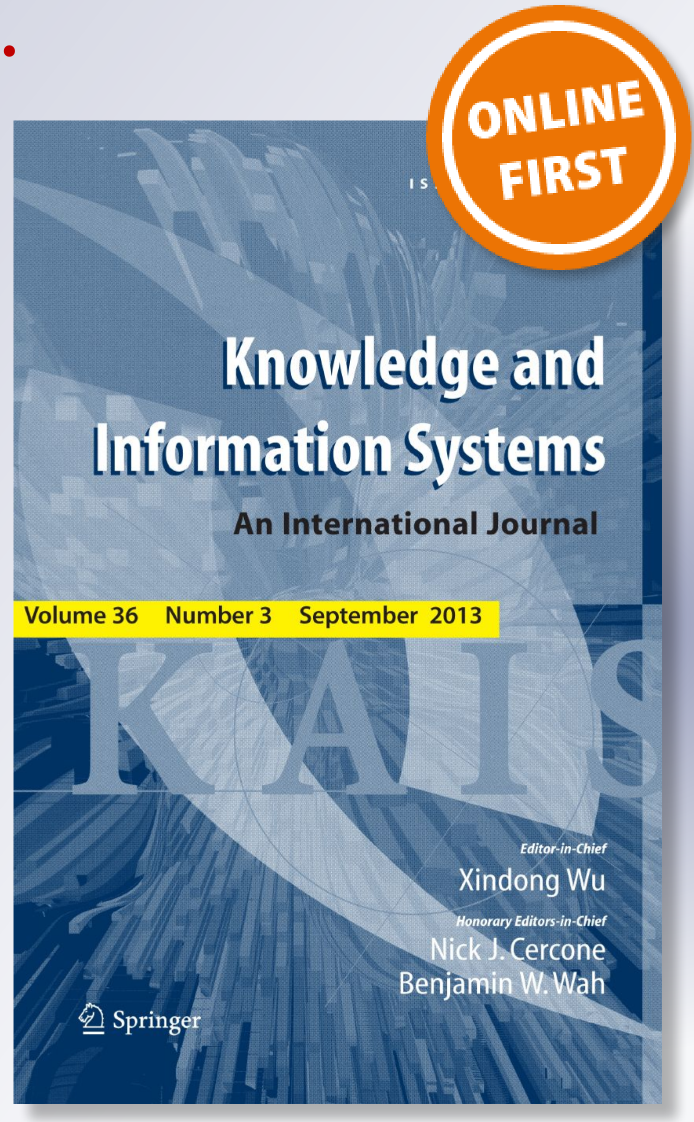
Exploiting a novel algorithm and GPUs to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins

Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, et al.

Knowledge and Information Systems
An International Journal

ISSN 0219-1377

Knowl Inf Syst
DOI 10.1007/s10115-017-1138-x



Knowl Inf Syst
https://doi.org/10.1007/s10115-017-1138-x

REGULAR PAPER

Exploiting a novel algorithm and GPUs to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins

Yan Zhu[1] · Zachary Zimmerman[1] · Nader Shakibay Senobari[2] · Chin-Chia Michael Yeh[1] · Gareth Funning[2] · Abdullah Mueen[3] · Philip Brisk[1] · Eamonn Keogh[1]

Received: 29 January 2017 / Revised: 28 July 2017 / Accepted: 14 November 2017

Abstract Time series motifs are approximately repeated subsequences found within a longer time series. They have been in the literature since 2002, but recently they have begun to receive significant attention in research and industrial communities. This is perhaps due to the growing realization that they implicitly offer solutions to a host of time series problems, including rule discovery, anomaly detection, density estimation, semantic segmentation, summarization, etc. Recent work has improved the scalability so exact motifs can be computed on datasets with up to a million data points in tenable time. However, in some domains, for example seismology or climatology, there is an immediate need to address even larger datasets. In

Yan Zhu and Zachary Zimmerman contributed equally, and should be considered joint first authors.

✉ Yan Zhu
yzhu015@ucr.edu

Zachary Zimmerman
zzimm001@ucr.edu

Nader Shakibay Senobari
nshak006@ucr.edu

Chin-Chia Michael Yeh
myeh003@ucr.edu

Gareth Funning
gareth@ucr.edu

Abdullah Mueen
mueen@cs.unm.edu

Philip Brisk
philip@cs.ucr.edu

Eamonn Keogh
eamonn@cs.ucr.edu

1 Department of Computer Science and Engineering, University of California, Riverside, USA

2 Department of Earth Sciences, University of California, Riverside, USA

3 Department of Computer Science, University of New Mexico, Albuquerque, USA

this work, we demonstrate that a combination of a novel algorithm and a high-performance GPU allows us to significantly improve the scalability of motif discovery. We demonstrate the scalability of our ideas by finding the full set of exact motifs on a dataset with one hundred and forty-three million subsequences, which is by far the largest dataset ever mined for time series motifs/joins; it requires ten quadrillion pairwise comparisons. Furthermore, we demonstrate that our algorithm can produce actionable insights into seismology and ethology.

Keywords Time series · Joins · Motifs · GPUs

1 Introduction

Time series motifs are approximately repeated subsequences found within a longer time series. While time series motifs have been in the literature for fifteen years [8], they recently have begun to receive significant attention *beyond* the data mining community. In recent years, they have been applied to a wide variety of problems, which include understanding the network of genes affecting the locomotion of the *C. elegans* nematode [5], severe weather prediction [16] and cataloging speech pathologies in humans [3].

Although significant progress has been made in how we score, rank, and visualize motifs, *discovering* them in large datasets remains a computational bottleneck. To date, we are unaware of any attempts to mine any dataset larger than one million data points [14]. In this work, we show how we can significantly improve the scalability of exact motif discovery both by leveraging GPU hardware and also by modifying the recently introduced STAMP algorithm [35]. The STAMP algorithm computes time series subsequence *joins* with an efficient *anytime* algorithm [35]. Our key observations follow below:

- The solution to the full exact 1NN *time series join* can be converted to the exact solution for any definition *of time series motif* [19] with only trivial extra effort. Moreover, full exact 1NN *time series join* also yields the exact solution for *time series discords*, a popular definition for *anomalies* in time series [7].
- The *anytime* property of STAMP may be useful to some users, but as we will explain below, in seismology, which is the domain motivating our work, it is not required or helpful. As we will demonstrate, if we forego this property, we can compute motifs at least an order of magnitude faster than STAMP. Moreover, forgoing the anytime property also makes it easier to leverage GPU hardware.

By maintaining the "STAMP" theme introduced in Yeh et al. [35], we call our faster algorithm STOMP, Scalable Time series Ordered-search Matrix Profile and its GPU-accelerated version GPU-STOMP.

In this work, we show that GPU-STOMP allows us to significantly expand the limits of scalability. We demonstrate the scalability of our ideas by extracting motifs from a dataset with one hundred and forty-three million subsequences objects. This requires computing (or *admissibly* pruning) 10,224,499,928,500,000 pairwise Euclidean distance values (i.e., more than ten quadrillion comparisons). If each Euclidean distance calculation took one microsecond, a brute-force algorithm would require 324 years.[1] As we will show, we can compute this join in only 9 days. We recognize the nine days seem like significant computational time, but it is important to note that these data represent 83 days of continuous seismology recording at 20 Hz. Therefore, even at this massive scale, our algorithm is much faster than real time.

[1] 10,224,499,928,500,000 * 0.000001 s is 324 years.

