

# ParIS+: Data Series Indexing on Multi-Core Architectures

Botao Peng, Panagiota Fatourou, Themis Palpanas

**Abstract**—Data series similarity search is a core operation for several data series analysis applications across many different domains. Nevertheless, even state-of-the-art techniques cannot provide the time performance required for large data series collections. We propose ParIS and ParIS+, the *first* disk-based data series indices carefully designed to inherently take advantage of multi-core architectures, in order to accelerate similarity search processing times. Our experiments demonstrate that ParIS+ completely removes the CPU latency during index construction for disk-resident data, and for exact query answering is up to 1 order of magnitude faster than the current state of the art index scan method, and up to 3 orders of magnitude faster than the optimized serial scan method. ParIS+ (which is an evolution of the ADS+ index) owes its efficiency to the effective use of multi-core and multi-socket architectures, in order to distribute and execute in parallel both index construction and query answering, and to the exploitation of the Single Instruction Multiple Data (SIMD) capabilities of modern CPUs, in order to further parallelize the execution of instructions inside each core.

**Index Terms**—data series, time series, indexing, similarity search, query answering, multi-core architectures, parallelization



## 1 INTRODUCTION

**[Motivation]** An increasing number of applications across many diverse domains continuously produce very large amounts of data series<sup>1</sup> (such as in finance, environmental sciences, astrophysics, neuroscience, engineering, multimedia, etc. [1], [2], [3], [4]), which makes them one of the most common types of data. When these sequence collections are generated (often times composed of a large number of short series [3]), users may need to query and analyze them as soon as they become available. This process is heavily dependent on data series similarity search (which apart from being a useful query in itself, also lies at the core of several machine learning methods, such as, clustering, classification, motif and outlier detection, etc.) [5], [6], [7], [8]. The brute-force approach for evaluating similarity search queries is by performing a sequential pass over the complete dataset. However, as data series collections grow larger, scanning the complete dataset becomes a performance bottleneck, taking hours or more to complete [9]. This is especially problematic in exploratory search scenarios, where every next query depends on the results of previous queries. Consequently, we have witnessed an increased interest in developing indexing techniques and algorithms for similarity search [10], [11], [12], [13], [9], [14], [15], [16], [17], [18], [19], [6], [20], [21].

**[Scalability problem]** Nevertheless, the continued increase in the rate and volume of data series production with collections that grow to several terabytes [1] renders existing data series indexing technologies inadequate. For example, the current state-of-the-art index, ADS+ [9], [22], requires more than 4min to answer any

single exact query on a moderately sized 250GB sequence collection. Moreover, index construction time also becomes a significant bottleneck in the analysis process [9], especially in cases where new data arrive frequently and need to be indexed [3]. Thus, traditional solutions and systems are inefficient at, or incapable of managing and processing the voluminous sequence collections that already exist in several domains. Finally, we note that, given the evolution of CPU performance, where the processor clock speed is not increasing due to the power wall constraint, efforts for algorithmic speedups now exploit the parallelism opportunities offered by modern hardware [23], [24], [25], [26], [27].

**[Parallel Indexing]** In this work, we propose the Parallel Index for Sequences (ParIS), the first data series index that inherently takes advantage of modern hardware parallelization, and incorporate the state-of-the-art techniques in sequence indexing, in order to accelerate processing times. ParIS, which is a disk-based index based on the principles of ADS+, takes advantage of multi-core and multi-socket architectures, in order to distribute and execute in parallel the computations needed for both index construction and query answering. Moreover, ParIS uses the Single Instruction Multiple Data (SIMD) CPU instructions, in order to further parallelize the execution of individual instructions inside each core. Overall, ParIS achieves very good overlap of the CPU computation with the required disk I/O. To completely remove the CPU cost during index creation, we present ParIS+, an alternative of ParIS that results in index creation that is purely I/O bounded. ParIS+ is 2.6x faster than the current state-of-the-art approach [9]. ParIS and ParIS+ employ the same algorithmic techniques for query answering. The experiments also demonstrate their effectiveness in exact query answering: they are up to 1 order of magnitude faster than the state-of-the-art index scan method [9], and up to 3 orders of magnitude faster than the state-of-the-art optimized serial scan [11]. We also note that ParIS and ParIS+ have the potential to deliver more benefit as we move to faster storage media.

In developing ParIS+ (and ParIS), we made careful design choices in the coordination of the compute and I/O tasks, and consequently, developed new algorithms for the construction of

- Botao Peng and Themis Palpanas are affiliated with LIPADE, Université de Paris. E-mail: botao.peng@parisdescartes.fr, themis@mi.parisdescartes.fr
- Panagiota Fatourou is affiliated with ICS, FORTH, and with the Dept. of Computer Science, University of Crete. E-mail: faturu@csd.uoc.gr

Manuscript received date; revised date.

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

the index and for answering similarity search queries on this index.

We note that even though scaling out to multiple machines is also a valid research direction [25], [14], [19], in this work, we focus on addressing the problem in the context of a single machine, so as to maximize the benefit we can get out of the hardware. Our results can be combined with a scale-out solution. Examining other hardware solutions, like GPUs and FPGAs are also very promising directions, but out of the scope of this work.

**[Contributions]** Our contributions<sup>2</sup> are summarized as follows:

- We propose ParIS, the first data series index designed for multi-core architectures. We describe parallel algorithms for index creation and *exact* query answering, which employ parallelism in reading the data from disk and processing them in the CPU. Moreover, we propose ParIS+, a ParIS alternative that completely masks out the CPU cost when creating the index. ParIS+ results in improved performance during index creation in systems that support a reasonable level of parallelism (more than four cores).
- In order to further speedup query answering, we exploit SIMD for complex vectorial computations: we develop novel vectorized implementations for computing lower bounding distances between the Piecewise Aggregate Approximation (PAA) [29] and indexable Symbolic Aggregate Approximation (iSAX) [10] representations.
- Finally, we experimentally evaluate ParIS and ParIS+ using a variety of synthetic and real datasets. The results demonstrate the efficiency of the proposed approach, which is orders of magnitude faster for exact query answering than the state-of-the-art methods. Moreover, the results show that, in settings of more than 4 cores, ParIS+ completely hides the CPU time during index creation.

## 2 PRELIMINARIES

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

### 2.1 Data Series and Similarity Search

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

**[Similarity Search]** Analysts perform a wide range of data mining tasks on data series including clustering [30], classification and deviation detection [31], [32], and frequent pattern mining [33]. Existing algorithms for executing these tasks rely on performing fast similarity search across the different series. Thus, efficiently processing nearest neighbor (NN) queries is crucial for speeding up the above tasks. NN queries are formally defined as follows: given a query series  $S_q$  of length  $n$ , and a data series collection  $\mathcal{S}$  of sequences of the same length,  $n$ , we want to identify the series  $S_c \in \mathcal{S}$  that has the smallest distance to  $S_q$  among all the series in the collection  $\mathcal{S}$ . Figure 1 depicts an example of a query series and a candidate answer (the 1-NN, in this case).

Common distance measures for comparing data series are Euclidean Distance (ED) [34] and dynamic time warping (DTW) [11]. While DTW is better for most data mining tasks, the error rate using ED converges to that of DTW as the dataset

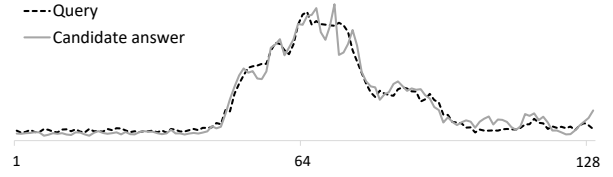


Fig. 1. Query series and candidate answer (length 128; SALD dataset)

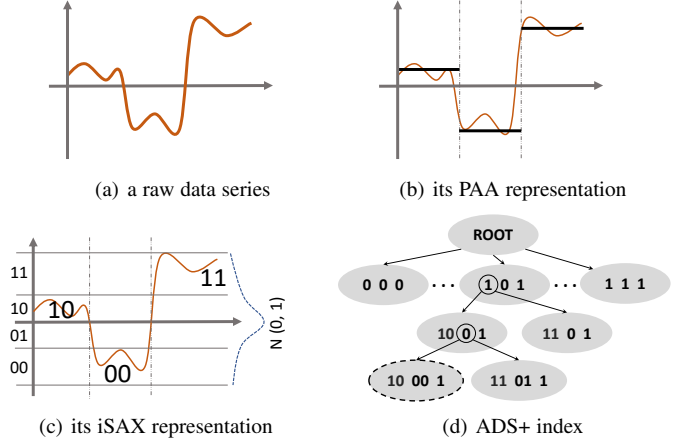


Fig. 2. The iSAX representation, and the ADS+ index structure

size grows [10]. Therefore, data series indices for massive datasets use ED as a distance metric [10], [11], [12], [13], [9], though simple modifications can be applied to make them compatible with DTW [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 [35].

**[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 [36]. 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. Modern CPUs support 256-bit wide SIMD vectors, which means that some floating point (or other 32-bit data) computations can be up to 8 times faster when executed using SIMD [36]. Even though no SIMD solutions have been proposed so far for data series indices, this idea has been exploited for the computation of distance functions [37]. In our study, we take an extra step, and we also use SIMD for operations related to the proposed data series index structure (i.e., for conditional branch calculations during the computation of the lower bound distances; see Section 3.3).

