the subtrees, this time may be no less than T (the sequential time). Note that this is a pathological case that would happen when all series in the dataset are very similar to one another[8].

[Query Answering] Query answering in MESSI is comprised of three phases. During the first phase, approximate search is executed. During the second phase (*tree traversal*), the search workers traverse the index tree and populate the priority queues. During the third phase (*queue processing*), the search workers process the elements of the priority queues to produce the final result. We analyze each of these phases separately. Assume that the time needed by a single thread to execute phase 1 is T_1, the time needed to execute phase 2 is T_2, the time needed to execute phase 3 is T_3, and the total sequential time for the last two phases is $T = T_2 + T_3$. The approximate search is executed by a single thread in MESSI and therefore this time is also T_1 in a multi-threaded environment. We therefore focus on the other two phases.

Best case. For the tree traversal phase, the best case occurs when threads never access the Fetch&Inc object concurrently and never find the lock of a queue taken. Moreover, each thread must add the same number of nodes in the priority queues, so that all threads perform about the same amount of work. Then, the time needed to execute phase 2 is $O(T_2/N_s)$. For phase 3, the best case occurs when no thread has to ever wait on a lock and each thread performs about the same amount of computation. Note that after acquiring a node from some queue, a thread has to perform computation (i.e., real distance calculations). These computations could be overlapping with the deletion of additional elements from the queue. Thus, the time needed for phase 3 in the best case is $O(T_3/N_s)$. Therefore, the total time is $O(T_1 + T/N_s)$.

We observe that in the concurrent case, it may happen that the final value of BSF is reached faster than in the sequential case, since all threads update the value of BSF in parallel. This may result in better pruning than in the single-thread case, where the thread may process the subtree (or the queue) that contains the node which results in the final value of BSF towards the end of the tree traversal (or the processing of the queues).

Worst case. Let T_a be the sequential time needed for performing the insertions to the priority queues. Since both the cost for an insertion and the cost for a deletion are logarithmic on the size of the priority queue, the sequential time needed for performing the deletions from the priority queue is also in $\Theta(T_a)$. Let T_b be the sequential time needed for updating the BSF. Due to the use of locks, these times are still sequential in the concurrent setting. Assuming queue locks [37], the steps needed to acquire or release a lock is $O(1)$. Note that the time a thread waits for the lock to be released is overlapping with the critical sections of other threads, and therefore we do not count waiting times on the locks.

Regarding the second phase, in the worst case, all N_s workers access the Fetch&Inc object at the same time. Thus, each block of N_s concurrent accesses to the Fetch&Inc object adds a total of N_s time units (due to contention). We have N_b/N_s such blocks of accesses, thus resulting in a total cost of $O(N_b)$ time units. Therefore, the worst-case time for the second phase is $O((T_2 - T_a)/N_s + T_a + N_b)$. Since $T_b \in O(T_a)$, the time to execute the third phase is $O((T_3 - T_a - T_d)/N_s + T_a + T_d)$. Therefore, the total worst-case time is $O(T_1 + (T - T_a - T_d)/N_s + T_a + T_d + N_b)$.

4 Proof of Correctness

We note that in concurrent algorithms, the non-deterministic nature of parallel execution may lead to errors that are not detected during testing. In this section, we provide proofs that the proposed algorithms for index creation and query answering always produce correct results, irrespective of the peculiarities of parallel execution.

[Index construction phase] MESSI builds the tree index with minor synchronization, i.e., by using two Fetch&Inc objects and a barrier. This makes the correctness proof for index creation relatively simple. However, for completeness, we include it below.

We say that a data series S of the *RawData* array *is processed* whenever a thread calculates its iSAX summary (i.e., executes line 7 of Algorithm 3). We say that a chunk of the *RawData* array *is processed* if a thread processes data series stored in it.

The use of the Fetch&Inc object, F_c, in Algorithm 3, ensures that for each $0 \leq i \leq size/chunk\_size$, (where $size$ is the size of the *RawData* array and $chunk\_size$ is the size of each of its chunks), there exists exactly one thread p that gets number i when accessing F_c (line 2 of Algorithm 3), and no thread other than p processes chunk i. By inspection of the code of Algorithm 3 (lines 2 and 4 and condition of the *if* statement of line 5), it follows that F_c is accessed until its value becomes as large as the number of chunks of the *RawData* array. Therefore, for every chunk of the *RawData* array, there is exactly one thread to which this chunk is assigned. Line 6 ensures that once a chunk is assigned to a thread, all the data series it contains are processed by this thread. These (and the pseudocode) imply the following:

Lemma 1 *For every data series, S, contained in the* RawData *array, the following hold: (1) S is processed exactly once, i.e. there is a single thread p that calculates the iSAX summary for S; (2) there exists exactly one iSAX buffer that*

[8] In such a case, indexing and similarity search would not be useful anyways.

contains an entry e corresponding to S, and this entry appears in the part of the buffer that is assigned to p.

We use Lemma 1 to argue that the constructed tree index is correct.

Theorem 1 *The data structure T constructed by executing the IndexConstruction phase (Algorithms 3 and 4) is a tree that contains a distinct element for every data series of the* RawData *array and no more elements.*

Proof Initially, T is a tree with a root node and $c \leq 2^w$ leaf children. By inspection of the code, additional elements can be added into T by executing Algorithm 4. The barrier on line 2 of Algorithm 2 ensures that no thread starts inserting additional elements in T, as long as there exist threads that still process data series stored in the *RawData* array (i.e., they still execute Algorithm 3). Thus, Lemma 1 implies that a thread calls Algorithm 4 only after the iSAX summaries of all data series stored in the *RawData* array have been placed in the iSAX buffers.

Recall that the number of iSAX buffers is also c (the same as the number of the root children of T). The use of the Fetch&Increment object, F_b, and lines 2 and 4 ensure that for each i, $0 \leq i \leq c$, exactly one thread p gets number i by accessing F_b (i.e., by executing line 3 of Algorithm 4). Recall that every calculated iSAX summary is placed in the appropriate iSAX buffer, i.e., in the iSAX buffer that corresponds to the root subtree of T in which the iSAX summary should be stored. Thus, iSAX buffer i contains only those data series that are to be stored in T's root subtree numbered i. It follows that the task to build the entire subtree has been assigned solely to process i. So, different threads work on different subtrees of T (and no synchronization is needed between them). It follows that T ends up to be an index tree.

Lines 2-4 ensure that all c iSAX buffers will be examined. The for loop of line 5 ensures that p will examine all parts of iSAX buffer i, and the for loop of line 6 guarantees that all iSAX summaries stored in each of these parts will be inserted in T (lines 7-12). Therefore, Lemma 1 (claim 2) implies that the constructed tree contains a distinct element for every data series stored in the $RawData$ array and no more elements.

Query Answering Phase. To argue that the response of a 1-NN query, QR, is correct, we need the following properties from [75].

Property 1 The distance between the PAA of QR and the iSAX summary of a node nd of the index lower bounds the real distance between QR and any of the series in the leaves of nd's subtree.

Property 2 Consider two leaf nodes nd and nd' of the index tree. Let d be the minimum real distance between QR and any series in nd. If d is smaller than the distance between the PAA of QR and the iSAX summary of nd', then all real distances between QR and every series in nd' are greater than d.

Lemma 2 *$TraverseRootSubtree$ is invoked exactly once for each of the root subtrees of the index tree.*

Proof The use of the Fetch&Increment object, N_b, and lines 3 and 5 of Algorithm 6 ensure that for every i, $0 \leq i \leq c$, exactly one thread p gets number i when accessing N_b (by executing line 4 of Algorithm 6). It follows that the function $TraverseRootSubtree$ is invoked (line 6, Algorithm 6) exactly once for each of the root children of the index tree (i.e., p is the only thread that traverses the subtree numbered i of the index tree).

Let t be the point in time when the last search worker meets the barrier at line 8 of Algorithm 6.

Lemma 3 *Consider any i, $0 \leq i < N_q$. The following hold at t: (1) queue[i] is a heap (i.e., it implements a priority queue); (2) every element of $queue[i]$ is a distinct leaf of the index tree; thus, for every j, $0 \leq j < N_q$, $j \neq i$, the set of elements stored in $queue[i]$ and the set of elements stored in $queue[j]$ are disjoint.*

Proof By inspection of the code, it follows that an insertion of an element in $queue[i]$ can be performed only when line 6 of Algorithm 7 is executed, whereas no deletions are performed on $queue[i]$ by t. Concurrent insertions on $queue[i]$ (executed by multiple search workers) are serialized by acquiring and releasing the lock for $queue[i]$ in lines 5 and 7 of Algorithm 7. Thus, line 6 (Algorithm 7), which performs an insertion of a leaf node in $queue[i]$, is executed in mutual exclusion. Specifically, line 6 executes the sequential code for a heap insertion with parameter a tree node that has as its priority the distance calculated in line 1. These imply that $queue[i]$ is a heap, so claim 1 holds.