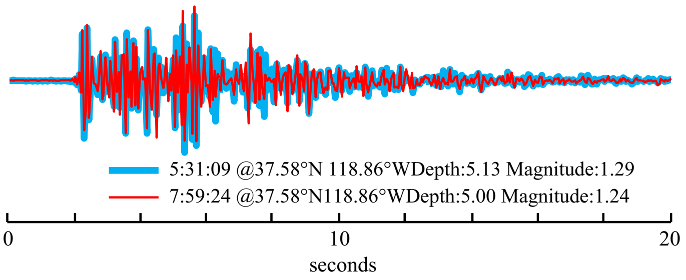


Fig. 1 A pair of repeating earthquake sequences (motifs) we discovered from seismic data recorded at a station near Mammoth Lakes on February 17, 2016. One occurrence (fine/red) is overlaid on top of another occurrence (bold/blue) that happened hours earlier. (best viewed in color) (color figure online)

Figure 1 previews a pair of repeating earthquake sequences, which is essentially a *time series motif* [8], discovered by our algorithm in a seismologic dataset.

Here, the two occurrences are very similar despite happening 148 minutes apart. Although the geophysics of earthquakes indicates that in principle we could see similar events millennia apart, we are unfortunately limited to the few decades humans have been recording such data (see Fig. 12).

The rest of this paper is organized as follows. In Sect. 2, we introduce background and related work in data mining and seismology. In Sect. 3, we introduce the necessary definitions and notations, which allow us to introduce our algorithms in Sect. 4. We conduct an extensive empirical comparison in Sect. 5, including comparisons to the most obvious rival methods. Finally, in Sect. 6, we offer conclusions and directions for related work.

2 Background and related work

2.1 Motif discovery background

Motif discovery for time series was introduced in 2003 [8] (although the classic paper of Agrawal, Faloutsos and Swami foreshadows motifs by computing all-pair similarity for time series [1]). Since then, it increased in research activity. One critical direction has been applying motifs to solve problems in a wide variety of domains such as bioinformatics [5], speech processing [3], robotics, human activity understanding [31,32], severe weather prediction [16], neurology and entomology [19]. The other key research focus has been in the extensions and generalizations of the original work, especially in the attempts to improve scalability [14,19]. These attempts to improve the scalability of motif discovery fall into two broad classes; *approximate* and *exact* motif discovery [14,18,19].

Clearly approximate motifs *can* be much faster to compute, and this may be useful in some domains. However, there are domains in which the risk of false negatives is unacceptable. Consider seismology, which is the domain motivating our work [36]. This is a domain in which false negatives could affect public policy, change insurance rates for customers and conceivably cost lives by allowing a dangerous site to be developed for dwellings. Given that the task-at-hand is to find *exact* motifs, all known methods based on *hashing* [36] and/or *data discretization* [8,18] can be dismissed from consideration.

Beyond being exact, the proposed approach has many advantages that are not shared by rival methods.

- The proposed method is simple and parameter-free: In contrast, other methods typically require building and tuning spatial access methods and/or hash functions [8,14,15,18, 31,32,36].
- It is space efficient: Our algorithm requires an inconsequential space overhead, just linear in the time series length, with a small constant factor. In particular, we avoid the need to actually explicitly extract the individual subsequences [8,18,19], something that would increase the space complexity by two or three orders of magnitude.
- It is incrementally maintainable: Having computed motifs for a dataset, we can incrementally update the best motifs very efficiently if new data arrive [35].
- It can leverage hardware: As we show below, our algorithm is embarrassingly parallelizable on multicore processors.
- Our algorithm has time complexity that is *constant* in subsequence length: This is a very unusual and desirable property; virtually all time series algorithms scale poorly as the subsequence length grows (the classic *curse of dimensionality*) [8,14,15,18,31,36].
- Our algorithm takes deterministic time, dependent on the data's length, but completely independent of the data's structure/ noise level, etc. This is also unusual and desirable property for an algorithm in this domain. For example, even for a fixed time series length, and a fixed subsequence length, all other algorithms we are aware of can radically different times to finish on two (even slightly) different datasets [8,14,15,18,31,36]. In contrast, given *only* the length of the time series, we can predict precisely how long it will take our algorithm to finish in advance.

Virtually every time series data mining technique has been applied to the motif discovery problem, including indexing [14,33], data discretization [8], triangular-inequality pruning [19], hashing [31,32,36], early abandoning, etc. However, all these techniques rely on the assumption that the *intrinsic* dimensionality of time series is much lower than the *recorded* dimensionality [8,31,32,34,36]. This is generally true for data such as short snippets of heartbeats and gestures, etc.;however, it is not true for seismographic data, which is intrinsically high dimensional. To ascertain this, we performed a simple experiment.

We measured the *tightness of lower bounds* (TLB) for three types of data, using the two most commonly used dimensionality reduction representations for time series, DFT and PAA. Additionally, PAA is essentially equivalent to the Haar wavelets for this purpose [34]. The TLB is defined as:

$$TLB = LowerBoundDist(\mathrm{A,B})/TrueEuclideanDist(\mathrm{A,B})$$

It is well understood that the TLB is near perfectly (inversely) correlated with wall-clock time, CPU operations, number of disk access or any other performance metric for similarity search, all-pair-joins, motifs discovery, etc. [34]. As the mean TLB decreases, we quickly degrade to simple brute-force search. The absolute minimum value of TLB is dependent on the data, the search algorithm and the problem setting (main memory based vs disk based). However as Wang et al. [34] demonstrates, lower bound values less than 0.5 generally do not "break even."

Figure 2 shows unambiguous results. There is *some* hope that we could avail current speedup techniques when considering (relatively smooth and simple) short snippets of ECGs, but there is *little* hope that the noisy and more complex human activity would yield to such optimizations, and there is *no* hope that anything currently in the literature will help with seismological data. This claim is further proven in our detailed experiments in Sect. 5.

Even if we ignore this apparent death-knell for indexing/spatial access techniques, we could still dismiss them for other reasons, including memory considerations. As demonstrated

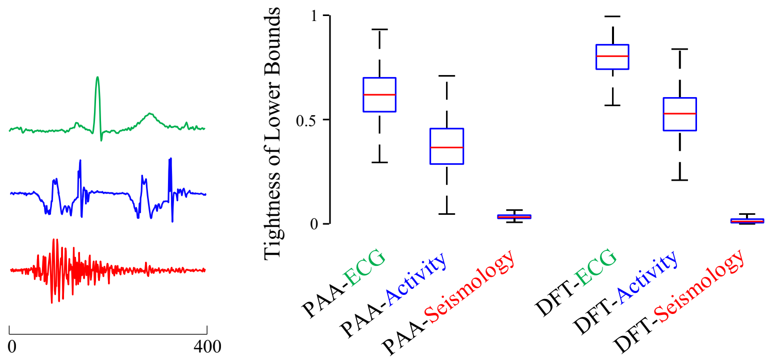


Fig. 2 Left Samples from three datasets, ECG, human activity and seismology (available in [22]). Right The tightness of lower bounds, averaged over 10,000 random pairs, using PAA and DFT

in Sect. 4, a critical property of our algorithm is that it does not need to explicitly *extract* the subsequences, which is unlike the indexing/spatial access methods. For example, consider a time series of length 100 million, with eight bytes per value, requiring 0.8 GB. Our algorithm requires an overhead of seven other vectors of the same size (including the output), for an easily manageable total of 6.4 GB (if memory *was* a bottleneck, we could reduce this by using reduced precision vectors or compression). However, any indexing algorithm that needs to extract the subsequences will increase memory requirements by *at least* O(d), where d is the reduced dimensionality used in the index [15]. Given that d may be 20 or greater, this indicates the memory requirements grow to at least 16 GB. With such a large memory footprint, we are almost certainly condemned to a random-access disk-based algorithm, dashing any hope of any speedup.