### 2.2 iSAX Representation and ADS+ Index

**[iSAX Representation]** The iSAX representation is based on the Piecewise Aggregate Approximation (PAA) representation [29], which divides the data series in *segments* of equal length, and uses the mean value of the points in each segment in order to summarize a data series. Figure 2(b) depicts an example of PAA representation with three segments (depicted with the black horizontal lines), for the data series depicted in Figure 2(a). Based on PAA, the indexable Symbolic Aggregate approxXimation (iSAX) representation was proposed in [10].

2. A preliminary version of this work appeared in [28].

This method first divides the (y-axis) space in different regions, and assigns a bit-wise symbol to each region. In practice, the number of symbols is small: iSAX achieves very good approximations with as few as 256 symbols, the maximum alphabet cardinality, which can be represented by 8 bits [13]. It then represents each segment of the series with the symbol of the region the PAA falls into, forming the word  $10_200_211_2$  shown in Figure 2(c) (subscripts denote the number of bits used to represent the symbol of each segment).

For an overview of iSAX-based indices, see [21].

**[ADS+ Index]** Based on this representation, the state-of-the-art ADS+ index was developed [9]. It makes use of variable cardinalities (i.e., variable degrees of precision for the symbol of each segment; see Figure 2(c)) in order to build a hierarchical tree index (see Figure 2(d)), consisting of three types of nodes: (i) the root node points to several children nodes,  $2^w$  in the worst case (when the series in the collection cover all possible iSAX representations), where  $w$  is the number of segments; (ii) each inner node contains the iSAX representation of all the series below it, and has two children; and (iii) each leaf node contains both the iSAX representation *and* the raw data of all the series inside it (in order to be able to prune false positives and produce exact, correct answers). When the number of series in a leaf node becomes greater than the maximum leaf capacity, the leaf splits: it becomes an inner node and creates two new leaves, by increasing the cardinality of the iSAX representation of one of the segments (the one that will result in the most balanced split of the contents of the node to its two new children [13], [9]). The two refined iSAX representations (new bit set to 0 and 1) are assigned to the two new leaves. In our example, the series of Figure 2(c) will be placed in the outlined node of the index (Figure 2(d)).

The ParIS and ParIS+ indices use the iSAX representation and basic ADS+ index structure, but implement techniques and algorithms specifically designed for multi-core architectures.

### 3 PROPOSED SOLUTION: PARIS AND PARIS+

In this section, we describe our approach, called Parallel Indexing of Sequences (ParIS), for parallel index construction and query answering. We then present, ParIS+, an improved version of ParIS (in settings with reasonable levels of parallelism).

Figure 3 provides a high level overview of the entire pipeline of how the ParIS index is created and then used for query answering. This pipeline is comprised of four stages. In Stage 1, a thread, called the *Coordinator* worker, reads raw data series from the disk and transfers them into the *raw data buffer* in main memory. In Stage 2, a number of *IndexBulkLoading* workers, process the data series in the raw data buffer to create their iSAX summarizations. Each iSAX summarization determines to which root subtree of the tree index the series belongs. Specifically, this is determined by the first bit of each of the  $w$  segments of the iSAX summarization. The summarizations are then stored in one of the index Receiving Buffers (RecBufs) in main memory. There are as many RecBufs as the root subtrees of the index tree, each one storing the iSAX summarizations that belong to a single subtree. This number is usually a few tens of thousands and at most  $2^w$ , where  $w$  is the number of segments in the iSAX representation of each time series ( $w$  is fixed to 16 in this paper, as in previous studies [9]). The iSAX summarizations are also stored in the array SAX (used during query answering).

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

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

#### 3.1 Index Construction: ParIS

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

##### 3.1.1 Index Initialization

In this section, we describe Stages 1 and 2. Figure 4(a) summarizes how the coordinator and IndexBulkLoading workers work.

The raw data buffer is implemented using double buffering. So, it is comprised of two parts, one on which the *Coordinator* works, and another on which the *IndexBulkLoading* workers work. In this way, the data the *Coordinator* is accessing and the data the *IndexBulkLoading* workers are handling form two independent sets. So, all these threads work in parallel (as much as possible). Our tuning experiments (refer to Figure 11) showed that setting the size of the double buffer to 2MB results in the best performance (the time cost reduces as the buffer size increases until we reach 2MB and then it stabilizes).

The pseudocode for the *Coordinator* worker is shown in Algorithm 1. We assume that the *index* variable is a data structure containing all buffers, a pointer to the root of the tree index, some arrays of locks that are needed for synchronizing access to RecBufs, and SAX. In this algorithm,  $B_1$  and  $B_2$  are pointers to the two parts,  $TS[0]$  and  $TS[1]$ , of the raw data buffer. Moreover, we denote by  $n_t$  the number of *IndexBulkLoading* workers that are created by the coordinator (see discussion below about the value of  $n_t$ ). The algorithm works as follows. The *Coordinator* worker first fills in the part of the raw data buffer pointed to by  $B_1$  (line 3). Then, the *Coordinator* worker creates the  $n_t$  *IndexBulkLoading* worker threads (lines 4). These threads create the iSAX summarizations of the data in the raw data buffer part pointed to by  $B_1$  and place them in the appropriate RecBufs and in SAX (see Figure 4(a)); for each data series, we also store in RecBuf its offset in the raw data file. While the *IndexBulkLoading* workers are performing this task, the *Coordinator* concurrently fills in the other part of the raw data buffer (line 8). This process is repeated until the main memory is exhausted.

The *Coordinator* worker is aware of the current memory usage by monitoring the number of data series that it has processed.

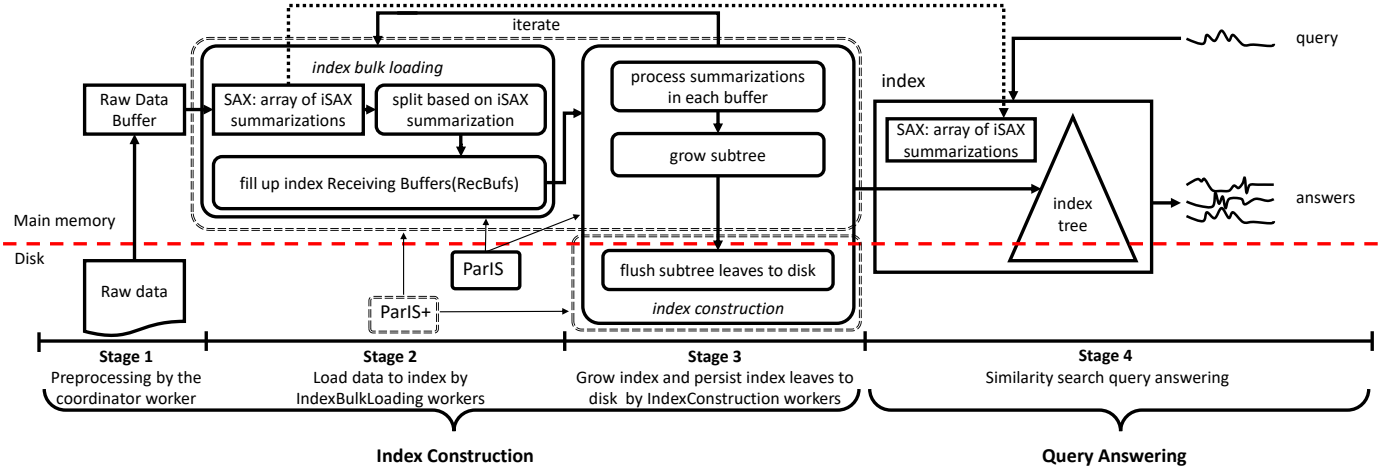


Fig. 3. Overview of the pipeline for creating the ParIS/ParIS+ index, and using the index for query answering.

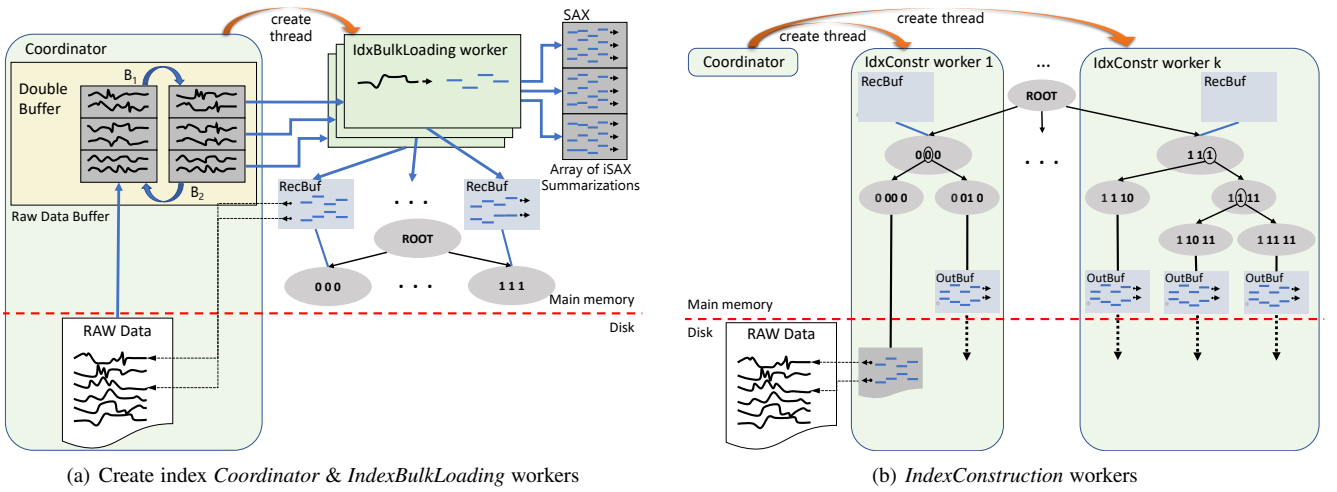


Fig. 4. Workflow and algorithms relevant to index creation.