By Lemma 2, $TraverseRootSubtree$ is invoked exactly once for each subtree of the index tree. $TraverseRootSubtree$ is a recursive algorithm that visits each tree node at most once. In particular, line 6 (of Algorithm 7) is executed at most once for each node. The condition of the *else if* statement of line 4 ensures that line 6 is executed only for leaf nodes, so only leaf nodes are inserted in $queue[i]$. These imply that claim 2 holds.

We say that an instance of $TraverseRootSubtree$ (Algorithm 7) *visits* a leaf node nd if it executes lines 5-9 with $node$ being equal to nd.

Lemma 4 *Let BSF_t be the value of shared variable BSF at time t. For every leaf node, nd, of the index tree that is not stored in the heaps of array queue at t, it holds that the real distance between the query and each of the data series stored in nd is larger than BSF_t.*

Proof By inspection of the code (Algorithms 6 and 7), it is easy to see that the value of BSF does not change from the point that it is first set (on line 3 of Algorithm 5) until t. Therefore, the value of BSF is equal to BSF_t during the execution of every instance of $TraverseRootSubtree$.

Consider any leaf node nd that is not stored in the heaps of the $queue$ array at t. This can happen only if no instance of $TraverseRootSubtree$ visits this node. By Lemma 2, $TraverseRootsubtree$ is invoked exactly once for each subtree of the index tree. By inspection of the code, it follows that nd belongs to the subtree of a node nd' (that might be nd or one of its proper ancestors) on which the condition of the *if* statement of line 2 (Algorithm 7) is evaluated to *false*, so that $TraverseRootSubtree$ is not called recursively on the nodes of nd''s subtree (including nd), and therefore all these nodes are not visited. Lines 1-2 (Algorithm 7) ensure that the distance between the PAA of the query and the iSAX representation of nd' is greater than BSF_t. Thus, Property 1 implies that the real distance between the query and each of the data series stored in nd is larger than BSF_t, as needed.

The following observation is a simple consequence of the fact that the BSF variable is protected by a distinct lock, and that the value of BSF is updated only if the *if* statement of line 12 (Algorithm 8) is evaluated to *true*.

Observation 1 *The sequence of values stored in shared variable BSF is strictly decreasing.*

Theorem 2 *The response of QR is correct.*

Proof Fix any i, $0 \leq i < N_q$ and let $Q = queue[i]$. Deletions from Q may occur only by executing line 3 of Algorithm 8. Concurrent deletions from Q are serialized by acquiring and releasing the lock for Q (lines 2-4, Algorithm 8). Note that this lock is distinct for each queue. This and Lemma 3 imply that Q respects the semantics of a priority queue. Therefore, when a node nd with its $dist$ field being equal to d is deleted from Q, all other nodes of Q have higher values than d in their $dist$ field.

Let t_f be the first point in time at which $Q.finished$ is set to *true*. Then, lines 8 and 17 imply that the distance between the PAA of the query QR and the iSAX summary of the last node nd deleted from Q is greater than or equal to BSF. Property 2 then implies that for every leaf node nd contained in the queue at t_f, the minimum real distance between the query QR and all the data series stored in nd is larger than the value of BSF at t_f (let this be BSF_f). This and Observation 1 imply that none of the data series of nd may result in a real distance to QR smaller than BSF_f (or future values of BSF), and therefore none of them needs to be further examined. Note that as soon as the $finished$ bit of Q changes to *true*, any future update of this field does not change its value (i.e., it simply re-writes *true* to it); this is why writes into this field are not protected by a lock.

Lemma 4 implies that by processing just the leaf nodes in the heaps of the $queue$ array (and not all leaf nodes of the index tree), the correctness of QR's response is not jeopardized. Every such heap is processed by at least one search worker. This is ensured by the fact that a search worker stops processing heaps of the $queue$ array only if it discovers that the $finished$ bits of all of them have been set to *true* (lines 10-16, Algorithm 6).

5 Experimental Evaluation

We use synthetic and real datasets in order to compare the performance of MESSI with that of competitors from the literature and baselines we developed.

We demonstrate that, under the same settings, MESSI is able to construct the index up to 4.2x faster, and answer similarity search queries up to 11.2x faster than the competitors. Overall, MESSI exhibits robust performance across datasets and settings, and enables for the first time the exploration of very large data series collections at interactive speeds, and leads to complex analytics that execute more than 1 order of magnitude faster than before.

5.1 Setup

[Environment] We used a server with two Intel Xeon E5-2650 v4 2.2Ghz CPUs and 256GB RAM; each one of the two CPUs comprises 12 cores/24 hyper-threads. All algorithms were implemented in C, and compiled using GCC v6.2.0 on Ubuntu Linux v16.04.

[Algorithms] We compared MESSI to the following algorithms:

(i) ParIS+ [67], the state-of-the-art modern hardware data series index.

(ii) ParIS+TS, our extension of ParIS+, where we implemented in a parallel fashion the traditional tree-based exact search algorithm [75]. In brief, this algorithm traverses the tree, and concurrently (1) inserts in the priority queue the nodes (inner nodes or leaves) that cannot be pruned based on the lower bound distance, and (2) pops from the queues nodes for which it calculates the real distances to the candidate series [75]. In contrast, MESSI (a) first makes a *complete pass* over the index using lower bound distance computations and then proceeds with the real distance computations; (b) it only considers the *leaves* of the index for insertion in the priority queue(s); and (c) performs a *second* filtering step using the lower bound distances when popping elements from the priority queue (and before computing the real distances). The performance results we present later justify the choices we have made in MESSI, and demonstrate that a straight-forward implementation of tree-based exact search leads to sub-optimal performance.

(iii) UCR Suite-P, our parallel implementation of the state-of-the-art optimized serial scan technique, UCR Suite [72], which implements all the known optimizations for exact data series similarity search. In UCR Suite-P, every thread is assigned a part of the in-memory data series array, and all threads concurrently and independently process their own parts, performing the real distance calculations in SIMD, and only synchronize at the end to produce the final result. (We do not consider the non-parallel UCR Suite version in our experiments, since it is almost 300x slower.)

In all cases, the algorithms operated exclusively in main memory (the datasets were already loaded in memory, as well). The code for all algorithms used in this paper is available online [5].

[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-200GB (with a default size of 100GB). For the synthetic datasets, we used 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) [18, 75, 81, 86, 89]. We used the same process to generate 100 query series.

For our first real dataset, *Seismic*, we used the IRIS Seismic Data Access repository [1] to gather 100M series representing seismic waves from various locations, for a total size of 100GB. The second real dataset, *SALD*, includes neuroscience MRI data series [4], 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 (chosen using our synthetic series generator).

We repeated all experiments 10 times and we report the average values. We omit the error bars, since all runs gave results that were very similar (less than 3% difference). The 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.

5.2 Parameter Tuning Evaluation

In all our experiments, we use 24 index workers and 48 search workers. We have chosen the chunk size to be 20MB (corresponding to 20K series of length 256 points). Each part of any iSAX buffer, initially holds a small constant number of data series, but its size changes dynamically depending on how many data series it needs to store. The capacity of each leaf of the index tree is 2000 data series (2MB). For query answering, MESSI-mq utilizes 24 priority queues (whereas MESSI-sq utilizes just one priority queue). In either case, each priority queue is implemented using an array whose size changes dynamically based on how many elements must be stored in it. Below we present the experiments that justify the choices for these parameters.

Figure 7 illustrates the time it takes MESSI to build the tree index for different chunk sizes on a random dataset of 100GB. The required time to build the index decreases when the chunk size is small and does not have any big influence in performance after the value of 1K (data series). Chunk sizes smaller than 1K result in high contention when accessing the fetch&increment object used to assign chunks to index workers. In our experiments, we have chosen a size of 20K, as this gives slightly better performance.

Figures 8 and 10 show the impact of varying the index tree leaf size on the time cost of index creation and query answering, respectively. As we see in Figure 8, the larger the leaf size is, the faster index creation becomes. However, once the leaf size becomes 5K or more, this time improvement is insignificant. On the other hand, Figure 10 shows that the query answering time takes its minimum value when the leaf size is set to 2K (data series). So, we have chosen this value for our experiments.