A related advantage of our framework is that we can choose the subsequence length *just* prior to performing the motif discovery. In contrast, any index-based technique must commit to a subsequence length before *building* the index, and it could take hours/days to build the data structure before any actual searching could begin [24,34]. If such an index is built to support subsequences of say length 200, it cannot be used to join subsequences of length 190 or 205, etc. (see Section 1.2.3 of Rakthanmanon et al. [24]). Thus, if we change our mind about the length of patterns we are interested in, we are condemned to a costly rebuilding of the entire index. It is difficult to overstate the utility of this feature. In Sect. 5.8, we will demonstrate how we can use STOMP to explore the behavior of a penguin. At the beginning of this case study, we had no idea of what time frame the penguin's behavior might be manifest. However, with no costly index to build, we simply tried a few possible lengths until it was obvious that we found a reasonable value.

In summary, while we obviously are unable to absolutely guarantee that there is no other scalable solution to our task-at-hand, we are confident that there is no existing off-the-shelf technology that can be used or adapted to allow us to get within two orders of magnitude of the results we obtain on the largest datasets.

2.2 Seismological background

While our algorithms are completely general and can be applied to any domain, seismological data are of particular interest to us, due to its sheer scale and importance in human affairs.

In the early 1980s, it was discovered that in the telemetry of seismic data recorded by the same instrument from sources in given region, there will be many similar seismograms [9]. Geller and Mueller [9] have suggested that, "*The physical basis of this clustering is that the earthquakes represent repeated stress release at the same asperity, or stress concentration, along the fault surface*." These patterns are called "repeating earthquake sequences" in seismology and correspond to the more general term "*time series motifs*." Figure 1 shows an example of a repeating earthquake sequence pair from seismic data.

A more recent paper notes that many fundamental problems in seismology can be solved by joining seismometer telemetry in locating these repeating earthquake sequences [36], which includes the discovery of foreshocks, aftershocks, triggered earthquakes, swarms, volcanic activity and induced seismicity. However, the paper further notes that an *exact* join with a query length of 200 on a data stream of length 604,781 requires 9.5 days. Their solution, a transformation of the data to allow LSH-based techniques, does achieve significant speedup, but at the cost of false negatives and necessary careful parameter tuning. For example, Yeh et al. [35] notes that they need to set the threshold to precisely 0.818 to achieve decent results. While we defer a full discussion of experimental results to Sect. 5, the ideas introduced in this paper can reduce the quoted 9.5 days for exact motif discovery from a dataset of size 604,781 to less than one minute, without tuning any parameters and also guaranteeing that false negatives will not occur.

It is vital to note that this kind of speedup really is game changing in this domain. It allows seismologists to quickly identify or detect earthquakes that are identical or similar in location without needing trilateration, and it can also provide useful information on relative timing and location of such events [2,12,13].

More controversially, some researchers have suggested that the slow slip on the fault accompanying non-volcanic tremors (a sequence of low-frequency earthquakes, many of which are repeated) may temporarily increase the probability of triggering a large earthquake. Therefore, detecting and locating these repeating LFEs allows more accurate short-term earthquake forecasting [12].

Finally, we note that seismologists have been early adopters of GPU technology [17] and other high-performance computing paradigms. However, their use of this technology has been limited to similarity search, not motif search.

2.3 Developing intuitions for the matrix profile

Unlike other motif/anomaly discovery systems, the matrix profile computes a score for *every* subsequence in the dataset. Here, we take the time to give some examples to demonstrate the utility of this more comprehensive annotation of data. We begin by considering the New York Taxi dataset of Rong and Bailis [25]. As shown in Fig. 3.*top*, the data are the normalized number of NYC taxi passengers over 10 weeks, October 1 to December 15 2014. The authors show this dataset to demonstrate the versatility of their "Attention Prioritization" technique for finding unusual patterns [4,25]. In essence, they transform the data (not shown here) in a way to make the discovery of anomalies easier. They note that Thanksgiving, on Thursday, November 27, can be considered an "anomaly" in this dataset, since the patterns of travel apparently change during this important US holiday.

We computed the matrix profile for this dataset, with a subsequence length of one and a half days. As Fig. 3.*middle* shows, the matrix profile peaks at the location that indicates Thanksgiving. However, there are additional observations that we can make with the matrix profile. There is a secondary anomaly occurring on Sunday, November 2; there appears to be a spike in taxi demand at about 2:00 am. With a little thought, we realize this is exactly the

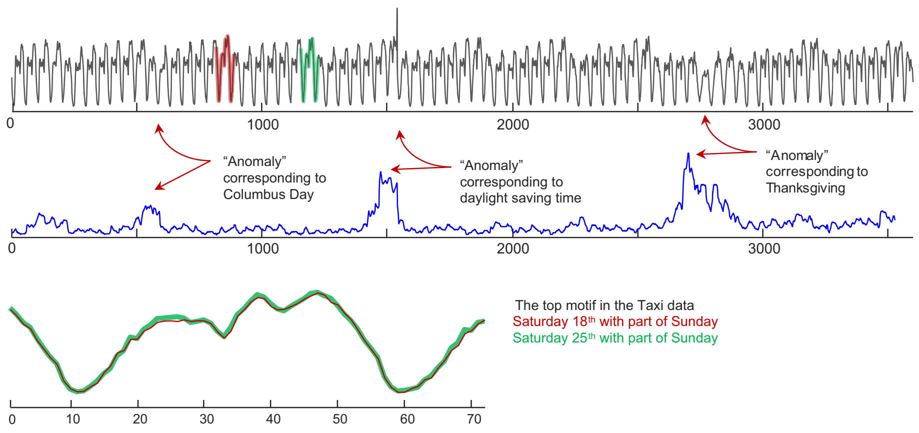


Fig. 3 Top Normalized number of NYC taxi passengers over 10 weeks [4,25]. Middle The matrix profile produces high values where the corresponding subsequences are unusual. Bottom The top motif corresponds to two consecutive Saturdays

hour in which daylight saving time is observed in the USA. Setting the clock back one hour gives the appearance of doubling the normal demand for taxis at that hour. There is arguably a third anomaly in the dataset, with a more subtle, but still significant peak at October 13. This day corresponds to Columbus Day. This holiday is all but ignored in most of the USA, but it is still observed in New York, which has a strong and patriotic Italian community.

In Fig. 3 *bottom,* we show the top-1 *motif* from the dataset, which is extremely well conserved. In many natural datasets, for example the circadian rhythm of an animal, the best motifs are typically exactly twenty-four hours apart (a phenomenon known as *persistence*). However, because this motif's two occurrences are exactly *seven* days apart, the importance of artificial divisions of the calendar on human behaviors becomes apparent. It is possible that the regions of lower conservation with the motif are also telling. For example, from 24 to 26 (about 10–11 am), the motif corresponding to the 25th (green/bold) is a little higher than the previous week. It was lightly raining (about 0.12 inches) at the time, which may explain the slightly higher taxi demand in the late morning.

3 Notation and definitions

While we mostly follow the framework introduced in Yeh et al. [35], for completeness we review all necessary definitions.