When the available memory is (nearly) exhausted<sup>3</sup> (line 10), then the *Coordinator* creates the *IndexConstruction* worker threads (lines 12), which build the part of the index that corresponds to the iSAX summarizations stored in the RecBuf, and flush the leaf nodes of the tree to disk.

The pool of *IndexBulkLoading* workers could be as big as the number of cores in our machine (minus one which is reserved for the *Coordinator*). The *IndexBulkLoading* workers are assigned each RecBuf one-at-a-time in round-robin fashion, by using either an atomic fetch and increment primitive, or a lock. As we will discuss later (in Section 4), for ParIS we see the best performance when we use five *IndexBulkLoading* workers and six *IndexConstruction* workers; note that these numbers are orders of magnitude less than the number of the index root subtrees (usually tens of thousands). Note that because of the small number of *BulkIndexLoading* (and *IndexConstruction* workers), the use of locks for synchronizing access to RecBufs (or the assignment of subtrees) does not result in any synchronization bottlenecks.

3. Note that we only need a small amount of additional memory for creating new index nodes in the subtree of the root currently being processed, which can have a maximum depth of  $w(|alphabet| - 1)$  [9], where  $|alphabet|$  is the cardinality of the alphabet. Moving data inside the index (e.g., from RecBuf to OutBuf, as we will discuss later) does not require extra memory: we reallocate the same memory addresses between the buffers.

#### Algorithm 1: Coordinator

---

**Input:** File\* *file*, Index *index*, Integer  $n_t$

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

---

Moreover, because the computation is heavily I/O bounded at this stage, the performance does not degrade even if the *Coordinator* creates the *IndexBulkLoading* workers from scratch each time it fills up a part of the raw data buffer. For the same reason, techniques like thread pinning does not improve performance.

**Algorithm 2: IndexBulkLoading**


---

**Input:** Index  $index$ , Raw data buffer  $TS[]$ , Integer  $p$

```

1 for  $i \leftarrow 0$  to  $chunksiz$  - 1 do
2    $index.SAX[p + i] = ConvertToSAX(TS[i]);$ 
3   acquire appropriate lock from  $index.RecBufLock[]$ ;
4    $InsertIntoRecBuf((index.SAX[p + i], p + i));$ 
5   release the acquired lock;
```

---

The pseudocode that the IndexBulkLoading workers execute is shown in Algorithm 2. Each such worker has been assigned a chunk, of size  $chunksiz$ , in each part of the raw data buffer (therefore, the size of the raw data buffer is  $2 * chunksiz * n_t$ ). Each worker operates only on its chunk. In this way, no synchronization is needed between the IndexBulkLoading workers for accessing the raw data buffer. Each IndexBulkLoading worker reads the data series in its chunk one after the other (line 2), and calculates the iSAX summarization for each of them by calling the ConvertToSAX() function (line 2). These summaries are stored in SAX, the Array of iSAX Summarizations (line 2), and in the appropriate RecBuf (line 4; refer also to Figure 4(a)). Recall that each RecBuf gathers together all data that must be stored into the same root subtree. These data may exist in chunks of the raw data buffer that are associated to different IndexBulkLoading workers. So, more than one such workers may require to concurrently access the same RecBuf. Therefore, synchronization is needed. This synchronization is achieved by using a lock for each such buffer, stored in array  $RecBufLock[]$  of  $index$ .

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

### 3.1.2 Subtree Construction and Leaf Materialization

We now describe Stage 3, where the index is gradually constructed and its leaves materialized. On top of the raw data buffer and the RecBufs, Paris+ makes use of an additional set of main memory buffers, the Output Buffers (OutBufs). Each OutBuf is associated to one leaf of the index tree and stores the iSAX representations of the data series and pointers to them in the raw data file.

The Coordinator worker creates a number of IndexConstruction workers when it discovers that the main memory is exhausted. (Based on our experiments, the best number of IndexConstruction workers is 6.) These workers process the data in the RecBufs in order to grow the corresponding subtree, until the data end up in the OutBufs of that subtree. Finally, the OutBufs are flushed to disk. This process is illustrated in Figure 4(b), (where we have assumed that the contents of the OutBuf for the leftmost leaf have been flushed to disk, whereas the rest OutBufs have not).

All IndexConstruction workers process different root subtrees, so they work independently and no synchronization is needed. A worker that finishes its work on one subtree gets assigned to a new RecBuf, until all RecBufs are processed. In order to maintain the scheme simple and efficient, we have chosen not to parallelize processing inside each one of the index root subtrees since that would require a lot of synchronization (due to node splitting). Our experiments have shown that this decision does not have any negative impact in the performance of our scheme.

The pseudocode that the IndexConstruction workers execute is shown in Algorithm 3. An IndexConstruction worker first selects one of the RecBufs to process in an atomic way (line 3). This