Figure 10 indicates that the influence of varying the leaf size is significant for query answering. Note that when the leaf size is small, there are more leaf nodes in the index tree and therefore, it is highly probable that more nodes will be inserted in the queues, and vice versa. As the leaf size increases, the number of real distance calculations performed to process each one of the leaves in the queue is larger. This causes load imbalance among the different search workers that process the priority queues. For these reasons, we see that at the beginning the time goes down as the leaf size increases, it reaches its minimum value for leaf size 2K series, and then it goes up again as the leaf size further increases.

Figure 9 shows the influence of the initial iSAX buffer size during index creation. This initialization cost is not negligible given that we allocate 2^w iSAX buffers, each consisting of 24 parts (recall that 24 is the number of index workers in the system). As expected, smaller initial sizes for the buffers result in better performance. We have chosen the initial size of each part of the iSAX buffers to be a small constant number of data series. (We also considered a design that collects statistics and allocates the iSAX buffers right from the beginning, but was slower.)

We finally justify the choice of using more than one priority queues for query answering. As Figure 13 shows, MESSI-mq and MESSI-sq have similar performance when the number of threads is smaller than 24. However, as we go

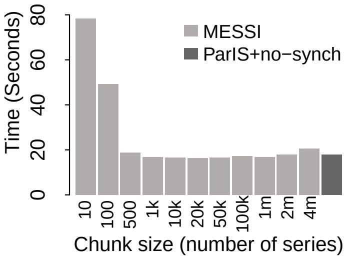


Fig. 7 Index creation, vs. chunk size

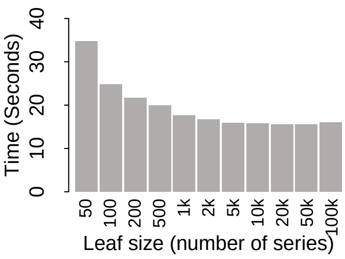


Fig. 8 Index creation, vs. leaf size

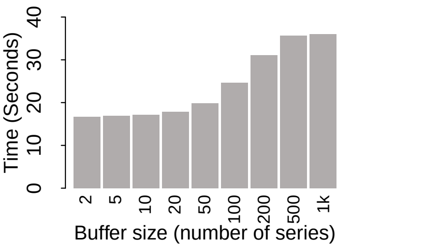


Fig. 9 Index creation, vs. initial iSAX buffer size

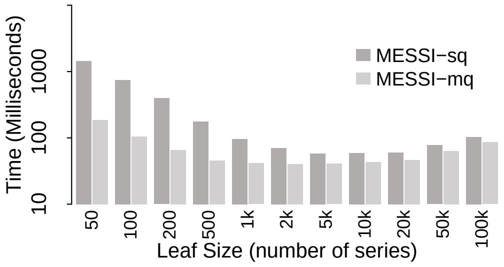


Fig. 10 Query answering, vs. leaf size

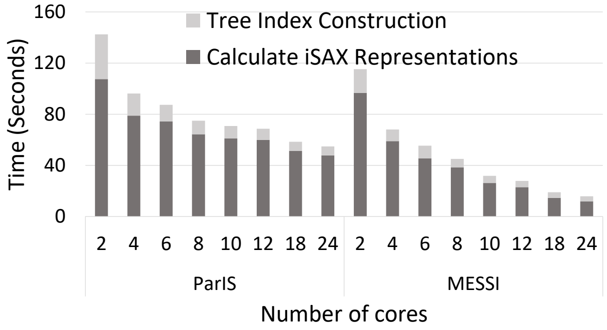


Fig. 11 Index creation, varying number of cores

from 24 to 48 cores, the synchronization cost for accessing the single priority queue in MESSI-sq has negative impact in performance. Figure 16 presents the breakdown of the query answering time for these two algorithms. The figure shows that in MESSI-mq, the time needed to insert and remove nodes from the list is significantly reduced. As expected, the time needed for the real distance calculations and for the tree traversal are about the same in both algorithms. This has the effect that the time needed for the distance calculations becomes the dominant factor. The figure also illustrates the percentage of time that goes on each of these tasks.

Finally, Figure 15 shows the impact of the number of priority queues on query answering performance. As the number of priority queues increases, the time goes down, and is minimized for 24 queues. So, we have chosen this value for our experiments. We also note that each one of these queues handles almost the same number of elements. Our experiments showed that the standard deviation of the number of elements in the queues was always less than 0.8% of the mean number of elements over all the queues used.

5.3 Comparison to Competitors

[Index Creation] Figure 11 compares the index creation time of MESSI with that of ParIS+ as the number of cores increases for a dataset of 100GB. The time MESSI needs for index creation is significantly smaller than that of ParIS+. Specifically, MESSI is 3.5x faster than ParIS+. The main reasons for this are on the one hand that MESSI exhibits lower contention cost when accessing the iSAX buffers in comparison to the corresponding cost paid by ParIS+ to fill in the Receiving Buffers, and on the other hand, that MESSI achieves better load balancing when performing the computation of the iSAX summaries from the raw data series. Note that due to synchronization cost, the performance improvement that both algorithms exhibit decreases as the number of cores increases; this trend is more prominent in ParIS+, while MESSI manages to exploit to a larger degree the available hardware.

In Figure 12, we depict the index creation time as the dataset size grows from 50GB to 200GB. We observe that MESSI performs up to 4.2x faster than ParIS+ (for the 200GB dataset), with the improvement becoming larger with the dataset size.

[Query Answering] Figure 13 compares the performance of the MESSI query answering algorithm to its competitors, as the number of cores increases, for a random dataset of 100GB (y-axis in log scale). The results show that both MESSI-sq and MESSI-mq perform much better than all the other algorithms. Note that the performance of MESSI-mq is better than that of MESSI-sq, so when we mention MESSI in our comparison below we refer to MESSI-mq. MESSI is 55x faster than UCR Suite-P and 6.35x faster than ParIS+ when we use 48 threads (with hyperthreading). In contrast to ParIS+, MESSI applies pruning when performing the lower bound distance calculations and therefore it executes this phase much faster. Moreover, the use of the priority queues result in even higher pruning power. As a side effect, MESSI also performs less real distance calculations than ParIS+.

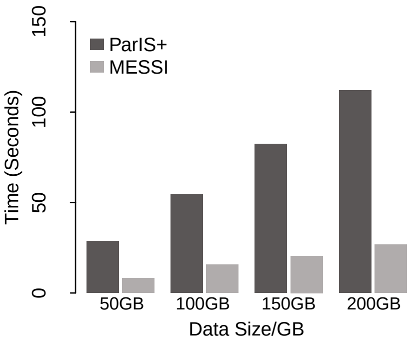


Fig. 12 Index creation, vs. data size

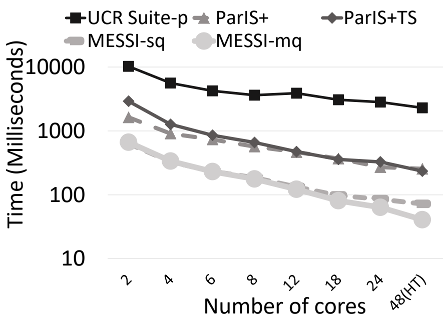


Fig. 13 Query answering, vs. number of cores

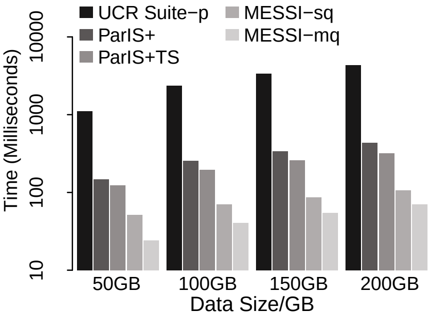


Fig. 14 Query answering, vs. data size

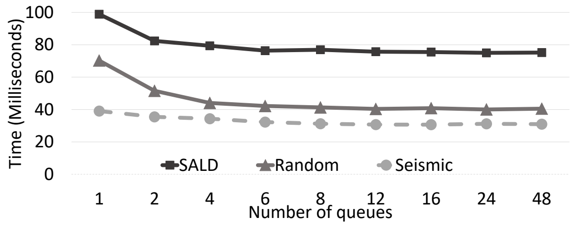


Fig. 15 Query answering, vs. number of queues

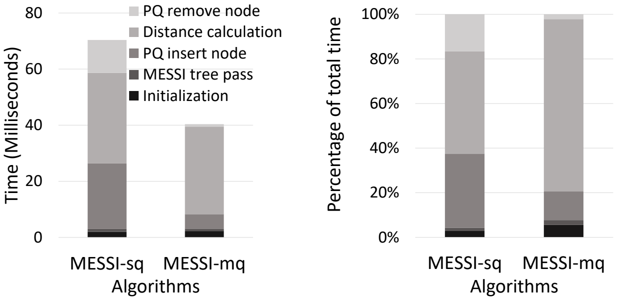


Fig. 16 Query answering with different queue type

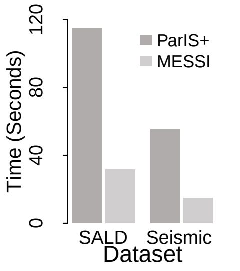


Fig. 17 Index creation for real datasets

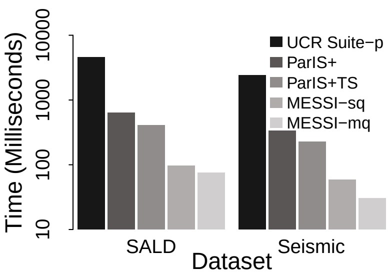
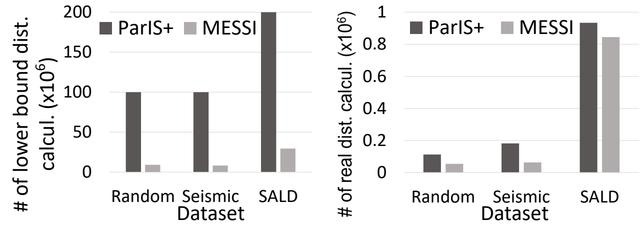