3.1 Definitions

We begin by defining the data type of interest, *time series*:

Definition 1 A *time series* T is a sequence of real-valued numbers t_i:$T = t_1, t_2, \dots, t_n$, where n is the length of T.

We are interested in *local*, not *global* properties of time series. A local region of time series is called a *subsequence*.

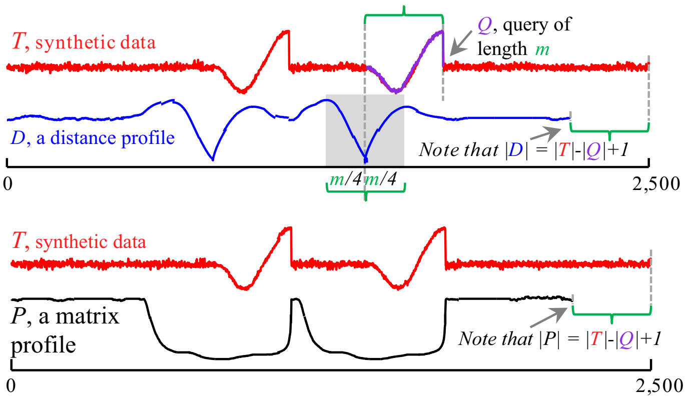


Fig. 4 Top One distance profile (Definition 3) created from a random query subsequence Q of T. If we created distance profiles for all possible query subsequences of T, the element-wise minimum of this set would be the matrix profile (Definition 4) shown at bottom. Note that the two lowest values in P are at the location of the first motif [8,19]

Definition 2 A *subsequence* $T_{i,m}$ of a time series T is a continuous subset of the values from T of length m, which begin at position i. Formally, $T_{i,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$, where $1 \leq i \leq n - m + 1$.

We can take a subsequence and compute its distance to *all* subsequences in the same time series. This is called a *distance profile*.

Definition 3 A *distance profile* D_i of time series T is a vector of the Euclidean distances between a given query subsequence $T_{i,m}$ and each subsequence in time series T. Formally, $D_i = [d_{i,1}, d_{i,2}, \ldots, d_{i,n-m+1}]$, where $d_{i,j}$ $(1 \leq i, j \leq n - m + 1)$ is the distance between $T_{i,m}$ and $T_{j,m}$.

We assume that the distance is measured by Euclidean distance between z-normalized subsequences [34].

We are interested in finding the nearest neighbors of all subsequences in T, as the closest pairs of this are the classic definition of time series motifs [8,19]. Note that by definition, the ith location of distance profile D_i is zero, and it is close to zero just before and after this location. Such matches are defined as *trivial matches* [19]. We avoid such matches by ignoring an "exclusion zone" of length $m/4$ before and after the location of the query. In practice, we simply set $d_{i,j}$ to infinity $(i - m/4 \leq j \leq i + m/4)$ while evaluating D_i.

We use a vector called *matrix profile* to represent the distances between all subsequences and their nearest neighbors.

Definition 4 A *matrix profile* P of time series T is a vector of the Euclidean distances between each subsequence $T_{i,m}$ and its nearest neighbor (i.e., the closest match) in time series T. Formally, $P = [\min(D_1), \min(D_2), \ldots, \min(D_{n-m+1})]$, where D_i $(1 \leq i \leq n - m + 1)$ is the distance profile D_i of time series T.

We call this vector a matrix profile, since it could be computed by using the full distance matrix of all pairs of subsequences in time series T, and evaluating the minimum value of each column (although this method is naïve and space-inefficient). Figure 4 illustrates both a *distance profile* and a *matrix profile* created on the same raw time series T.

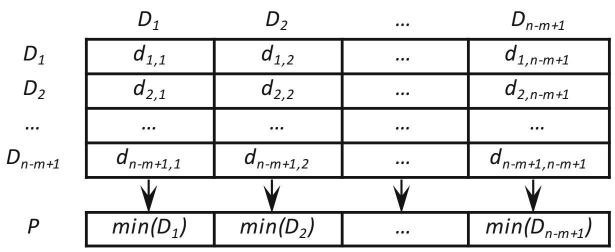


Fig. 5 An illustration of the relationship between the distance profile, the matrix profile and the full distance matrix. For clarity, note that we do *not* actually create the full distance matrix, as this would have untenable memory requirements

It is important to note that the full distance matrix is symmetric: D_i is both the ith row and the ith column of the full distance matrix. Figure 5 shows this more concretely.

The ith element in the matrix profile P indicates the Euclidean distance from subsequence $T_{i,m}$ to its nearest neighbor in time series T. However, it does not indicate the location of that nearest neighbor. This information is recorded in a companion data structure called the *matrix profile index*.

Definition 5 A *matrix profile index* I of time series T is a vector of integers: $I = [I_1, I_2, \ldots I_{n-m+1}]$, where $I_i = j$ if $d_{i,j} = \min(D_i)$.

By storing the neighboring information in this manner, we can efficiently retrieve the nearest neighbor of query $T_{i,m}$ by accessing the ith element in the matrix profile index.

To briefly summarize this section, we can create two meta time series, the *matrix profile* and the *matrix profile index*, to annotate a time series T with the distance and location of all its subsequences' nearest neighbors within T. As the reader may already have realized, the smallest pair of values in the *matrix profile* correspond to the best motif pair by the classical definition [8,14,19], and the corresponding values in the *matrix profile index* indicate the location of the motif. Moreover, as both Mueen et al. [19] and Yeh et al. [35] argue, the *top-k* motifs, *range* motifs, and any other reasonable variant of motifs can trivially be computed from the information in the *matrix profile*, which is the focus of the remainder of this paper.

3.2 A brief review of the STAMP algorithm

The recently introduced STAMP algorithm can efficiently compute the *full* and *exact* matrix profile and matrix profile index of a time series [35]. The STAMP algorithm essentially evaluates the distance profile D_i of a query subsequence $T_{i,m}$ by utilizing the FFT(fast Fourier transform) to calculate the dot product between $T_{i,\mathrm{m}}$ and all of the subsequences of the time series T. The overall time complexity of the algorithm is $O(n^2 \log n)$, and the space complexity is $O(n)$, where n is the length of time series T. The STAMP algorithm can process a time series with up to a million data points in tenable time. However, to solve the problems in our motivating domain seismology, it is necessary to process even larger datasets. It would take STAMP more than 20 years to analyze a seismology time series sampled at 20 Hz for about 2 months, which is of length 100 million (see Table 4). In the next section, we will show a new and fast algorithm, which can finish processing the same time series in only 4 days when it is built on top of a GPU.

4 Algorithms

In this section, we begin by demonstrating that we can improve upon the STAMP algorithm [35] to create the much faster STOMP algorithm. Then we demonstrate that the structure of STOMP lends itself to porting to GPUs.

4.1 The STOMP algorithm

As explained below, STOMP is similar to STAMP [35] in that it can be viewed as highly optimized nested loop searches with repeating calculations of distance profiles in the inner loop. However, while STAMP must evaluate the distance profiles in a random order (to allow its anytime behavior), STOMP performs an *ordered* search. By exploiting the locality of these searches, we can reduce the time complexity by a factor of O(log n).