**Algorithm 3: IndexConstruction**


---

**Input:** Index  $index$

```

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

---

can be done by using either an atomic *fetch and increment* primitive ( $n_b$  in Algorithm 3), or a lock. Then, it moves the data to the appropriate OutBuf in the index (line 10), and if necessary (i.e., if the leaf node is full), it (repeatedly) performs node splitting (line 8). When node splitting is performed, the iSAX summarizations (i.e., the contents of the leaf node to be split) are read from disk and they are placed in the appropriate OutBuf (if they have already been flushed). Then, the leaf node is split to two new leaf nodes, the data of the original leaf are moved to the new leaves, and finally the OutBufs corresponding to the leaves of the subtree currently processed are flushed to disk (line 11).

We note that 80% of the leaves (and therefore also the corresponding OutBufs) have size less than the block size. So, flushing them to disk results in disk random accesses. For this reason, the use of a lock to synchronize disk accesses of threads during leaf materialization would cause performance degradation.

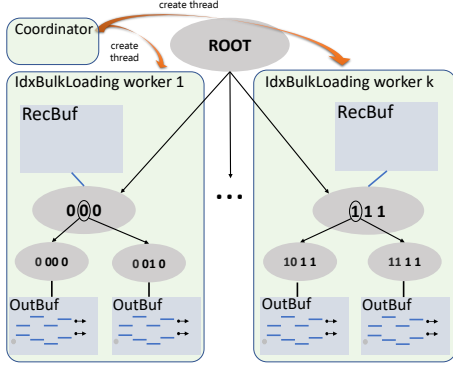
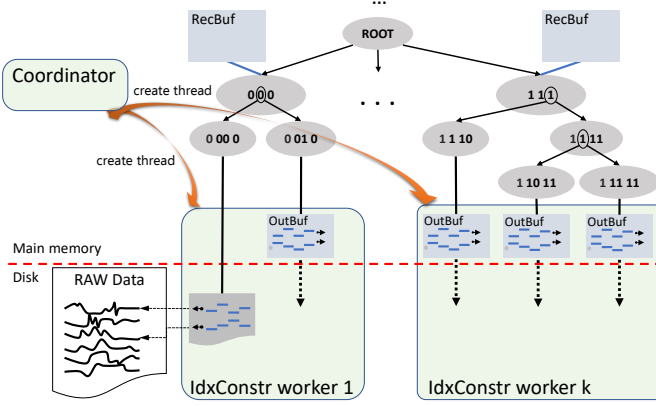
## 3.2 Index Construction: Paris+

In this section, we present Paris+, which improves Paris by completely masking out the CPU cost when creating the index. This is not true for Paris, whose index creation (stages 1-3) is not purely I/O bounded (as we will see in Figure 9). The reason for this is that, in Paris, the IndexConstruction workers do not work concurrently with the Coordinator worker. Moreover, the IndexBulkLoading workers do not have enough CPU work to do to fully overlap the time needed by the Coordinator worker to read the raw data file.

Paris+ (Algorithms 4-6) is an optimized version of Paris, which achieves a complete overlap of the CPU computation with the I/O cost. In Paris+, the IndexBulkLoading workers have undertaken the task of building the index tree, in addition to performing the tasks of stage 2. The IndexConstruction workers now simply materialize the leaves by flushing them on disk.

In Paris+, the Coordinator worker (Algorithm 4) creates the IndexBulkLoading workers right after it has finished filling in one part of the raw data buffer for the first time (line 2). Note that before starting to fill in this part again, the Coordinator reaches a barrier (line 8 of Algorithm 4), which ensures that the IndexBulkLoading workers have finished processing it (line 10 of Algorithm 5). An additional barrier (line 14 of Algorithm 4) between the Coordinator and each of the IndexBulkLoading workers is necessary to ensure that no IndexBulkLoading worker accesses the OutBuf buffers as long as the IndexConstruction workers operate on them.

Algorithm 5 provides pseudocode for the IndexBulkLoading workers. Note that the IndexBulkLoading workers have to reach

(a) *IndexBulkLoading* worker after synchronization barrier

(b) index construction

Fig. 5. Workflow and algorithms relevant to ParIS+ index building.

a barrier (line 10) after they finish processing the part of the raw data buffer they have been assigned. This barrier is necessary since more than one *IndexBulkLoading* worker adds items in each of the *RecBufs*, and the tree index construction should start only after all of them have finished their current phase of adding items in the *RecBufs*. Additional barriers (lines 22 and 23) ensure the necessary synchronization of the *IndexBulkLoading* workers with the Coordinator thread (as discussed above).

The *IndexConstruction* workers (Algorithm 6) simply flush the *Outbufs* of the leaves of each subtree of the index tree on disk. This is depicted in Figure 5.

Finally, we note that we also considered an alternative, where the *IndexBulkLoading* workers were performing their work concurrently with the tree index construction phase performed by the *IndexConstruction* workers. However, this design did not have any positive impact in performance, and thus, do not discuss it further.

### 3.3 Query-Answering

In this section, we describe methods for parallel query-answering.

The algorithm first performs an *approximate search* to obtain the first Best-So-Far (BSF) answer, and then proceeds with a sequential scan of the raw data that could not be pruned using the BSF, in order to produce the exact, final answer to the query. The approximate search is really fast, requiring only a negligible percentage (a few msec) of the (mostly) on-disk sequential scan cost. It is a simple, in-memory path traversal from the index root

#### Algorithm 4: *Coordinator (ParIS+)*

---

**Input:** File\* *file*, Index *index*, Integer  $n_b$ , Integer *chunksize*

- 1 Pointer  $B_1 \leftarrow index.TS[0]$ ,  $B_2 \leftarrow index.TS[1]$ ;
- 2  $B_1 \leftarrow$  read data from *file*;
- 3 for  $i \leftarrow 0$  to  $n_w - 1$  do
- 4     create a thread to execute an instance of *IndexBulkLoading*(*index*,  $i$ ,  $n_w$ , *chunksize*);
- 5 while not reached end of file do
- 6      $B_2 \leftrightarrow B_1$ ;
- 7      $B_1 \leftarrow$  read data from *file*;
- 8     Barrier to synchronize with the *IndexBulkLoading* workers;
- 9     if main memory is full then
- 10         for  $i \leftarrow 0$  to  $n_w$  do
- 11             create a thread to execute an instance of *IndexConstruction*(*index*);
- 12             Wait for *IndexConstruction* workers to finish;
- 13         for  $i \leftarrow 0$  to  $n_w - 1$  do
- 14             Barrier to synchronize with *IndexBulkLoading* worker  $i$ ;
- 15     Kill *IndexBulkLoading* workers;

---

#### Algorithm 5: *IndexBulkLoading (ParIS+)*

---

**Input:** Index *index*, Integer *id*, Integer  $n_w$ , Integer *chunksize*

- 1 Shared integer  $f_w = 0$ ;
- 2 Integer  $p = id * chunksize$ ,  $cnt = 1$ ;
- 3 Boolean *toggle* = 0;
- 4 while (TRUE) do
- 5     for  $i \leftarrow 0$  to *chunksize* - 1 do
- 6          $index.SAX[p + i] = ConvertToSAX$   
           ( $index.TS[toggle][i]$ );
- 7         acquire appropriate lock from *index.RecBufLock*[];
- 8         InsertIntoRecBuf ( $\langle index.SAX[p + i], p + i \rangle$ );
- 9         release the acquired lock;
- 10        Barrier to synchronize the *IndexBulkLoading* workers with one another;
- 11        while (TRUE) do
- 12             $i \leftarrow$  Atomically fetch and increment  $f_w$ ;
- 13            if ( $i \geq cnt * 2^w$ ) then break;
- 14            for every (*isax*, *pos*) pair  $\in index.RecBuf[i]$  do
- 15                 $targetLeaf \leftarrow$  Leaf of *index* tree to insert  $\langle isax, pos \rangle$ ;
- 16                while  $targetLeaf$  is full do
- 17                    SplitNode( $targetLeaf$ );
- 18                     $targetLeaf \leftarrow$  New leaf to insert  $\langle isax, pos \rangle$ ;
- 19                    Insert ( $\langle isax, pos \rangle$ ) in  $targetLeaf$ 's *OutBuf* buffer;
- 20             $p = p + n_b * chunksize$ ;
- 21             $toggle = 1 - toggle$ ;
- 22            Barrier to synchronize the *IndexBulkLoading* workers with one another and with the Coordinator worker;
- 23            Barrier to synchronize this *IndexBulkLoading* worker with the Coordinator worker;

---

to the leaf with the iSAX representation that is the most similar to that of the query. Once a leaf is reached, the distance between the query and each of the leaf's data series is calculated. The minimum distance found is the first BSF answer (see left part of Figure 7). This BSF is used to prune the candidate series by computing lower bound distances to their summarizations. The series that are not

#### Algorithm 6: *IndexConstruction (ParIS+)*

---

**Input:** Index *index*

- 1 Shared integer  $f_c = 0$ ;
- 2 while (TRUE) do
- 3      $i \leftarrow$  Atomically fetch and increment  $f_c$ ;
- 4     if ( $i \geq 2^w$ ) then break;
- 5     For each leaf in the subtree rooted at the  $i$ -th root child;
- 6         Flush leaf's *OutBuf* buffer to disk;
- 7         Clear the *OutBuf* buffer;

---

pruned will be visited in the raw file, and the true distance will be computed (the BSF may be updated during this phase).

In the following, we concentrate on our algorithm for parallelizing the scan phase. We first describe how we exploit SIMD for performing the lower bound distance calculations. Then, we present, in Section 3.3.2, a simple technique for further parallelizing this phase (nb-ParIS+, which stands for non-balanced ParIS+), which however does not result in optimal performance, because of the lack of load balancing. Finally, we present in Section 3.3.3 our proposed method for exact search in ParIS+ (the exact search algorithm for ParIS is the same).

### 3.3.1 Lower-Bound Distance Calculation

The algorithm starts by calculating the lower bound distance between the query series and the iSAX summarizations of all series in the index. This is a main memory operation, since the iSAX summarizations are small enough to fit in the memory of modern servers<sup>4</sup>. This is a procedure that we execute using SIMD, since both the queries and the index series are vectors, on which we perform the same operation (i.e., a distance calculation).