Fig. 18 Query answering for real datasets

Note that UCR Suite-P does not perform any pruning, thus resulting in a much lower performance than the other algorithms.

Figure 14 shows that this superior performance of MESSI is observed across various data set sizes: MESSI is up to 61x faster than UCR Suite-p (for 200GB), up to 6.35x faster than ParIS+ (for 100GB), and up to 7.4x faster than ParIS+TS (for 50GB).



(a) Lower bound dist. calculations (b) Real distance calculations

Fig. 19 Number of distance calculations

In the following experiment, we studied the query answering performance as we varied the length of the data series. We measured the query answering time for data series, whose length ranged from 128 to 2048 points. The five datasets with different data series length that we used in this experiment are random, and in order to factor out the effect of dataset size (following previous work [27, 28]), they are all 100GB in size. This means that the datasets with longer series contain a smaller number of series overall. In all cases, the iSAX summaries were built using 16 segments.

Figure 20 shows that the query answering performance of all algorithms increases with the length of the series. This is to be expected, since the total number of series, and therefore distances that should be computed, is decreasing. We observe that as we increase the series length, the lower bounds become looser. Consequently, the proportion of lower bound and real distance calculations increase, as reported in Figures 21 and 22, respectively. These results also show that ParIS+ spends time to perform lower bound calculations for all series in the dataset in order to save on the real distance calculations, while ParIS+TS ends up performing a large number of both lower bound and real distance calculations. On the other hand, the query answering strategy of MESSI proves very efficient in terms pruning, leading to a small number of lower bound and real distance calculations. We also observe that when compared to ParIS+TS, MESSI is much more efficient in handling the priority queues. Figure 23 shows the time spent on priority queue insertion and deletion operations for the two algorithms. The time to han-

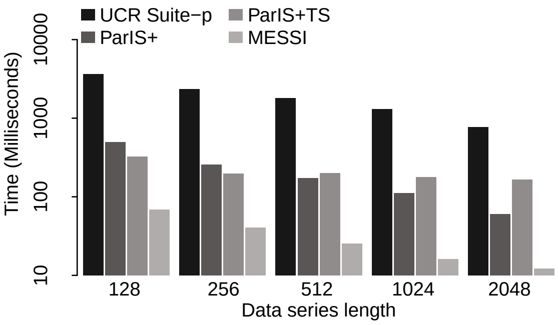


Fig. 20 Query answering, vs. data series length.

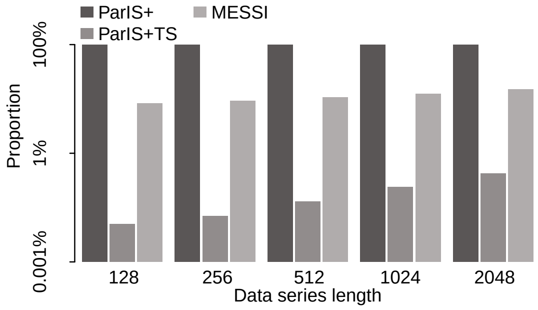


Fig. 21 Proportion of lower bound distance calculations, vs. data series length.

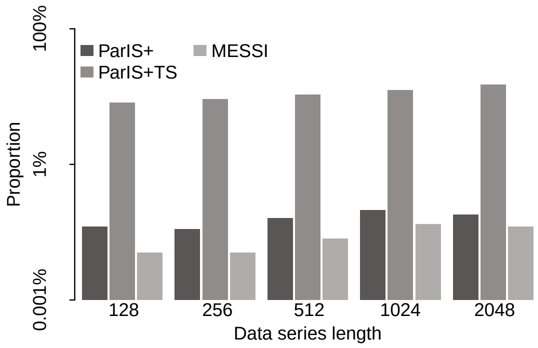


Fig. 22 Proportion of real distance calculations, vs. data series length.

dling queue become stable for both 2 algorithms. The slow performance of ParIS+TS is due to the fact that it inserts in the queue not only the leaf nodes (like MESSI), but also the inner nodes. All the above performance characteristics make MESSI the overall winner in terms of query answering time, across all data series lengths we tried (Figure 20).

[Performance Benefit Breakdown] We now evaluate each of the design choices of MESSI in isolation. This evaluation serves as a methodological analysis that can help understand the benefit of individual design decisions, and if/how they apply to other indices.

Note that some of our design decisions stem from the fact that in our index the root node has a large number of children. Thus, the same design ideas are applicable to the

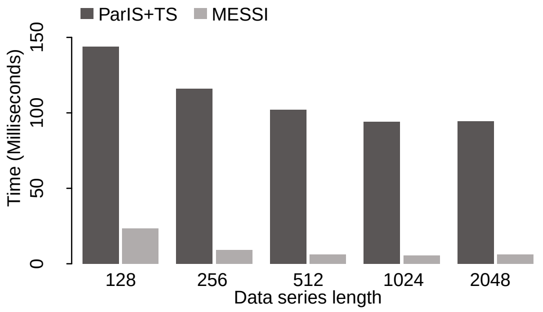


Fig. 23 Time spent in priority queue insertions and deletions, vs. data series length.

iSAX family of indices (e.g., iSAX2+ [18], ADS+ [89], ULISSE [51]). Other indices however [27], use a binary tree (e.g., DSTree), or a tree with a very small fanout (e.g., SFA trie, M-tree), so new design techniques are required for efficient parallelization. However, some of our techniques (e.g., the use of SIMD, priority queues, and some of the data structures designed to reduce the syncrhonization cost) can be applied to all other indices.

We first examine index creation (refer to Figure 24). The main performance benefit comes from removing the synchronization cost of ParIS+ when filling up the receiving buffers. In accessing the buffers, we completely eliminated the synchronization cost by splitting each such buffer into as many chunks as the number of worker threads. Then each thread inserts and removes elements without encountering any contention, leading to a 2.5x speedup (shown as ParIS+no-synch in the graph) when compared to ParIS+. To achieve better load balancing, MESSI splits the array into smaller chunks and uses a Fetch&Add object to assign chunks to threads, resulting to a further performance improvement of 11%.

Then, we examine the query answering performance (refer to Figure 25). The leftmost bar (ParIS+SISD) shows the performance of ParIS+ when SIMD is *not* used. By employing SIMD, ParIS+ becomes 60% faster than ParIS+SISD. We then measure the performance for ParIS+TS, which is about 10% faster than ParIS+. This improvement comes form the fact that using the index tree (instead of the SAX array that ParIS+ uses) to prune the search space and determine the data series for which a real distance calculation must be performed, significantly reduces the number of lower bound distance calculations. ParIS+ calculates lower bound distances for all the data series in the collection, and pruning is performed only when calculating real distances, whereas in ParIS+TS pruning also occurs when calculating lower bounds.

Next, we apply the following technique to reduce the number of real distance calculations we perform: for each node extracted from the priority queue (where ParIS+TS stores the nodes it needs to process), we first calculate the lower bound distance and only if it is smaller than the BSF, the algorithm calculates the real distance. We call this algorithm ParIS+TS-LB, which results in a further performance improvement of 13%. MESSI-sq improves upon ParIS+TS-LB in that it inserts in the priority queue only leaf nodes. This reduces the number of nodes that are inserted in the priority queue and therefore also the size of the queue, as well as the contention incurred when accessing it. Given that in a priority queue, all threads need to synchronize at the root node, this contention is usually rather high. Figure 25 shows that MESSI-sq is 65% faster than ParIS+TS-LB. MESSI-mq further reduces the synchronization cost by maintaining more than one queues and having different threads choose on which queues to work on. This makes MESSI-mq 42% faster than MESSI-sq.