Before we explain the details of the algorithm, we first introduce a formula to calculate the z-normalized Euclidean distance $d_{i,j}$ of two time series subsequences $T_{i,m}$ and $T_{j,m}$ by using their dot product, $QT_{i,j}$:

$$d_{i,j} = \sqrt{2m\left(1 - \frac{QT_{i,j} - m\mu_i\mu_j}{m\sigma_i\sigma_j}\right)} \tag{1}$$

Here m is the subsequence length, μ_i is the mean of $T_{i,m}$, μ_j is the mean of $T_{j,m}$, σ_i is the standard deviation of $T_{i,m}$, and σ_j is the standard deviation of $T_{j,m}$.

The technique introduced in Rakthanmanon et al. [24] allows us to obtain the means and standard deviations with O(1) time complexity; thus, the time required to compute $d_{i,j}$ depends only on the time required to compute $QT_{i,j}$. Here, we claim that $QT_{i,j}$ can also be computed in O(1) time, once $QT_{i-1,j-1}$ is known.

Note that $QT_{i-1,j-1}$ can be decomposed as the following:

$$QT_{i-1,j-1} = \sum_{k=0}^{m-1} t_{i-1+k}t_{j-1+k} \tag{2}$$

and $QT_{i,j}$ can be decomposed as the following:

$$QT_{i,j} = \sum_{k=0}^{m-1} t_{i+k}t_{j+k} \tag{3}$$

Thus we have:

$$QT_{i,j} = QT_{i-1,j-1} - t_{i-1}t_{j-1} + t_{i+m-1}t_{j+m-1} \tag{4}$$

Therefore, our claim is proved.

Figure 6 visualizes the algorithm. Based on (1), we can map the distance matrix in Fig. 5 (also shown in Fig. 6.left) to its corresponding dot product matrix (shown in Fig. 6.right).

The arrows in Fig. 6.right show the data dependency indicated by (4): Once we have $QT_{i-1,j-1}$, we can compute $QT_{i,j}$ in O(1) time. However, note that (4) does not apply to the special case when $i = 1$ or $j = 1$ (the first row and the first column of Fig. 6.right, shown in red). This problem is easy to solve: We can pre-compute the dot product values in these two special cases with FFT, as shown in Table 1. Concretely, we use *SlidingDotProduct*($T_{1,m}$, T) to calculate the first dot product vector: $QT_1 = [QT_{1,1}, QT_{1,2}, \ldots, QT_{1,n-m+1}] = [QT_{1,1}, QT_{2,1}, \ldots, QT_{n-m+1,1}]$. The dot product vector is stored in memory and used as needed.

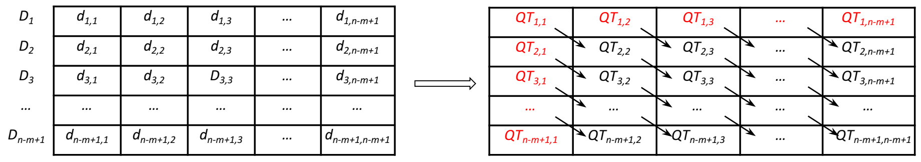

D_1 $d_{1,1}$ $d_{1,2}$ $d_{1,3}$ ... $d_{1,n-m+1}$
D_2 $d_{2,1}$ $d_{2,2}$ $d_{2,3}$ ... $d_{2,n-m+1}$
D_3 $d_{3,1}$ $d_{3,2}$ $D_{3,3}$ ... $d_{3,n-m+1}$
D_{n-m+1} $d_{n-m+1,1}$ $d_{n-m+1,2}$ $d_{n-m+1,3}$ ... $d_{n-m+1,n-m+1}$
$QT_{1,1}$ $QT_{1,2}$ $QT_{1,3}$ ... $QT_{1,n-m+1}$
$QT_{2,1}$ $QT_{2,2}$ $QT_{2,3}$ ... $QT_{2,n-m+1}$
$QT_{3,1}$ $QT_{3,2}$ $QT_{3,3}$ ... $QT_{3,n-m+1}$
$QT_{n-m+1,1}$ $QT_{n-m+1,2}$ $QT_{n-m+1,3}$ ... $QT_{n-m+1,n-m+1}$

Fig. 6 Mapping the computation of the distance matrix (left) to the computation of its corresponding dot product matrix (right) (color figure online)

Table 1 Calculate sliding dot product with FFT

```
Procedure SlidingDotProduct(Q, T)
Input: A query Q, and a user provided time series T
Output: The dot product between Q and all subsequences in T
1   n ← Length(T), m ← Length(Q)
2   T_a ← Append T with n zeros
3   Q_r ← Reverse(Q)
4   Q_ra ← Append Q_r with 2n-m zeros
5   Q_raf ← FFT(Q_ra), T_af ← FFT(T_a)
6   QT ← InverseFFT(ElementwiseMultiplication(Q_raf, T_af))
7   return QT[m:n]
```

Table 2 STOMP algorithm

```
Procedure STOMP(T, m)
Input: A time series T and a subsequence length m
Output:  Matrix profile P and the associated matrix profile index I of T
1   n ← Length(T), l ← n-m+1
2   μ, σ ← ComputeMeanStd(T, m) // see [24]
3   QT ← SlidingDotProduct(T[1:m], T), QT_first ← QT
4   D ← CalculateDistanceProfile(QT, μ, σ, 1) // see (1)
5   P← D, I← ones                  // initialization
6   for i = 2 to l                 // in-order evaluation
7      for j= l downto 2           // update dot product, see (4)
8         QT[j] ← QT[j-1]-T[j-1]×T[i-1]+T[j+m-1]×T[i+m-1]
9      end for
10     QT[1] ← QT_first[i]
11     D ← CalculateDistanceProfile(QT, μ, σ, i)  // see (1)
12     P, I ← ElementWiseMin(P, I, D, i)
13  end for
14  return P, I
```

After the first row and the first column in Fig. 6.right are computed, the rest of the dot product matrix is evaluated row after row.

We are now in the position to introduce our STOMP algorithm in Table 2.

The algorithm begins in line 1 by computing the matrix profile length l. In line 2, the mean and standard deviation of every subsequence in T are pre-calculated. Line 3 calculates the first dot product vector QT with the algorithm in Table 1. In line 5, we initialize the matrix profile P and matrix profile index I. The loop in lines 6–13 calculates the distance profile of every subsequence of T in sequential order. Lines 7–9 update *QT* according to (4). We update *QT*[1] in line 10 with the pre-computed *QT_first* in line 3. Line 11 calculates distance profile D according to (1). Finally, line 12 compares every element of P with D: if $D[j] < P[j]$, then $P[j] = D[j]$, $I[j] = i$.

The time complexity of STOMP is O(n^2); thus, we have achieved a O(log n) factor speedup over STAMP [35]. Moreover, it is clear that O(n^2) is optimal for any exact motif algorithm in the general case. The O(log$n$) speedup makes little difference for small datasets and for those with just a few tens of thousands of data points [8]. However, as we consider the datasets with millions of data points, this O(log n) factor begins to produce a very useful order-of-magnitude speedup.

To better understand the efficiency of STOMP, it is important to clarify that the time complexity of the classic brute-force algorithm is O(n^2m). The value of m depends on the domain, but in Sect. 5.8, we consider real-world applications where it is 2000. Most techniques in the literature gain speedup by slightly reducing the n^2 factor; however, we gain speedup by reducing the m factor to O(1). Moreover, it is important to remember that the techniques in the literature can only reduce this n^2 factor if the data have a low intrinsic dimensionality (recall Fig. 2), *and* the domain requires a short subsequence length. In contrast, the speedup for STOMP is completely independent of *both* the structure of the data and the subsequence length.

Despite this dramatic improvement, it still takes STOMP approximately 5–6 h to process a time series of length one million. Can we further reduce the time?