Using SIMD, we can perform eight calculations in parallel, using a single instruction (we assume 256-bit SIMD vectors, containing 8 32-bit float elements). We need to implement a conditional branch in SIMD, but contrary to previous solutions [37], this is a complex branch: not only do we have to use different conditional branches for different positions in the SIMD vector, but also need to make different assignments for different branches.

In our case, the calculation of the lower bound distance between the PAA of the query series and an iSAX summarization has 3 branches (conditions): checking whether the PAA lies (i) *ABOVE*, (ii) *BELOW*, or (iii) *IN* the iSAX interval. Thus, we need to choose different values from different dictionaries in order to perform the distance computations in SIMD (see Figure 6). We first calculate the distance results of the above 3 branches for every point in the SIMD vector. We then use a conditional mask to extract the results in the correct branch.

In particular, we generate 3 branch masks, i.e., *ABOVE*, *BELOW*, and *IN*. These masks contain a value of *true* (i.e., 1) only in the SIMD vector positions for which the corresponding branch is true. In Figure 6 for example, the first query PAA segment is above the corresponding candidate series iSAX representation, which means that only the *ABOVE* mask will be true for this position; consequently we will consider the *Dist\_ABOVE* distance value for this position of the SIMD vector. Using the appropriate SIMD instruction (AVX, AVX2 and SSE3) [38], we can efficiently calculate the value of the 3 branch masks. Next we apply a logical "AND" between the 3 branch results and their masks. After that, all bits of the branch result in the wrong branch will be zero. Now there is only one value at the same position in those 3 branch results. Finally, we merge all possible branches in one vector, which is the correct final result.

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

4. The highest granularity iSAX summarizations for 1 billion data series (occupying 1TB on disk) only need about 10GB of space in main memory.

---

#### Algorithm 7: nb-ParIS+: *ExactSearch*

---

**Input:** querySeries  $QTS$ , query iSAX  $isax$ , Index  $index$ , File\*  $file$   
**Output:**  $realDistance$

- 1 float  $BSF, V_{bsf}[]$ ; // Perform an approximate search
- 2  $BSF = approxSearch(QTS, isax, index)$ ; // distribute BSF into  $V_{bsf}$
- 3  $V_{bsf} \leftarrow BSF$ ; // calculate  $minDist$  and  $realDist$  in parallel
- 4 create a number of threads, each executing an instance of  $DistCompWorker(TS, isax, V_{bsf}$ , appropriate part of  $index.SAX, file)$ ;
- 5 Wait for all threads to finish;
- 6 **return** ( $min(V_{bsf})$ );

---



---

#### Algorithm 8: nb-ParIS+: *DistComp*

---

**Input:** querySeries  $TS$ , query iSAX  $isax$ , float  $V_{bsf}[]$ , iSAX summarizations  $SAX\_part[]$ , Index  $index$ , File\*  $file$   
**Output:**  $BSF$

- 1 float  $BSF = read\ initial\ BSF\ value\ from\ V_{bsf}$ ;
- 2 **for**  $i \leftarrow 1$  **to** size of  $SAX\_part$  **do**
- 3      $minDist = LowerBound\_SIMD(TS, SAX\_part[i])$ ;
- 4     **if**  $minDist < BSF$  **then**
- 5         Move file pointer to appropriate position in  $file$ ;
- 6          $rawData = read\ raw\ data\ series\ from\ file$ ;
- 7          $realDist = Dist(rawData, TS)$ ;
- 8         **if**  $realDist < BSF$  **then**
- 9              $BSF \leftarrow realDist$ ;

---

### 3.3.2 Exact Search in nb-ParIS+

We now present nb-ParIS+ that served as an intermediate step for developing ParIS+, using a simple design with no communication among the distance computation worker threads (see also § 3.3.4).

Exact Search in nb-ParIS+ is illustrated in Figure 8, and shown in Algorithm 7. It employs a standard parallelization technique, which splits SAX in blocks and has different workers, called *Distcomp workers*, work on different blocks independently. When a thread  $t$  executes an *ExactSearch* (Algorithm 7), it first performs an approximate search to get the initial BSF value (line 2). BSF is used for pruning. Each *Distcomp* worker updates its own copy of BSF to store the minimum distance it has calculated so far. This copy is stored in an appropriate element of vector  $V_{bsf}$ . Note that since each worker calculates its own estimate of BSF, no synchronization is needed among them.

When  $t$  creates the *Distcomp* workers (line 4), it informs them about the initial BSF value through vector  $V_{bsf}$ . Each such worker computes the lower bound distance between the query PAA and each iSAX summarization in its SAX part (Algorithm 8, line 3). It does so using the SIMD approach we described in Section 3.3.1. If this distance is higher than the current value of the worker's copy of BSF, then the data series is pruned. Otherwise, the *Distcomp* worker reads the required data from disk, calculates the real distance (line 7), and if necessary, updates the appropriate element of  $V_{bsf}$  (line 9). Finally,  $t$  waits for all *DistComp* workers to finish, calculates the minimum value stored by these workers in  $V_{BSF}$ , and returns this value. We use one *DistComp Worker* thread per core (thus resulting in 24 *DistComp* workers in total). Note that nb-ParIS+ does not necessarily balance the work among the *DistComp* workers, since the pruning degree may turn out to be different for each worker. Moreover, different threads produce disk requests concurrently, which results in random accesses to disk. ParIS+ improves upon nb-ParIS+ to address these problems.

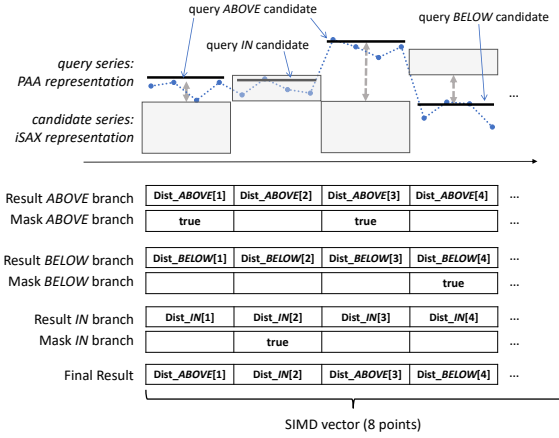


Fig. 6. SIMD conditional branch calculation.

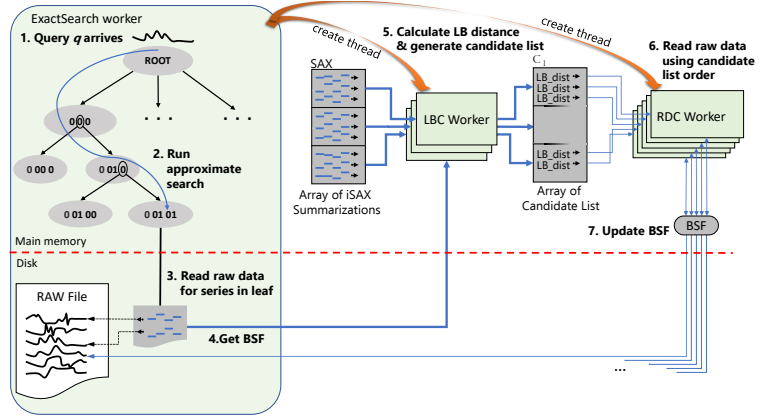


Fig. 7. Workflow and algorithms for query answering with ParIS+ (balanced).

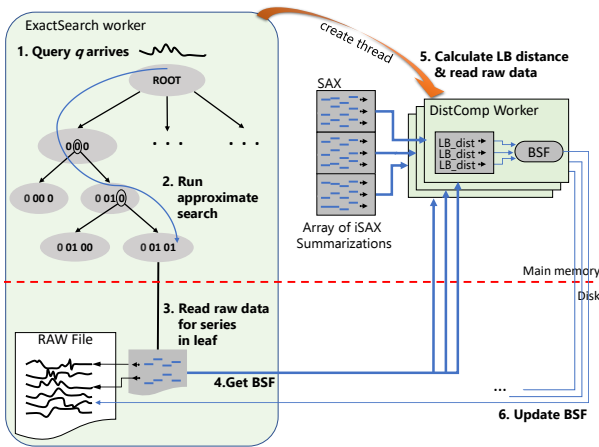


Fig. 8. Workflow and algorithms for query answering with nb-ParIS+ (non-balanced).

### 3.3.3 Exact Search in ParIS+

As in nb-ParIS+, the exact search algorithm in ParIS+<sup>5</sup> employs approximate search as a first step and uses the approximate answer as the initial BSF (see Algorithm 9). Unlike to nb-ParIS+ though, BSF is now stored in a variable shared by all workers.

We note that the state-of-the-art *sequential* index similarity search algorithm spends more than 95% of its time on I/O (on our server, described below) and in particular, on reading data from disk. In order to achieve better parallelism, the ExactSearch algorithm separates the phase of the lower bound calculation from that of the real distance calculation and has two types of worker threads, namely the Lower Bound Computation (LBC) and the Real Distance Computation (RDC) workers, respectively, executing each type of calculation (see right part of Figure 7).

When a thread  $t$  executes an ExactSearch (Algorithm 9), it first performs an approximate search to get the initial BSF answer (line 3), and then it initiates a number of LBC workers (line 5). Different LBC workers work on different parts of SAX. Each such worker computes the lower bound distance between the query PAA and each iSAX summarization in its SAX part and records the data series for which this distance is less than the current BSF in a local candidate list, which it eventually returns to  $t$  (see

5. Note that the exact search algorithm of ParIS is the same as in ParIS+.

### Algorithm 9: ParIS+: *ExactSearch*

**Input:** querySeries  $QTS$ , query iSAX  $isax$ , Index  $index$ , File\*  $file$

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

Algorithm 10). This list contains the position and the lower-bound distance, needed to read the raw data and to calculate the real distance for the data series. Once all LBC workers have finished,  $t$  merges the candidate lists they have created (Algorithm 9, line 6) and initiates the RDC Workers (line 7).

Each RDC Worker (Algorithm 11) repeatedly retrieves a (minDistance, position) pair from the merged candidate list ( $C_l$ ) in an atomic way (line 2). Atomicity is achieved with the use of a lock which all RDC workers share. The worker then reads the required data from disk, calculates the real distance (line 6), and if necessary, updates the shared BSF variable (line 8). A thread lock ensures that the BSF modification is done atomically. Storing BSF in shared memory and updating it during the course of the execution contributes towards reducing the number of calculations that RDC workers perform.

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

### 3.3.4 Discussion of nb-ParIS+ and ParIS+

Using nb-ParIS+, we were able to identify some design choices that resulted in bad performance during query answering. Specif-



**Algorithm 10:** ParIS+: *LBC* – *Worker*


---

**Input:** querySeries  $QTS$ , iSAX summarizations  $SAX\_part[]$ , float  $BSF$

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