In Table 1, we report a breakdown of the number of operation executions that the different algorithms perform. These numbers help explain the observations (and design choices) mentioned above. Observe that ParIS+TS performs many real distance calculations, because it does not use the second-level filter opportunity offered by the full-length iSAX representation of each data series. ParIS+TS-LB improves on this aspect. When compared to ParIS+, ParIS+TS-LB only performs 9% of the lower bound distance calculations, because it uses the index tree and the priority queue in order to prune. This also means that ParIS+TS-LB performs less than 50% of the real distance calculations. However, ParIS+TS-LB still executes many insert/delete node operations on the priority queue. MESSI sq/hq are much more efficient in handling the priority queue, since it only inserts leaf nodes in the queue.

[Real Datasets] Figures 17 and 18 reaffirm that MESSI exhibits the best performance for both index creation and query answering, even when executing on the real datasets, SALD and Seismic (for a 100GB dataset), for the reasons listed in the previous paragraphs. Regarding index creation, MESSI is 3.6x faster than ParIS+ on SALD and 3.7x faster than ParIS on Seismic, for a 100GB dataset. Moreover, for SALD, MESSI query answering is 60x faster than UCR Suite-P and 8.4x faster than ParIS+, whereas for Seismic, it is 80x faster than UCR Suite-P, and almost 11x faster than ParIS+.

Figures 19(a) and 19(b) illustrate the number of lower bound and real distance calculations, respectively, performed by the different query algorithms on the three datasets. ParIS+ calculates the distance between the iSAX summaries of every single data series and the query series (because, as we discussed in Section 2, it implements the SIMS strategy for query answering). In contrast, MESSI performs pruning even during the lower bound distance calculations, resulting in much less time for executing this computation. Moreover, this results in a significantly reduced number of series whose real distance to the query must be calculated.

The use of the priority queues lead to even less real distance calculations, because they help the BSF to converge faster to its final value. MESSI performs no more than 15% of the lower bound distance calculations performed by ParIS+.

In the next experiment, we report results with query workloads of increasing difficulty (similarly to earlier work [90]). For these workloads, we select series at random from the collection, add to each point Gaussian noise ($\mu = 0$, $\sigma = 0.01$-0.1), and use these as our queries. Finally, we also select series at random and remove them from the collection, and use these as our *Real* workload.

Figure 26 shows that the pruning proportion of all algorithms increases as we increase the level of noise in the query workloads, while *Real* is even more difficult: for the Seismic dataset, we can only prune 40-55% of the real distance calculations. Nevertheless, MESSI achieves in all cases the best pruning, thanks to its use of the priority queue, where the BSF is always updated as early as possible. The query answering time performance for these workloads is depicted in Figure 27. The results show that as the queries get harder, ParIS+ becomes worse that UCR Suite-p. ParIS+ pays the penalty of having to generate and process the candidate list, which grows very large when pruning is small. ParIS+TS has an advantage in this respect, because it only needs to handle the priority queue (of the non-pruned nodes), which is smaller in size, resulting in an overhead that is much less than ParIS+. MESSI is always better than all competitors: it performs 3.5x-100x faster than UCR Suite-p on the Seismic dataset, and 16x-135x faster on SALD. (ParIS+ was much slower, and we terminated its execution after 10K milliseconds per query.)

In Table 2, we report the MESSI index expansion rate (i.e., the index size as a percentage of the original data size) for the synthetic and real datasets in our study. We observe that for our 100GB datasets, the MESSI index occupies ~5GB of space for Synthetic and Seismic, and ~10GB for SALD. Note that the series in SALD have a length of 128 points (compared to the 256 of Synthetic and Seismic); hence, this dataset contains double the number of series than the other two datasets. This means that the index contains double the number of iSAX summaries. Overall, we conclude that the MESSI index expansion rate is small, rendering MESSI a space efficient index.

[MESSI DTW] Figures 28 and 29 compare the performance of MESSI and UCR Suite for the case of DTW distance. Overall, MESSI is up to 2.5 orders of magnitude faster than UCR Suite-p, a parallel version of UCR Suite that uses SIMD and supports DTW. (For comparison, we also report the performance of the single-core implementation of UCR Suite, which is 1-2 orders of magnitude slower than UCR Suite-p.)

The experimental results on a 100GB dataset show that as we increase the warping window size from 1% to 20% of

Table 1 Query answering algorithms comparison: number of times an operation is executed (average over 100 queries).

	ParIS+	**ParIS+TS**	**ParIS+TS-LB**	**MESSI-sq**	**MESSI-mq**
PQ ins. node	n/a	69,117	69,134	14,620	14,611
PQ del. node	n/a	20,051	20,111	11,152	10,747
LBD calcul.	100 M	69,117	9,173,401	9,175,400	9,170,162
RD calcul.	112,321	9,183,312	52,139	54,207	53,919

Table 2 Index expansion rate (index size as a percentage of the original data size).

	Synthetic 100GB 100M series	Seismic 100GB 100M series	SALD 100GB 200M series
index expansion rate	5.7%	5.1%	10.5%

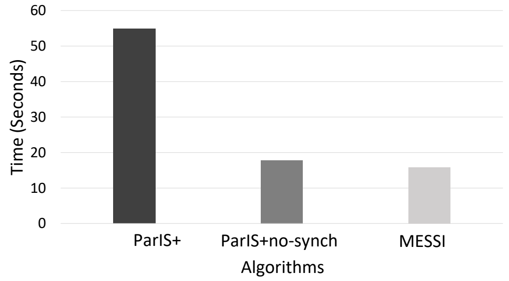


Fig. 24 Index creation time

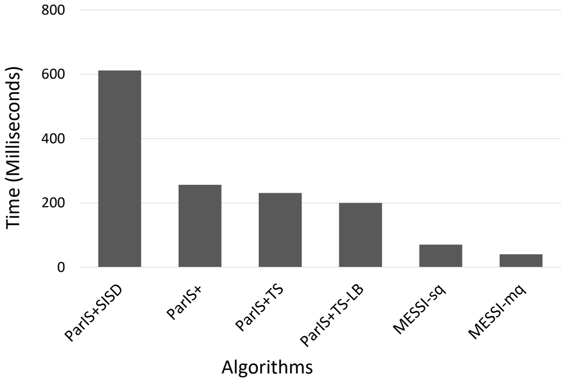


Fig. 25 Query answering time

the data series length, the query answering time of MESSI increases as well: the LB_Keogh envelope of the query becomes wider, and consequently, pruning in the index is smaller (refer to Figure 28). However, MESSI is in all cases at least 9x faster than UCR Suite-p, while for the most common warping window sizes of 5-10% [40], the speedup is between 35-170x. Figure 29 shows query answering time when varying the dataset size (warping window size: 10%). As we increase the size of the data series collection from 50GB to 200GB, MESSI remains 25-35x faster than UCR Suite-p.

Table 3 Update Frequency of the BSF array (Euclidean distance).

	1-NN	**5-NN**	**10-NN**	**50-NN**
number of BSF updates/query	11.9	20.9	45.6	258.1
BSF update time μsec/query	0.5	5.1	19.1	186.5
BSF update time query time %	0.001%	0.01%	0.04%	0.3%

5.4 Complex Analytics Task: Classification

In the following experiment, we tested MESSI 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. This classifier assigns a new object to the majority class of the k NN of that object (a data series, in our case).

The results, depicted in Figure 30, report the performance of MESSI and ParIS+ for different values of k on a 100GB dataset (100M series of size 256 values, generated with our synthetic data generator). The results show that a k-NN Classifier using MESSI can finish a classification task up to 13x faster than when using ParIS+, which can reduce the total processing time for classifying 100K objects from 1 day down to 93min.

We note that the purpose of this experiment was to measure the time performance of executing a k-NN classification task. Even though we did not study a real classification problem, the results are useful in that they report the expected time performance of using MESSI in such a task with a large data series collection.

We evaluated the overhead of executing k-NN queries. MESSI implements k-NN by simply maintaining a sorted array of the best k distances seen so far (i.e., BSF is now an array of k elements). The elements of the array are initialized by performing a single approximate search (like in 1-NN): we choose the k series with the smallest distances to the query and initialize the BSF array with their distances. Whenever a smaller distance than the biggest element of this array is calculated, the array is updated. This process does not result in more operations on the priority queues, or more tree traversals. Table 3 shows that the additional time for executing k-NN instead of 1-NN is negligible (times reported in microseconds).