It is important to note that the STOMP algorithm is extremely amenable to parallel computing frameworks. This is not a coincidence; the algorithm was conceived with regard to eventual hardware acceleration. Recall that the space requirement for the algorithm is only O(n); there is no data dependency in the main inner loop of the algorithm (lines 7–9 of Table 2), so we can update all entries of *QT* in parallel. The evaluation of each entry in vectors D, P and I in lines 11 and 12 is also independent of each other. In the next section, we will introduce a GPU-based version of STOMP, utilizing these observations to further speed up the evaluation of the matrix profile and thus motif discovery.

4.2 Porting STOMP to a GPU framework

The Graphic Processor Unit, or GPU, is "*especially well-suited to address problems that can be expressed as data-parallel computations*" [20]. It has its own memory, and it can launch multiple threads in parallel. Here, we use the ubiquitous Single Instruction Multiple Data (SIMD) NVIDIA CUDA architecture, where we can assign multiple threads to process the same set of instructions on multiple data.

The threads on the GPU are managed in thread blocks. Threads in a thread block run simultaneously, and they can cooperate with each other through shared local resources. A CUDA function is called a *kernel*. When we launch a kernel, we can specify the number of blocks and the number of threads in each block to run on GPU. For example, the NVIDIA Tesla K80 allows launching at most 1024 threads within a block and as many as 2^{63} blocks (a total of 2^{73} threads), which is plentiful for processing a time series of length 100 million.

The GPU implementation of the STOMP algorithm in Table 2 can be decomposed into four steps:

- CPU copies the time series to GPU global memory.
- CPU launches GPU kernels to evaluate μ, σ, the initial *QT*, D, P and I.

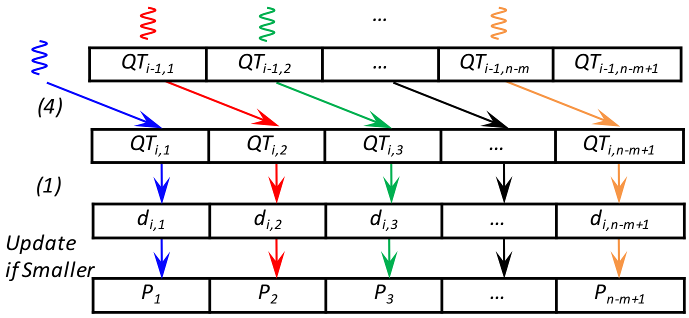


Fig. 7 Division of work among threads in the third step of GPU-STOMP

- CPU iteratively launches GPU kernels to update QT, D, P and I.
- CPU copies the final output (P and I) from GPU.

In the first step, the CPU copies time series T (input vector of Table 2) to the global memory of GPU. The time used to copy a time series of length 100 million takes less than a second. Note that in order to run the STOMP algorithm, we need to allocate space to store eight vectors in the GPU global memory: T, μ, σ, QT, QT_first, D, P and I. A double-precision time series of length 100 million is approximately 0.8 GB, so the algorithm utilizes approximately 6.4 GB global memory space. This is feasible for NVIDIA Tesla K40 and K80 cards; however, if the device used has less memory space available, we can split the time series into small sections and evaluate one section at a time with the GPU.

In the second step, the CPU launches GPU kernels to evaluate the vectors in parallel. The mean and standard deviation vectors in line 2 of Table 2 can be efficiently evaluated by CUDA Thrust [20]. The first QT vector in line 3 can be evaluated in parallel by applying CUFFT, the NVIDIA CUDA fast Fourier transform [21] to the *SlidingDotProduct* function in Table 1. We assign a total of $n - m+1$ threads to evaluate QT_first, D, P and I in lines 3–5 in parallel. The jth thread processes the jth entry of these vectors one by one.

Now that we have initialized QT, D, P, and I, we update them iteratively. In the third step, the CPU runs the outer loop in lines 6–13 of Table 2 iteratively. In every iteration, the CPU launches a GPU kernel with $n - m + 1$ threads, parallelizing the evaluation of QT, D, P, and I. As shown in Fig. 7, the first thread reads $QT[1]$ from the pre-computed QT_first vector, while the second to the last threads evaluate their corresponding entry of QT using (4).

Note that in contrast to the CPU STOMP algorithm, which uses only one vector QT to store both QT_{i-1} and QT_i, here we use two vectors to separate them. This is necessary because as the threads evaluate entries in QT in parallel, we need to avoid writing entries *before* they are read. A simple and efficient way to accomplish this is to create two vectors, QT_odd and QT_even. When the outer loop variable i in line 6 is even, the threads read from QT_odd and write to QT_even; when i is odd, the threads read data from QT_even and write to QT_odd. Following this, the threads evaluate D with (1), and the jth thread updates P and I if $D[j] < P[j]$.

When all of the iterations are complete, we have reached the last step of GPU-STOMP, where the CPU copies P and I back to the system memory.

4.3 Further parallelizing STOMP with multiple GPUs

The parallelization scheme above is suitable if we only have one GPU device. Can we further reduce the processing time if there are two or more GPUs available?

Thus far, we have been using CPU to iteratively control the outer loop of the STOMP algorithm in Table 2. We start by computing the first distance profile (the first row) in Fig. 5 and its corresponding QT vector. Then in each iteration, we compute a new row of the distance matrix in Fig. 5 and maintain the minimum-so-far values of each column in vector P. When the iteration is complete, P becomes the exact matrix profile.

This outer loop computation can be further parallelized. Assume we have k independent GPU devices, and we also have $(n-m+1)/k = q$. We can then divide the distance matrix in Fig. 5 into k sections: device 1 evaluates the first to the qth rows, device 2 evaluates the $(q+1)$th to the $(2q)$th rows, etc. Essentially, device k uses the parallelized version of *SlidingDotProduct* function in Table 1 to calculate $QT_{q(k-1)+1}$ and $D_{q(k-1)+1}$, and then it evaluates the following $q-1$ rows iteratively. The k devices can run in parallel, and after the evaluation completes, we can simply find the minimum among all the k matrix profile outputs. In summary, we can achieve a k-times speedup by using k identical GPU devices.

By porting all the introduced techniques to NVIDIA Tesla K80, which contains two GPU devices on the same unit, we are able to obtain the matrix profile and matrix profile index of a seismology time series of length 100 million within 19 days. Are there any further optimizations left?

4.4 A technique to further accelerate GPU-STOMP

Figure 7 shows the process to compute the ith row of the distance matrix in Fig. 5 by $n-m+1$ parallel threads. Recall that the distance matrix is symmetric; half of the distance computations can be saved if we only evaluate the ith to the last columns. We show this strategy in Fig. 8.*top*.

However, note that it is desirable to maintain the $O(n)$ space complexity of our algorithm; if we move on to the $(i+1)$th row Fig. 5 without further processing, then $P_i = \min(d_{1,i}, d_{2,i} \ldots, d_{i,i})$, and it would no longer be updated. To correct this, it is necessary to launch another kernel after Fig. 8.*top* is completed. The new kernel is shown in Fig. 8.*bottom*.

Essentially, we have used an analogous reduction technique as in Harris [10] to obtain $d_{\min} = \min(d_{i,i+1}, d_{i,i+2}, \ldots, d_{i,m+n-1})$, which also is equivalent to $\min(d_{i+1,i}, d_{i+2,i}, \ldots, d_{n-m+1,i})$ as a result of symmetry. If $d_{\min} < P_i$, we set $P_i = d_{\min}$, so $P_i = \min(D_i)$. Although it is necessary to launch an additional kernel to process each row, which will require extra time, the extra time is still less than what is saved when handling large time series.