---

**Algorithm 11:** ParIS+: *RDC* – *Worker*


---

**Input:** querySeries  $QTS$ , candidate list  $C_l$ , float  $BSF$ , File\*  $file$

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

---

ically, nb-ParIS+ needs synchronization between the different threads only for computing the minimum BSF value, but may result in load imbalance in terms of real distance calculations performed in each chunk. Since each real distance calculation performs I/O (to read the raw data series), some threads may finish much later than others. Moreover, as the requests of different threads are interleaved, nb-ParIS+ may perform random I/Os. ParIS+ addresses these problems in common cases (when the pruning ratio is large), by separating the phase of lower bound distance calculations from that of real distance calculations through the use of the candidate list. The candidate list is sorted to ensure that random accesses to disk are minimized. Moreover, a fetch&add is used to assign entries of the candidate list to threads for processing, in order to achieve load balancing. In this way, it is ensured that all threads finish at about the same time.

## 4 EXPERIMENTAL EVALUATION

**[Setup]** We ran the experiments on two servers, whose physical memory was limited to 75GB<sup>6</sup>. The first server (default) comprises two Intel Xeon E5-2650 v4 2.2Ghz processors with 12 cores each, and has 10.8TB (6 x 1.8TB) 10K RPM SAS HDD drives in RAID0, with sequential access throughput of the RAID0 array being 1200MB/sec and random access throughput 12MB/sec. The second server, with the same setup for CPUs and memory, has 3.2TB (2 x 1.6TB) SATA SSD drives in RAID0, with 500MB/sec sequential throughput and 450MB/sec random access throughput.

All algorithms were implemented in C, and compiled using the GCC6.2.0 with the O3 optimization flag on Ubuntu Linux 16.04. Unless otherwise mentioned, in our experiments we use one socket for index creation and two sockets for query answering.

**[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.

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

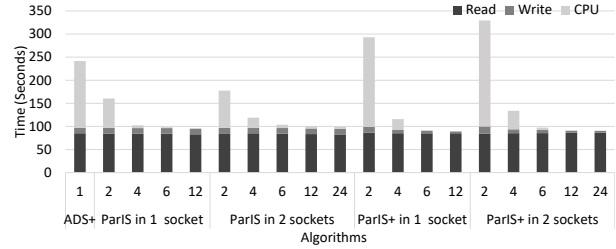


Fig. 9. Index creation time (HDD) as the number of cores increases.

We used synthetic datasets of sizes 50GB-250GB (default size: 100GB), and a random walk data series generator that works as follows: a random number is first drawn from a Gaussian distribution  $N(0,1)$ , and then at each point in time a new number is drawn from this distribution and added to the value of the last number. This generator has been extensively used in the past (and has been shown to model real-world financial data) [39], [10], [12], [13], [9]. We used this process to generate 100 query series.

For the *Seismic* real dataset, we used the IRIS Seismic Data Access repository [40] to gather 100M series representing seismic waves from various locations, for a total size of 110GB. The *SALD* real dataset includes neuroscience MRI data series [41], for a total of 200M series of length 128 points each, and total size 100 GB. In both cases, we used as queries 100 series that were not part of the datasets (produced using our synthetic series generator, since these datasets do not come with query workloads).

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

**[Algorithms]** We experiment with our ParIS and ParIS+ algorithms, and compare those to the sequential state-of-the-art data series index, ADS+ [9]. We also compare to (i) the UCR Suite [11], the state-of-the-art, optimized serial scan technique for exact similarity search, and (ii) DS-Tree [12], a modern data series index that stores the raw data in the leaves. All algorithms are available online [42]. For the disk-resident experiments, we never load the datasets in main memory. In order to mitigate the effects of caching, we clear the caches before each experiment (i.e., before running index creation and before executing each query).

### 4.1 Results

We present the performance results for ParIS/ParIS+, and compare them to two modern data series indices, ADS+ and DS-Tree.

#### 4.1.1 Index Creation Performance Evaluation

In our first experiment (Figure 9), we evaluate the time it takes to create the tree index for a synthetic dataset of 100M series. The figure illustrates that the performance of ParIS and ParIS+ improves as the number of cores grows from 4 to 6 (note that a single thread runs on each core); after 6 cores the improvement is rather small. The reason for this behavior is illustrated in Figure 10. Note that there are 4 types of time costs: (i) read raw data from disk; (ii) write raw data to disk; (iii) CPU time by IndexBulkLoading workers; and (iv) CPU time by Index-Construction workers. When we use more than one core, the

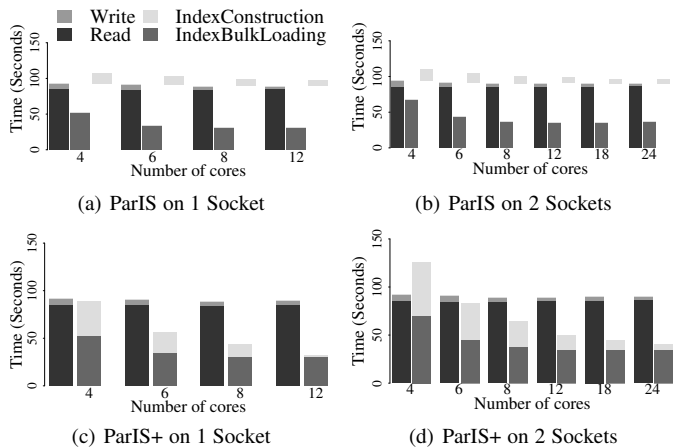


Fig. 10. Overlap of I/O time and CPU time during index creation (HDD).

time to read the data from disk and the management of data series are performed concurrently. Moreover, the time cost for the management of data series decreases with the number of cores, since the data that each core needs to process gets reduced. The cost of the index construction also reduces. However, the time cost to read data is always the same, since we access the same disk.

Figure 9 shows that ParIS results in performance which is up to 2.4x faster than ADS+. Still, ParIS does not completely hide the CPU latency. This is achieved by ParIS+, when 6 or more threads are used, as can be seen in Figures 9 and 10. Note that in ParIS+, more work is performed than in ParIS, because the IndexBulkLoading workers traverse the tree more than once. This cost is more evident in the 2 sockets case, where the threads do not benefit from the use of the L3 cache. However, ParIS+ achieves better overlap of CPU time with I/O cost (Figure 10). Therefore, the time to execute the additional work completely overlaps with the I/O cost when the number of threads is at least 6, and ParIS+ achieves better performance than ParIS.

Overall, these results demonstrate that not only does the proposed solution completely hide the CPU latency (using  $\geq 6$  cores), but it will continue to do the same when the storage medium of the dataset becomes much faster, e.g., with NVRAMs. In the following, we use 6 cores by default. The results with SSD follow the same trends (in this case ParIS+ completely hides the CPU latency when using  $\geq 4$  cores), and we omit them for brevity.

Figure 11 shows the impact of the double buffer size on performance (for the same experiment as in Figure 9). The results show that a good choice for the size of the raw data buffer is 1MB for ParIS, whereas it is 5MB for ParIS+. The reason for this difference is that as the buffer size increases, the IndexBulkLoading workers in ParIS+ traverse the index tree fewer times, and achieve better overlap with the work performed by the coordinator.

We now turn our attention to datasets of increasing size, and additionally compare ParIS to another competitive data series index, DS-Tree. Figures 12 and 13 depict the results for HDD and SSD, respectively. The results show that the performance of ParIS and ParIS+ is always better than that of ADS+ and DS-Tree. Moreover, ParIS+ is always faster than ParIS. This improvement is up to 7% on HDD. However, it is smaller on SSD because the SSD I/O bandwidth in our server is smaller than that of the HDD, resulting in higher read cost. However, the time to build the tree index does not change, and therefore, it now accounts for a smaller percentage of the I/O time. Note that the DS-Tree is always one

order of magnitude slower than the other approaches, so we do not consider the DS-Tree in the rest of our experiments.

#### 4.1.2 Query Answering Performance Evaluation

We now present results on ParIS+'s efficiency in query answering.

Figure 14 shows the exact query answering time for ParIS+, nb-ParIS+, and ADS+, as we vary the number of cores. We observe that the performance improves as we increase the number of cores (though the improvement is rather small when we go beyond 6 cores). For example, for 24 cores, nb-ParIS+ is no more than 2 times faster than ADS+, whereas ParIS+ is almost 6 times faster than ADS+.

Figure 15 shows how the time for executing the two stages of query answering in ParIS+ is influenced as we increase the number of cores and the number of threads running on each core. The results show that the LBC workers execution time decreases as the number of cores increases, with the degree of oversubscribing not playing an important role in performance (Figure 15(a)). On the contrary, for the RDC execution time the degree of oversubscribing is crucial, both for HDD (Figures 15(b)) and SSD (Figure 15(c)). The reason is that the LBC workers perform in memory computations, for which it is important to use more cores to execute them faster. On the other hand, the RDC workers perform I/O to read the required data from disk, and thus, oversubscribing is useful to keep the CPU busy at all times. These diagrams justify the use of 1 LBC worker per core and 5 RDC workers per core, which are the default values we have used here.