Finally, we repeated the previous k-NN classification experiment using the DTW distance, which is slower, but can

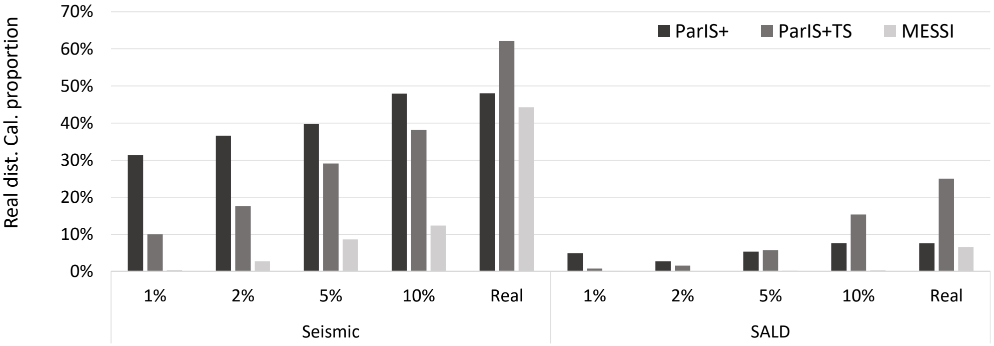


Fig. 26 Number of real distance calculations: real datasets, various query workloads.

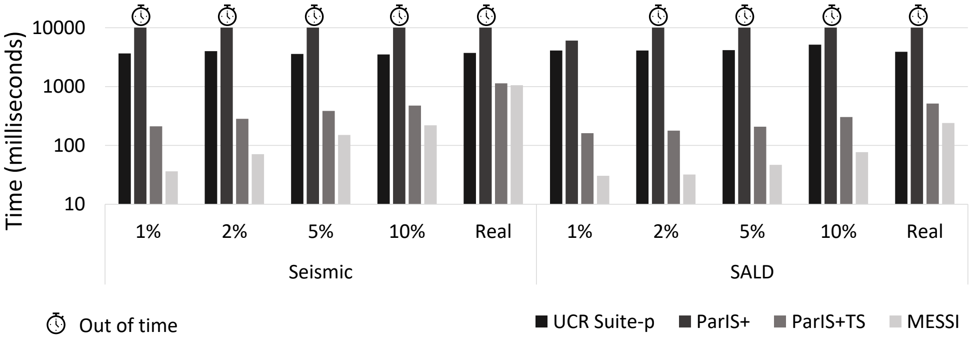


Fig. 27 Query answering time: real datasets, various query workloads.

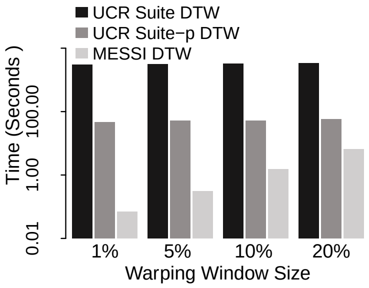


Fig. 28 DTW time (synthetic data, varying warping window size, 100GB dataset).

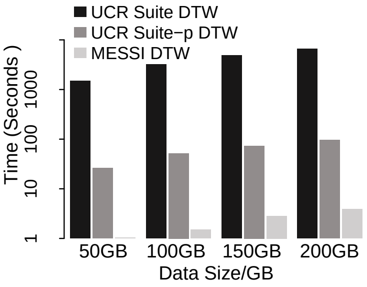


Fig. 29 DTW time (synthetic data, varying data size, 10% warping window).

lead to more accurate classifications [10]. Figures 31 and 32 show the results (the y-axis is expressed in seconds). Observe that when compared to the previous experiment, the execution time (as expected) is now approximately 10x and 30x larger for the 5% and 10% warping window sizes, respectively. Nevertheless, MESSI is up to 9.5x faster than ParIS+ for the 5% warping window size, and up to 4.5x faster for the 10% warping. Therefore, MESSI can considerably reduce the k-NN classification time for large sequence collections. Tables 4 and 5 report the number of BSF updates and the time needed to update the BSF during k-NN similarity search queries using the DTW distance with 5% and 10% warping window sizes, respectively. Similarly to the case of Euclidean distance, we observe that the overhead as the number k of nearest neighbors increases is negligible, even for $k = 50$. Moreover, since the DTW is computationally more expensive than Euclidean, the percentage of the total query answering time dedicated to updating the BSF shrinks significantly (now expressed as *per thousand*).

In Tables 6 and 7, we report the execution time of lower bound and real distance calculations for both the Euclidean and DTW distance measures. The results show that the average time cost per lower-bounding calculation is 6.6x slower for DTW than for Euclidean (Table 6). This is due to the fact that DTW needs to pay the cost of computing the LB_Keogh envelope (for every query). As expected, the time cost difference between Euclidean and DTW is much larger for the real distance calculation, with DTW being 16x slower than Euclidean (Table 7).

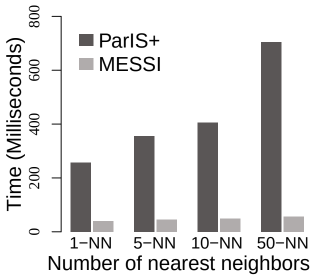


Fig. 30 Time for a k-NN Classifier to classify one object using the Euclidean distance (100GB dataset).

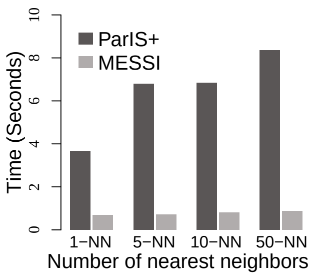


Fig. 31 Time for a k-NN Classifier to classify one object using the DTW distance using a warping window size of 5% of the series length (100GB dataset).

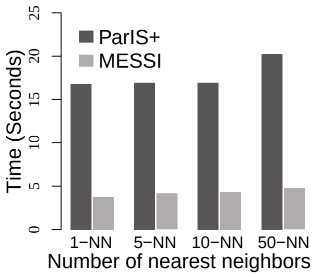


Fig. 32 Time for a k-NN Classifier to classify one object using the DTW distance using a warping window size of 10% of the series length (100GB dataset).

Table 4 Update Frequency of the BSF array (DTW distance, 5% warping).

	1-NN	**5-NN**	**10-NN**	**50-NN**
number of BSF updates/query	22.7	83.9	160.2	672.4
BSF update time μsec/query	4.9	19.9	50.2	473.3
BSF update time query time ‰	0.007‰	0.03‰	0.06‰	0.5‰

Table 5 Update Frequency of the BSF array (DTW distance, 10% warping).

	1-NN	**5-NN**	**10-NN**	**50-NN**
number of BSF updates/query	45.8	124.7	221.4	854.1
BSF update time μsec/query	11.9	31.5	72.5	574.2
BSF update time query time ‰	0.003‰	0.008‰	0.002‰	0.1‰

Table 6 Time cost of lower bound distance calculations.

Distance measure	SISD	SIMD
Euclidean	107.5ns	31.4ns
DTW	122.7ns	30.5ns

Table 7 Time cost of real distance calculations.

Distance measure	Query	Time (milliseconds)
Euclidean	1-NN	40.3
	50-NN	56.3
DTW	1-NN	679.3
	50-NN	879.7

6 Related Work

Various dimensionality reduction techniques exist for data series, which can then be scanned and filtered [38, 49] or indexed and pruned [20–22,42–44,52,61,65,75,76,81,89] during query answering, including deep-learned methods [80]; for a complete discussion of such techniques, we refer the reader to two recent tutorials on the subject [25, 26].

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 in-memory index construction and similarity search for data series. The work closest to ours is Paris/ParIS+ [65, 67], which exploits modern hardware, but was designed for disk-resident datasets (see also Section 2).

FastQuery is an approach used to accelerate search operations in scientific data [23], 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).

The interest in using SIMD instructions for improving the performance of data management solutions is not new [88]. However, it is only more recently that relatively complex algorithms were extended in order to take advantage of this hardware characteristic. Polychroniou et al. [70] introduced design principles for efficient vectorization of in-memory database operators (such as selection scans, hash tables, and partitioning). For data series in particular, previous work has used SIMD for Euclidean distance computations [78]. Following [65], in our work we use SIMD both for the computation of Euclidean distances, as well as for the computation of lower bounds, which involve branching operations.

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 used to increase efficiency [31].

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 [55]. Binna et al. [11], present the Height Optimized Trie (HOT), a general-purpose index structure for main-memory database systems, while Leis et al. [46] describe an in-memory adaptive Radix indexing technique that is designed for modern hardware. Xie et al. [84], study and analyze five recently proposed indices, i.e., FAST, Masstree, BwTree, ART and PSL and identify the effectiveness of common optimization techniques, including hardware dependent features such as SIMD, NUMA and HTM. They argue that there is no single optimization strategy that fits all situations, due to the differences in the dataset and workload characteristics. Moreover, they point out the significant performance gains that the use of modern hardware features bring to in-memory indices.