For example, this new technique reduced the time to process a time series of length 100 million from 19 days to approximately 12 days on NVIDIA Tesla K80. This indicates that it is possible to finish five quadrillion pairwise comparison of subsequences within 12 days.

Note that fewer and fewer threads are being launched in each iteration. To apply this new technique to multiple GPUs, it is necessary to ensure that each GPU is loaded with similar amount of work, so they will finish in similar time. Here, for NVIDIA Tesla K80, we computed the first $(n-m+1)(1-1/\sqrt{2})$ distance profiles with the first GPU and the last $(n-m+1)/\sqrt{2}$ distance profiles with the second GPU.

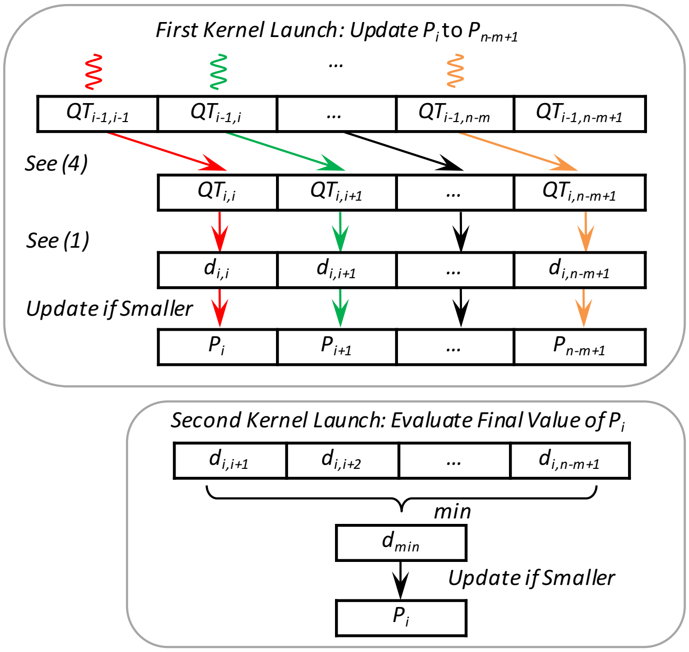


Fig. 8 Modifying the third step of GPU-STOMP. Top Launch only $n - m - i + 2$ threads (instead of the $n - m + 1$ threads in Fig. 7) this time at the ith iteration. Bottom Launch another kernel to evaluate the final value of P_i

4.5 A final optimization: breaking the ten quadrillion pairwise comparison barrier

In the last section, we demonstrated a technique to use parallel threads to evaluate the rows of the distance matrix in Fig. 5 iteratively. Note that to compute one row, the technique needs to launch two kernels, all threads need to be synchronized following the evaluation, and the corresponding QT vector needs to be updated in GPU global memory. As there are $n - m + 1$ rows in Fig. 5, when n becomes large, the time cost for kernel launch, and the thread synchronization and memory writing become nontrivial.

As impressive as the results are in the last section, which breaks the 5 quadrillion pairwise comparison barrier, there is one more optimization we can perform to further speed up the GPU code. We denote this optimized version GPU-STOMP$_{\text{OPT}}$. To help the reader better understand how the GPU-STOMP$_{\text{OPT}}$ works, we will first show our initial optimization scheme in Fig. 9 and then further refine it in Figs. 10 and 11.

Figure 9 shows our key scheme to save the kernel launch and thread synchronization time: Instead of launching a kernel for every single row in Fig. 5, we issue only one single kernel to generate the entire matrix profile. Note that based on the one-to-one correspondence between $d_{i,j}$ and $QT_{i,j}$ (as shown in (1)), we can convert the symmetric distance matrix computation into Fig. 9, where we evaluate the upper-right half of the dot product matrix. Since the value of $QT_{i,j}$ is only dependent on $QT_{i-1,j-1}$ (according to [4]), the computation of each diagonal in Fig. 9 is independent of any other diagonal. Thus, we assign $n - m + 1$ threads to compute these diagonals in parallel.

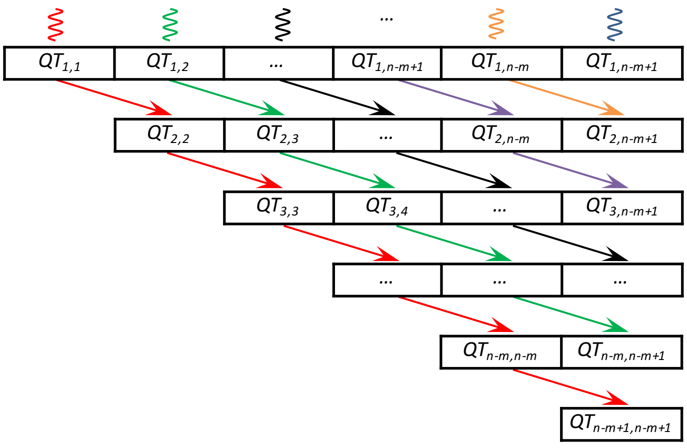


Fig. 9 An optimization scheme for the third step of GPU-STOMP. We only need to launch one kernel to evaluate all the rows of the distance matrix in Fig. 5

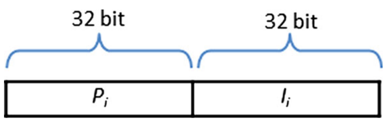


Fig. 10 We reduced the matrix profile to 32 bits and then combined each matrix profile entry and its corresponding matrix profile index entry into a double-precision value to allow fast atomic updates

Once we obtain $QT_{i,j,}$, we can easily evaluate $d_{i,j}$ based on (1). Then we examine two elements of the matrix profile: If $d_{i,j} < P_i$, we set $P_i = d_{i,j}$; and if $d_{i,j} < P_j$, we set $P_j = d_{i,j}$. Note that as each thread in Fig. 9 operates independently, multiple threads may attempt to update the same entry of the matrix profile at the same time. We need to use CUDA atomic operations to organize this. Essentially, we set a lock for each entry of the matrix profile. When multiple threads try to update the same matrix profile entry, they line up to get the lock and perform an atomic Min operation in order. The reader may doubt that this can result in a significant cost of time, as it is possible that all threads can be lining up to update the same single matrix profile entry. However, in practice, we find that a large portion of these atomic operations can be pruned from the calculation.

Assume we have twenty atomic operations lined up to update a matrix profile entry, which has an initial value of 6.81, with the following distance values in order:

0.6, 4.46, 1.99, 6.98, 2.29, 2.95, 7.05, 1.47, 6.04, 2.72, 2.31, 3.2, 6.25, 9.33, 0.27, 2.62, 2.00, 2.74, 6.67, 2.34.

Since the matrix profile entry keeps track of the minimum distance value, only two updates would be executed: 0.6 and 0.27, that is, only 10% of this short sequence of data. Now let us randomly shuffle the data:

7.05, 2.29, 1.47, 0.27, 2.74, 2.95, 9.33, 2.34, 4.46, 2.00, 6.04, 2.72, 2.31, 3.2, 6.25, 6.98, 0.6, 2.62, 1.99, 6.67.

This time three updates would be executed: 2.29, 1.47, 0.27, that is, only 15% of the data; so again, it is only a small portion.