Figure 16 (log-scale y-axis) shows the performance of query answering for UCR Suite, ADS+, nb-ParIS+, and ParIS+ as the dataset size increases. We observe that nb-ParIS+ is about 2 times faster than ADS+ and about 20 times faster than UCR Suite in general. ParIS+ is much better than this: it is one order of magnitude faster than ADS+, and more than two orders of magnitude faster than UCR Suite. We also note that the performance improvement of ParIS+ gets larger with increasing dataset sizes, so ParIS+ is able to scale better than UCR Suite. This is because ParIS+ can effectively prune the search space, while UCR Suite always has to read all the data from disk.

Figure 17 (log-scale y-axis) shows the performance of exact query answering for the SSD server. All three algorithms, ADS+, nb-ParIS+, and ParIS+ benefit from the SSD's low random access latency. The performance improvement of ParIS+ is increasing with the size of the dataset (since the number of random disk accesses increases, too), achieving in our experiments performance up to 15x faster than ADS+, and 2000x faster than UCR Suite. (Note that nb-ParIS+ results in lower numbers: it is about 7.5x faster than ADS+ and up to 1000x faster than UCR Suite.)

**[Vectorial (SIMD) Lower Bound Distance Calculation]** In order to evaluate the effect on performance of our new lower bound distance calculation function that uses SIMD, we conducted an experiment that factors out the disk I/O cost: we measured the execution time of exact similarity search when all data are loaded in main memory. We compared our solution to the case where all computations are performed using Single Instruction Single Data (SISD). The results (refer to Table 1) show that the average time cost per lower-bounding calculation when using SIMD is 3.5x faster than the SISD solution. This is a non-negligible speedup, attributed to the large number of vectorial computations executed in data series similarity search (refer to Algorithm 10).

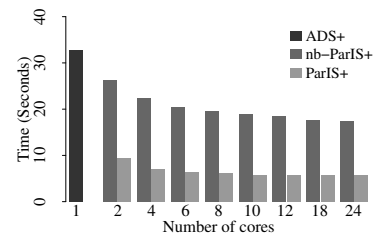
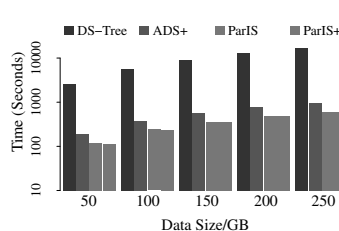
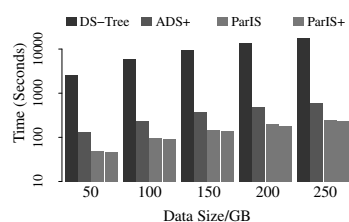
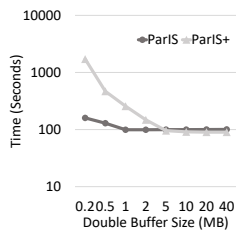
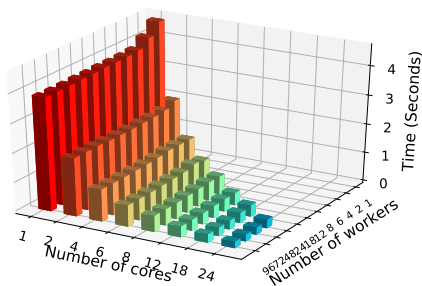


Fig. 11. Index creation time vs double buffer size.

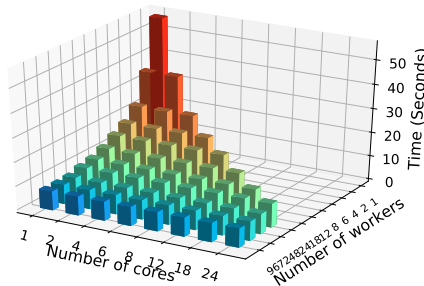
Fig. 12. Index creation time (HDD) vs dataset size.

Fig. 13. Index creation time (SSD) vs dataset size.

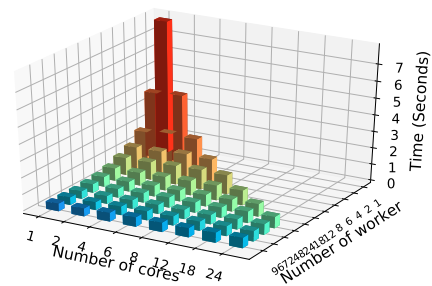
Fig. 14. Exact query answering time vs number of cores (HDD).



(a) Lower Bound Calculation (LBC) worker



(b) Real Dist. Calculation (RDC) worker - HDD



(c) Real Dist. Calculation (RDC) worker - SSD

Fig. 15. Time cost of Paris+'s query answering workers, varying the number of cores and the number of workers.

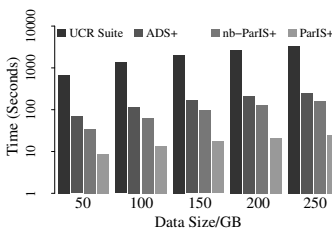


Fig. 16. Exact query answering time (HDD), varying dataset size.

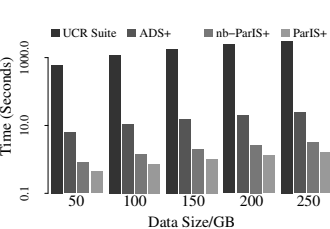


Fig. 17. Exact query answering time (SSD), varying dataset size.

TABLE 1  
Time cost of lower bound distance calculations.

Implementation technology	Time (nanoseconds)
SISD	107.5
SIMD	31.142

#### 4.1.3 Real Datasets

Figure 18(a) shows the result of index creation time cost on the SALD and Seismic real datasets. Similar to our previous results, Paris+ is faster than ADS+ during index creation: Paris+ is up to 2.4x faster for SALD, and 2x faster for Seismic. Moreover, Paris+ is slightly faster than Paris, as expected. Figure 18(b) (log scale y-axis) reports the exact similarity search time cost on HDD for UCR Suite, ADS+, nb-Paris+, and Paris+. The results differ for the two real datasets. For SALD, Paris+ is 140x faster than UCR Suite and 4x faster than ADS+, while for Seismic, Paris+ is 130x faster than UCR Suite and 5x faster than ADS+. The SSD experiments show similar, yet more pronounced trends (Figure 18(c), log scale y-axis): Paris+ is almost 1 order of magnitude faster than ADS+, and 3 orders of magnitude faster than UCR Suite.

We also observe that nb-Paris+ is much slower than Paris+ on HDD. Apart from the fact that nb-Paris+ does not manage in

an optimal way the disk read operations and the corresponding load in the individual worker threads, the major reason for its low performance is the lack of communication among the nb-Paris+ worker threads during the execution of the similarity search algorithm. (Recall that during query answering in nb-Paris+, the workers have a local copy of the BSF, while in Paris+, all workers share a common copy of BSF.) Consequently, when one worker finds a better BSF value that can help prune more data series, this value (contrary to Paris+) is not shared with the rest of the workers, who perform unnecessary expensive disk read operations. Figures 19(a) and 19(b), illustrate the above observation. The results show that the extra work for sharing a common BSF pays off for Paris+, since it leads to both a smaller number of BSF updates (i.e., we arrive to a better BSF earlier), and a reduced number of raw data to read (i.e., we prune more). This gives Paris+ an edge that is more pronounced on the HDD server, rather than on the SSD one: having to read fewer raw data translates to a smaller number of the expensive HDD seek/rotation operations.

We note that it is possible for nb-Paris+ to perform better than Paris+: this happens when the queries are very hard [43] and the resulting pruning ratio is small. In such cases, the creation and manipulation of the (long) candidate list results in high overheads, while the benefit of having all RDC worker threads of Paris+ communicating in order to update the BSF, which leads to saving some real distance computations, is not significant (results omitted for brevity). Usually though, the query workload is not very hard overall, which justifies the use of Paris+ as the method of choice.

## 5 RELATED WORK

**[Data series summarization and indexing]** Various dimensionality reduction techniques exist for data series, which can then be scanned and filtered [44], [45] or indexed and pruned [46], [47], [12], [10], [31], [9], [15], [17], [21] during query answering. We follow the same approach of indexing the series based on their

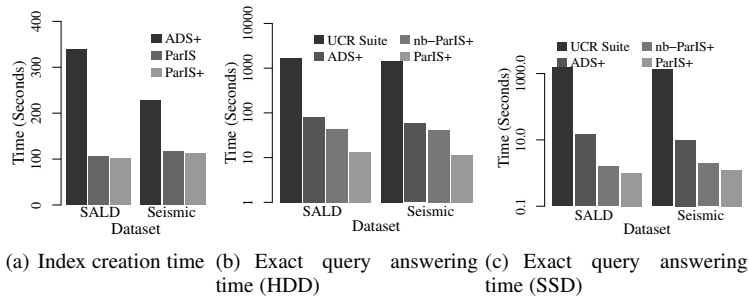


Fig. 18. Time cost for index creation and similarity search for real data.

summaries, though our work is the first to exploit the parallelization opportunities offered by multi-core architectures, in order to accelerate data series index construction and similarity search. FastQuery is an approach used to accelerate search operations in scientific data [48], 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.

**[Data structures for SIMD]** While the interest in using SIMD for improving performance is not new [49], there are still many algorithms that do not take advantage of this hardware characteristic. The problem of developing a SIMD-friendly B+-Tree index was recently studied [50], with a focus on a basic B+-Tree method, the k-ary search algorithm. For data series in particular, previous work has used SIMD for Euclidean distance computations [37]. In our work, we go beyond this straightforward use of SIMD, and we propose an algorithm that uses SIMD for the computation of lower bounds, which involve branching operations.

**[Modern Hardware]** Multi-core CPUs offer thread parallelism through multiple cores and simultaneous multi-threading (SMT). Thread-Level Parallelism (TLP) methods, like multiple independent cores and hyper-threads are commonly used to increase algorithm efficiency [51]. 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 [52]. However, this is designed for temporal data, which are 2-dimensional, while in our case, data series can have thousands of dimensions (i.e., the length of the sequence). Graphics Processing Units (GPUs) are another modern hardware option, which allows for massively parallel computations. A recent study described the use of GPUs for accelerating similarity search in a Trajectory Indexing system [53]. In this work, we do not use GPUs.

**[Scans vs indexing]** Even though recent works have shown that sequential scans can be performed efficiently [11], [54], such techniques are applicable when the dataset consists of a single, very long data series, and queries are looking for potential matches in small subsequences of this long series. Such approaches, in general, do not provide any benefit when the dataset is composed of a large number of small data series, like in our case. Therefore, indexing is required in order to efficiently support data exploration tasks, where the query workload is not known in advance.

## 6 CONCLUSIONS

We presented ParIS and ParIS+, the first data series indices that exploit multi-core architectures, leading to performance 2-3 orders of magnitude faster than previous approaches. In our future work we will study in more depth parallel I/O techniques [55], combine our approach with solutions developed for distributed systems [19],