We note that the indices described above are not suitable for data series (or very high-dimensional data), which is the focus of our work, and which pose very specific data management challenges with their hundreds, or thousands of dimensions (i.e., the length of the sequence). Techniques specifically designed for modern hardware and in-memory operation have also been studied in the context of adaptive indexing [8], and data mining [79].

Piatov et al. propose DeltaTop, a fast time series subsequence matching method [68], where the query sequence is itself part of the dataset (i.e., self-join). Their method uses a prefix-sum Euclidean distance matrix to accelerate subsequence matching, and supports search in multi-variate time series. The authors provide a parallel (multi-core) implementation of their method. Compared to our work, we observe that they solve the problem of (self-join) subsequence similarity matching, only for non Z-normalized sequences using the Euclidean distance. In contrast, we solve the whole-matching problem [27], supporting non Z-normalized and Z-normalized sequences, using both the Euclidean and the DTW distances. Even when the two approaches are compared in the specific setting of subsequence similarity search[9] on non Z-normalized data using the Euclidean distance, we observed that DeltaTop's high index creation time and memory cost did not allow it to scale to sequence collections with more than 100K points (i.e., considerably smaller than the datasets we used for evaluating MESSI).

Finally, KV-Match [82] and its improvement, L-Match [30], are index structures that can support similarity search in a distributed setting. Nevertheless, we note that they were developed for subsequence matching on disk-resident data, while the focus of MESSI is on whole-matching for in-memory data.

7 Conclusions

Data series are a very common data type, with increasingly larger collections being generated by applications in many and diverse domains. In many exploration and analysis pipelines, similarity search is a key operation, which is nevertheless challenging to efficiently support over large data series collections.

In this work, we proposed MESSI, a data series index designed for in-memory operation by exploiting the parallelism opportunities of modern hardware. MESSI is up to 4x faster in index construction and up to 11x faster in query answering than the state-of-the-art solution, and is the first technique to answer answering exact similarity search queries on 100GB datasets in $\sim$50msec. This level of performance enables for the first time interactive analytics on very large data series collections.

Finally, we note that the ideas presented in this work are applicable to other indices that have a root node with a large fanout degree. This is true for other iSAX-based indices. For example, we could parallelize in a way similar to MESSI the ULISSE index [53], which supports queries of variable length, as well as the DPiSAX index [85], which is a distributed index operating on top of Spark (but currently not supporting parallel execution within each node of the Spark cluster). It is an interesting open problem to study whether there exist efficient parallelization techniques for indexing schemes whose tree index does not satisfy this large fanout property that would result in better perfomance than MESSI.

[Acknowledgements] Work supported by Investir l'Avenir, Univ. of Paris IDEX Emergence en Recherche ANR-18-IDEX-000, CSC, FMJH PGMO, EDF, Thales, HIPEAC 4. Partly performed when P. Fatourou visited LIPADE and B. Peng visited CARV, FORTH ICS.

References

1. Incorporated Research Institutions for Seismology – Seismic Data Access. http://ds.iris.edu/data/access/ (2016)
2. Adhd-200. `http://fcon_1000.projects.nitrc.org/indi/adhd200/` (2017)
3. Sloan digital sky survey. `https://www.sdss3.org/dr10/data_access/volume.php` (2017)
4. Southwest university adult lifespan dataset (sald). `http://fcon_1000.projects.nitrc.org/indi/retro/sald.html` (2018)
5. http://helios.mi.parisdescartes.fr/ themisp/messi/ (2020)
6. Agrawal, R., Faloutsos, C., Swami, A.N.: Efficient similarity search in sequence databases. In: FODO (1993)
7. Ailamaki, A.: Databases and hardware: The beginning and sequel of a beautiful friendship. VLDB (2015)
8. Alvarez, V., Schuhknecht, F.M., Dittrich, J., Richter, S.: Main memory adaptive indexing for multi-core systems. In: DaMoN (2014)
9. Bagnall, A.J., Cole, R.L., Palpanas, T., Zoumpatianos, K.: Data series management (dagstuhl seminar 19282). Dagstuhl Reports (9(7), 2019)
10. Bagnall, A.J., Lines, J., Bostrom, A., Large, J., Keogh, E.J.: The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. Data Min. Knowl. Discov. **31**(3), 606–660 (2017). DOI 10.1007/s10618-016-0483-9. URL `https://doi.org/10.1007/s10618-016-0483-9`
11. Binna, R., Zangerle, E., Pichl, M., Specht, G., Leis, V.: Hot: A height optimized trie index for main-memory database systems. In: SIGMOD. ACM (2018)

[9] MESSI can be adapted to support subsequence matching as follows: given a long series (in which we need to identify the most similar subsequence to the query), we extract subsequences from the long series by sliding a window window (of the same length as the query) over the entire length of the series, and then index all these subsequences.