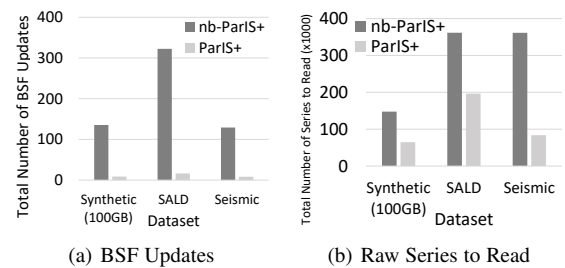


Fig. 19. Effort of ParIS+ and nb-ParIS+ (number of non-pruned raw data series).

extend it to support the DTW distance, and study other hardware parallelization opportunities, e.g., GPUs and FPGAs.

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

## REFERENCES

- [1] T. Palpanas, “Data series management: The road to big sequence analytics,” *SIGMOD Record*, 2015.
- [2] K. Zoumpatianos and T. Palpanas, “Data series management: Fulfilling the need for big sequence analytics,” in *ICDE*, 2018.
- [3] T. Palpanas and V. Beckmann, “Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA),” *SIGMOD Rec.*, vol. 48, no. 3, 2019.
- [4] A. J. Bagnall, R. L. Cole, T. Palpanas, and K. Zoumpatianos, “Data series management (dagstuhl seminar 19282),” *Dagstuhl Reports*, vol. 9, no. 7, 2019.
- [5] C. W. Tan, G. I. Webb, and F. Petitjean, “Indexing and classifying gigabytes of time series under time warping,” in *ICDM*, 2017.
- [6] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim, “The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art,” *PVLDB*, 2019.
- [7] P. Boniol, M. Linardi, F. Roncallo, and T. Palpanas, “Automated Anomaly Detection in Large Sequences,” in *ICDE*, 2020.
- [8] P. Boniol and T. Palpanas, “Series2Graph: Graph-based Subsequence Anomaly Detection for Time Series,” *PVLDB*, vol. 13, no. 11, 2020.
- [9] K. Zoumpatianos, S. Idreos, and T. Palpanas, “Ads: the adaptive data series index,” *VLDB J.*, vol. 25, no. 6, 2016.
- [10] J. Shieh and E. Keogh, “i sax: indexing and mining terabyte sized time series,” in *SIGKDD*, 2008.
- [11] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh, “Searching and mining trillions of time series subsequences under dynamic time warping,” in *SIGKDD*, 2012.
- [12] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang, “A data-adaptive and dynamic segmentation index for whole matching on time series,” *VLDB*, vol. 6, no. 10, 2013.
- [13] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh, “Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+,” *KAIS*, vol. 39, no. 1, pp. 123–151, 2014.
- [14] D.-E. Yagoubi, R. Akbarinia, F. Masegaglia, and T. Palpanas, “Dpisax: Massively distributed partitioned isax,” in *ICDM*, 2017.
- [15] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas, “Coconut: A scalable bottom-up approach for building data series indexes,” *PVLDB*, vol. 11, no. 6, pp. 677–690, 2018.
- [16] M. Linardi and T. Palpanas, “Ulisse: Ultra compact index for variable-length similarity search in data series,” in *ICDE*, 2018.
- [17] —, “Scalable, variable-length similarity search in data series: The ulisse approach,” *PVLDB*, 2019.

- [18] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas, “Coconut palm: Static and streaming data series exploration now in your palm,” in *SIGMOD*, 2019.
- [19] D.-E. Yagoubi, R. Akbarinia, F. Massegli, and T. Palpanas, “Massively distributed time series indexing and querying,” *TKDE*, vol. 32, no. 1, 2019.
- [20] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim, “Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search,” *PVLDB*, 2019.
- [21] T. Palpanas, “Evolution of a Data Series Index,” *Communications in Computer and Information Science (CCIS)*, vol. 1197, pp. 61–75, 2020.
- [22] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos, “Progressive similarity search on time series data,” in *Proceedings of the Workshops of the EDBT/ICDT Joint Conference*, 2019.
- [23] L. Xiao, Y. Zheng, W. Tang, G. Yao, and L. Ruan, “Parallelizing dynamic time warping algorithm using prefix computations on gpu,” in *(HPCC\_EUC)*. IEEE, 2013, pp. 294–299.
- [24] A. Ailamaki, “Databases and hardware: The beginning and sequel of a beautiful friendship,” *VLDB*, 2015.
- [25] T. Palpanas, “The parallel and distributed future of data series mining,” in *HPCS*, 2017.
- [26] B. Peng, P. Fatourou, and T. Palpanas, “MESSI: In-Memory Data Series Indexing,” in *ICDE*, 2020.
- [27] Botao Peng (supervised by Panagiota Fatourou and Themis Palpanas), “Data Series Indexing Gone Parallel,” in *Proc. ICDE*, 2020.
- [28] B. Peng, P. Fatourou, and T. Palpanas, “Paris: The next destination for fast data series indexing and query answering,” in *IEEE BigData*, 2018.
- [29] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, “Dimensionality reduction for fast similarity search in large time series databases,” *KIS*, 2001.
- [30] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans, “Time series epenthesis: Clustering time series streams requires ignoring some data,” in *ICDM*, 2011, pp. 547–556.
- [31] J. Shieh and E. Keogh, “iSAX: disk-aware mining and indexing of massive time series datasets,” *DMKD*, no. 1, 2009.
- [32] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *CSUR*, 2009.
- [33] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, M. B. Westover, and N. B. Shamlo, “A disk-aware algorithm for time series motif discovery,” *DAMI*, vol. 22, no. 1-2, pp. 73–105, 2011.
- [34] R. Agrawal, C. Faloutsos, and A. N. Swami, “Efficient similarity search in sequence databases,” in *FODO*, 1993.
- [35] A. Mueen, S. Nath, and J. Liu, “Fast approximate correlation for massive time-series data,” in *SIGMOD*, 2010.
- [36] C. Lomont, “Introduction to intel advanced vector extensions,” *Intel White Paper*, pp. 1–21, 2011.
- [37] B. Tang, M. L. Yiu, Y. Li *et al.*, “Exploit every cycle: Vectorized time series algorithms on modern commodity cpus,” in *IMDM*, 2016.
- [38] I. Corporation, “Intel 64 and ia-32 architectures optimization reference manual,” <https://software.intel.com/>, 2016.
- [39] B.-K. Yi and C. Faloutsos, “Fast time sequence indexing for arbitrary lp norms,” in *VLDB*, 2000.
- [40] “Incorporated Research Institutions for Seismology – Seismic Data Access,” <http://ds.iris.edu/data/access/>, 2016.
- [41] “Southwest university adult lifespan dataset (sald),” [http://fcon\\_1000.projects.nitrc.org/indi/retro/sald.html](http://fcon_1000.projects.nitrc.org/indi/retro/sald.html), 2018.
- [42] “Source code and datasets used in this paper,” <http://www.mi.parisdescartes.fr/~themisp/paris/>, 2018.
- [43] K. Zoumpatianos, Y. Lou, I. Ileana, T. Palpanas, and J. Gehrke, “Generating data series query workloads,” *VLDBJ* 27(6), 2018.
- [44] S. Kashyap and P. Karras, “Scalable knn search on vertically stored time series,” in *SIGKDD*, 2011, pp. 1334–1342.
- [45] C. Li, P. S. Yu, and V. Castelli, “Hierarchyscan: A hierarchical similarity search algorithm for databases of long sequences,” in *ICDE*, 1996, pp. 546–553.
- [46] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *SIGMOD*, 1984, pp. 47–57.
- [47] I. Assent, R. Krieger, F. Afschari, and T. Seidl, “The ts-tree: efficient time series search and retrieval,” in *EDBT*, 2008.
- [48] J. Chou, K. Wu *et al.*, “Fastquery: A parallel indexing system for scientific data,” in *CLUSTER*. IEEE, 2011, pp. 455–464.
- [49] J. Zhou and K. A. Ross, “Implementing database operations using simd instructions,” in *SIGMOD*. ACM, 2002.
- [50] S. Zeuch, J. Freytag, and F. Huber, “Adapting tree structures for processing with SIMD instructions,” in *EDBT*, 2014.
- [51] P. Gepner and M. F. Kowalik, “Multi-core processors: New way to achieve high system performance,” in *PAR ELEC*, 2006.
- [52] D. B. Lomet and F. Nawab, “High performance temporal indexing on modern hardware,” in *ICDE*, 2015.
- [53] M. G. Gowanlock and H. Casanova, “Distance threshold similarity searches: Efficient trajectory indexing on the GPU,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, 2016.
- [54] A. Mueen, H. Hamooni, and T. Estrada, “Time series join on subsequence correlation,” in *ICDM*, 2014, pp. 450–459.
- [55] P. Ghodsnia, I. T. Bowman, and A. Nica, “Parallel i/o aware query optimization,” in *SIGMOD*. ACM, 2014.



**Botao Peng** received a BSc on optronics engineering from the Huazhong University of Science and Technology (China), in 2014, and a MSc on embedded systems from the University of Paris-Saclay (France), in 2016. He is currently pursuing a Ph.D. degree on database management at the Department of Computer Science, University of the Paris, under the supervision of Themis Palpanas.



**Panagiota Fatourou** is an Associate Professor of Computer Science at the University of Crete and at the Foundation of Research and Technology-Hellas (FORTH ICS), Greece. She has worked as a visiting Professor at EPFL, she is the chair of the ACM Europe Council and an ACM Distinguished Speaker, and has served as General Chair of ACM PODC 2013, and PC chair of OPODIS 2016 and SSS 2017.



**Themis Palpanas** is Senior Member of the French University Institute (IUF), and Professor of computer science at the University of Paris (France). He is the author of 9 US patents, 3 of which have been implemented in commercial products. He is the recipient of 3 Best Paper awards, and the IBM SUR Award. He is serving as Editor in Chief for BDR Journal, Associate Editor for PVLDB 2019 and TKDE journal, and Editorial Advisory Board member for IS journal.