12. Blanas, S.: Query processing for datacenter-scale computers. In: CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings (2017)
13. Boniol, P., Linardi, M., Roncallo, F., Palpanas, T.: Automated Anomaly Detection in Large Sequences. In: ICDE (2020)
14. Boniol, P., Linardi, M., Roncallo, F., Palpanas, T., Meftah, M., Remy, E.: Unsupervised and Scalable Subsequence Anomaly Detectionin Large Data Series. VLDBJ (2021)
15. Boniol, P., Palpanas, T.: Series2Graph: Graph-based Subsequence Anomaly Detection for Time Series. PVLDB (2020)
16. Boniol, P., Paparrizos, J., Palpanas, T., Franklin, M.J.: SAND in Action: Subsequence Anomaly Detection for Streams. PVLDB (2021)
17. Boniol, P., Paparrizos, J., Palpanas, T., Franklin, M.J.: SAND: Streaming Subsequence Anomaly Detection. PVLDB (2021)
18. Camerra, A., Shieh, J., Palpanas, T., Rakthanmanon, T., Keogh, E.: Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+. KAIS **39**(1) (2014)
19. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. CSUR (2009)
20. Chatzigeorgakidis, G., Skoutas, D., Patroumpas, K., Palpanas, T., Athanasiou, S., Skiadopoulos, S.: Local pair and bundle discovery over co-evolving time series. In: Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD (2019)
21. Chatzigeorgakidis, G., Skoutas, D., Patroumpas, K., Palpanas, T., Athanasiou, S., Skiadopoulos, S.: Local similarity search on geolocated time series using hybrid indexing. In: Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL (2019)
22. Chatzigeorgakidis, G., Skoutas, D., Patroumpas, K., Palpanas, T., Athanasiou, S., Skiadopoulos, S.: Twin subsequence search in time series. In: Proceedings of the 24th International Conference on Extending Database Technology, EDBT (2021)
23. Chou, J., Wu, K., et al.: Fastquery: A parallel indexing system for scientific data. In: CLUSTER, pp. 455–464. IEEE (2011)
24. Coorporation, I.: Intel 64 and ia-32 architectures optimization reference manual (2016)
25. Echihabi, K., Zoumpatianos, K., Palpanas, T.: Big sequence management: on scalability. In: Proceedings of the IEEE International Conference on Big Data, IEEE BigData (2020)
26. Echihabi, K., Zoumpatianos, K., Palpanas, T.: Big sequence management: Scaling up and out. In: Proceedings of the 24th International Conference on Extending Database Technology, EDBT (2021)
27. Echihabi, K., Zoumpatianos, K., Palpanas, T., Benbrahim, H.: The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. PVLDB (2018)
28. Echihabi, K., Zoumpatianos, K., Palpanas, T., Benbrahim, H.: Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. PVLDB (2019)
29. Fekete, J.D., Primet, R.: Progressive analytics: A computation paradigm for exploratory data analysis. CoRR (2016)
30. Feng, K., Wang, P., Wu, J., Wang, W.: L-match: A lightweight and effective subsequence matching approach. IEEE Access **8**, 71572–71583 (2020)
31. Gepner, P., Kowalik, M.F.: Multi-core processors: New way to achieve high system performance. In: PAR ELEC (2006)
32. Gogolou, A., Tsandilas, T., Echihabi, K., Bezerianos, A., Palpanas, T.: Data series progressive similarity search with probabilistic quality guarantees. In: D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, H.Q. Ngo (eds.) Proceedings of the 2020 International Conference on Management of Data, SIGMOD (2020)
33. Gogolou, A., Tsandilas, T., Palpanas, T., Bezerianos, A.: Progressive similarity search on time series data. In: EDBT (2019)
34. Gowanlock, M.G., Casanova, H.: Distance threshold similarity searches: Efficient trajectory indexing on the GPU. IEEE Trans. Parallel Distrib. Syst. **27**(9) (2016)
35. Grabocka, J., Schilling, N., Schmidt-Thieme, L.: Latent time-series motifs. TKDD **11**(1), 6:1–6:20 (2016)
36. Guillaume, A.: Head of Operational Intelligence Department Airbus. Personal communication. (2017)
37. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann Publishers Inc. (2012)
38. Kashyap, S., Karras, P.: Scalable knn search on vertically stored time series. In: SIGKDD, pp. 1334–1342 (2011)
39. Keogh, E., Chakrabarti, K., Pazzani, M., Mehrotra, S.: Dimensionality reduction for fast similarity search in large time series databases. KAIS (2001)
40. Keogh, E., Ratanamahatana, C.A.: Exact indexing of dynamic time warping. Knowledge and information systems (2005)
41. Keogh, E.J., Pazzani, M.J.: An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In: KDD (1998)
42. Kondylakis, H., Dayan, N., Zoumpatianos, K., Palpanas, T.: Coconut: A scalable bottom-up approach for building data series indexes. PVLDB (2018)
43. Kondylakis, H., Dayan, N., Zoumpatianos, K., Palpanas, T.: Coconut palm: Static and streaming data series exploration now in your palm. In: SIGMOD (2019)
44. Kondylakis, H., Dayan, N., Zoumpatianos, K., Palpanas, T.: Coconut: sortable summarizations for scalable indexes over static and streaming data series. VLDBJ **28**(6) (2019)
45. Laviron, P., Dai, X., Huquet, B., Palpanas, T.: Electricity demand activation extraction: From known to uknown signatures, using similarity search. In: Proceedings of the ACM International Conference on Future Energy Systems, e-Energy (2021)
46. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: ICDE (2013)
47. Lemire, D.: Faster retrieval with a two-pass dynamic-time-warping lower bound. Pattern Recognit. **42**(9), 2169–2180 (2009)
48. Levchenko, O., Kolev, B., Yagoubi, D.E., Akbarinia, R., Masseglia, F., Palpanas, T., Shasha, D.E., Valduriez, P.: Bestneighbor: efficient evaluation of knn queries on large time series databases. Knowl. Inf. Syst. **63**(2), 349–378 (2021)
49. Li, C., Yu, P.S., Castelli, V.: Hierarchyscan: A hierarchical similarity search algorithm for databases of long sequences. In: ICDE (1996)
50. Liao, T.W.: Clustering of time series data - a survey. Pattern Recognition **38**(11), 1857–1874 (2005)
51. Linardi, M., Palpanas, T.: ULISSE: ULtra compact Index for Variable-Length Similarity SEarch in Data Series. In: ICDE (2018)
52. Linardi, M., Palpanas, T.: Scalable, variable-length similarity search in data series: The ulisse approach. PVLDB (2019)
53. Linardi, M., Palpanas, T.: Scalable data series subsequence matching with ULISSE. VLDB J. **29**(6), 1449–1474 (2020)
54. Linardi, M., Zhu, Y., Palpanas, T., Keogh, E.J.: Matrix Profile Goes MAD: Variable-Length Motif And Discord Discovery in Data Series. In: DAMI (2020)
55. Lomet, D.B., Nawab, F.: High performance temporal indexing on modern hardware. In: ICDE (2015)
56. Lomont, C.: Introduction to intel advanced vector extensions. Intel White Paper (2011)
57. Mueen, A., Keogh, E.J., Zhu, Q., Cash, S., Westover, M.B., Shamlo, N.B.: A disk-aware algorithm for time series motif discovery. DAMI (2011)
58. Mueen, A., Nath, S., Liu, J.: Fast approximate correlation for massive time-series data. In: SIGMOD (2010)
59. Palpanas, T.: Data series management: The road to big sequence analytics. SIGMOD Record (2015)

60. Palpanas, T.: The parallel and distributed future of data series mining. In: HPCS (2017)
61. Palpanas, T.: Evolution of a Data Series Index. CCIS **1197** (2020)
62. Palpanas, T., Beckmann, V.: Report on the first and second interdisciplinary time series analysis workshop (ITISA). SIGREC (48(3), 2019)
63. Pelkonen, T., Franklin, S., Cavallaro, P., Huang, Q., Meza, J., Teller, J., Veeraraghavan, K.: Gorilla: A fast, scalable, in-memory time series database. VLDB (2015)
64. Peng, B., Fatourou, P., Palpanas, T.: SING: Sequence Indexing Using GPUs. In: ICDE (2021)
65. Peng, B., Palpanas, T., Fatourou, P.: Paris: The next destination for fast data series indexing and query answering. IEEE BigData (2018)
66. Peng, B., Palpanas, T., Fatourou, P.: Messi: In-memory data series indexing. In: ICDE (2020)
67. Peng, B., Palpanas, T., Fatourou, P.: Paris+: Data series indexing on multi-core architectures. TKDE (2020)
68. Piatov, D., Helmer, S., Dignös, A., Gamper, J.: Interactive and space-efficient multi-dimensional time series subsequence matching. Information Systems **82**, 121–135 (2019)
69. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, pp. 1493–1508 (2015)
70. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking simd vectorization for in-memory databases. In: SIGMOD. ACM (2015)
71. Polychroniou, O., Ross, K.A.: Vectorized bloom filters for advanced SIMD processors. In: Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014, pp. 6:1–6:6 (2014)
72. Rakthanmanon, T., Campana, B.J.L., Mueen, A., Batista, G.E.A.P.A., Westover, M.B., Zhu, Q., Zakaria, J., Keogh, E.J.: Searching and mining trillions of time series subsequences under dynamic time warping. In: SIGKDD (2012)
73. Rakthanmanon, T., Keogh, E.J., Lonardi, S., Evans, S.: Time series epenthesis: Clustering time series streams requires ignoring some data. In: ICDM, pp. 547–556 (2011)
74. Rodrigues, P.P., Gama, J., Pedroso, J.: Hierarchical clustering of time-series data streams. TKDE (2008)
75. Shieh, J., Keogh, E.: i sax: indexing and mining terabyte sized time series. In: SIGKDD (2008)
76. Shieh, J., Keogh, E.: iSAX: disk-aware mining and indexing of massive time series datasets. DMKD (1) (2009)
77. Tan, C.W., Webb, G.I., Petitjean, F.: Indexing and classifying gigabytes of time series under time warping. In: ICDM (2017)
78. Tang, B., Yiu, M.L., Li, Y., et al.: Exploit every cycle: Vectorized time series algorithms on modern commodity cpus. In: IMDM (2016)
79. Tatikonda, S., Parthasarathy, S.: An adaptive memory conscious approach for mining frequent trees: implications for multi-core architectures. In: SIGPLAN. ACM (2008)
80. Wang, Q., Palpanas, T.: Deep Learning Embeddings for Data Series Similarity Search. In: SIGKDD (2021)
81. Wang, Y., Wang, P., Pei, J., Wang, W., Huang, S.: A data-adaptive and dynamic segmentation index for whole matching on time series. VLDB (2013)
82. Wu, J., Wang, P., Pan, N., Wang, C., Wang, W., Wang, J.: Kv-match: A subsequence matching approach supporting normalization and time warping. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 866–877. IEEE (2019)
83. Xiao, L., Zheng, Y., Tang, W., Yao, G., Ruan, L.: Parallelizing dynamic time warping algorithm using prefix computations on gpu. In: (HPCC_EUC). IEEE (2013)
84. Xie, Z., Cai, Q., Chen, G., Mao, R., Zhang, M.: A comprehensive performance evaluation of modern in-memory indices. In: ICDE (2018)
85. Yagoubi, D.E., Akbarinia, R., Masseglia, F., Palpanas, T.: Massively distributed time series indexing and querying. IEEE Trans. Knowl. Data Eng. **32**(1), 108–120 (2020)
86. Yi, B.K., Faloutsos, C.: Fast time sequence indexing for arbitrary lp norms. In: VLDB. Citeseer (2000)
87. Zeuch, S., Freytag, J., Huber, F.: Adapting tree structures for processing with SIMD instructions. In: EDBT (2014)
88. Zhou, J., Ross, K.A.: Implementing database operations using simd instructions. In: SIGMOD (2002)
89. Zoumpatianos, K., Idreos, S., Palpanas, T.: Ads: the adaptive data series index. VLDB J. (2016)
90. Zoumpatianos, K., Lou, Y., Ileana, I., Palpanas, T., Gehrke, J.: Generating data series query workloads. VLDB J. **27**(6) (2018)
91. Zoumpatianos, K., Palpanas, T.: Data series management: Fulfilling the need for big sequence analytics. In: ICDE (2018)