Note that our toy example here is a very short data sequence. In practice, for most time series only less than 0.1% distance values end up smaller than their corresponding matrix

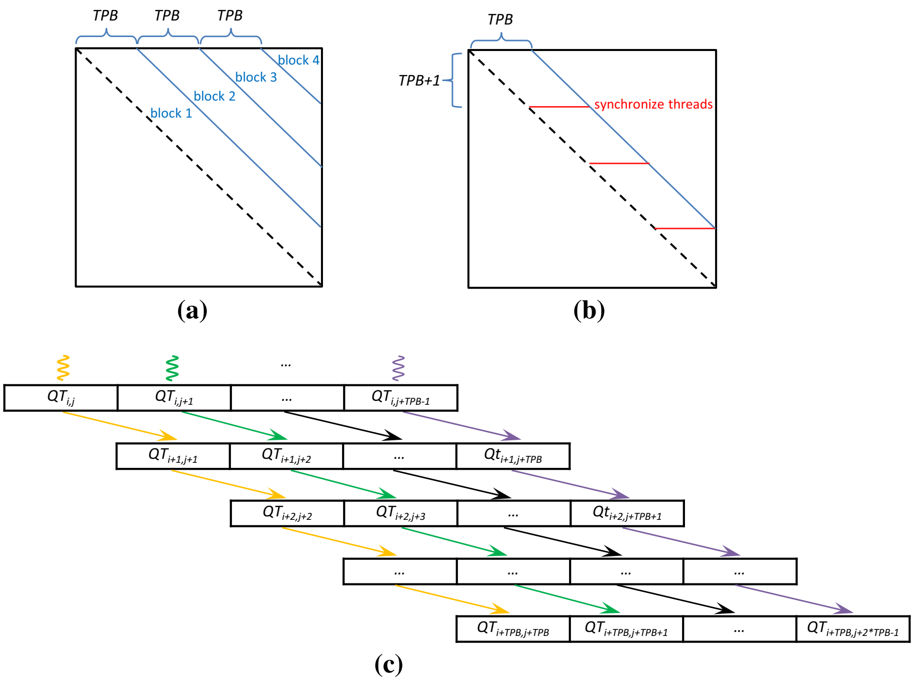


Fig. 11 **a** Each thread block evaluates one meta diagonal of the distance matrix. **b** The parallelograms in a meta diagonal are evaluated iteratively by a thread block. **c** The threads in a block evaluate diagonals of a parallelogram in parallel

profile elements. For example, for a random walk time series of length one million, we executed on average only 39 atomic calls for each matrix profile entry; more than 99.996% of the atomic operations are pruned.

By implementing the optimization scheme shown in Fig. 9, we have obtained about 3X speedup over GPU-STOMP for medium-size time series (i.e., with less than 4 million data points). However, as the time series gets even longer, less speedup is observed, as the time spent on atomic operations and global memory writes become nontrivial.

To solve this, we use two strategies to refine our optimization scheme in Fig. 9.

The first strategy aims to accelerate each atomic write. As stated previously, multiple independent threads can be attempting to update the matrix profile at the same time, so we are using CUDA atomic Min operation to organize them. Note that when a matrix profile entry (which is a 64-bit double-precision value) is updated, the corresponding matrix profile index value (a 32-bit integer value) also needs to be updated. However, currently CUDA only supports atomic operations on either one single 32-bit value or one single 64-bit value. To tackle this, we initially set a lock on every entry of the matrix profile and used a critical section to update both the matrix profile entry and the matrix profile index value when a thread gets the lock; however, this solution is not scalable with longer time series inputs. As a result, we turned to a better solution as shown in Fig. 10. Instead of using a time-consuming critical section, we lower the precision of the matrix profile to 32 bits. We then combine the matrix profile and the matrix profile index into one double-precision vector in the global memory that can be atomically updated. For the ith entry of the double-precision vector, 32 bits are used to store the ith matrix profile value, and another 32 bits are used to store the ith matrix profile index.

This refinement strategy largely accelerated the speed for atomic operations. Note that the strategy will not result in large precision loss, as only the precision of the output is reduced; we are still using 64 bits to store all the intermediate results during the evaluation process.

The second strategy is to utilize the CUDA shared memory to ease the contention for global memory writes. The strategy, as shown in Fig. 11, can be viewed as 2-level hierarchy of Fig. 9. Here we define *TPB* as the number of threads per block on CUDA.

Different from Fig. 9, in which each thread evaluates one single diagonal of the distance matrix, here we divide the distance matrix into $k$meta diagonals (as shown in Fig. 11.a, a meta diagonal consists of *TPB* diagonals of the distance matrix; $(k-1) \times TPB < n - m + 1 \leq k \times TPB$). Each meta diagonal is evaluated by one CUDA thread block. As shown in Fig. 11.b, the thread block evaluates one parallelogram at a time, managing a local copy of the matrix profile in the shared memory. The threads in a block (shown in Fig. 11.c) work very similarly as those in Fig. 9, except that they atomically update the shared memory instead of the global memory. After a parallelogram (Fig. 11.b) is evaluated, all the threads in the block are synchronized. If any value in the shared memory is smaller than its corresponding entry in the global memory, the global memory is updated.

With this refinement strategy, the contention of atomic updates in Fig. 9 is largely relieved. The original scheme in Fig. 9 allowed a global memory location to be visited by all active threads in all the thread blocks [which can be as many as $(n-m+1)$ threads] simultaneously. In contrast, with the refined scheme in Fig. 11, the number of threads racing for a shared memory location cannot be larger than *TPB*, and a global memory location cannot receive more than k atomic update requests at the same time. This brings about a large performance gain.

Similar to GPU-STOMP, GPU-STOMP$_{\text{OPT}}$ can easily be adapted to multiple GPUs as well. For example, to evenly divide the work for an NVIDIA Tesla K80, we compute the odd (first, third, fifth, etc., from the left) meta diagonals in Fig. 11.a with the first GPU, and the even (second, fourth, sixth, etc., from the left) meta diagonals in Fig. 11.a with the second GPU.

With all the optimization strategies, GPU-STOMP$_{\text{OPT}}$ achieved more than 2X speedup over GPU-STOMP for large datasets. Concretely, it further reduces the time to process a time series of length 100 million from 12 days to about 4 days on NVIDIA Tesla K80. Furthermore, for the first time in the literature, we are able to process a time series of length 143 million, which is slightly more than ten quadrillion pairwise comparison of subsequences, within just 9 days.

5 Empirical evaluation

Although some parts of our experiments require access to a GPU, we have designed them so they can be reproduced easily. To allow for the reproduction of our experiments, we have constructed a webpage [22], which contains all datasets and code used in this work. We begin with a careful comparison to STAMP, which is obviously the closest competitor, and we consider more general rival methods later.

Unless otherwise noted, we used an Intel i7@4 GHz PC with 4 cores to evaluate all the CPU-based algorithms; we used a server with two Intel Xeon E5-2620@2.4 GHz cores and an NVIDIA Tesla K80 GPU to evaluate GPU-STOMP.

Table 3 Time required for motif discovery with $m = 256$, varying n, for the three algorithms under consideration

Algorithm	Value of n				
	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
STAMP	15.1 min	1.17 h	5.4 h	24.4 h	4.2 days
STOMP	4.21 min	0.3 h	1.26 h	5.22 h	0.87 days
GPU-STOMP	10 s	18 s	46 s	2.5 min	9.25 min

Table 4 Time required for motif discovery with various m and various n, for the three algorithms under consideration

Algorithm	$m\|n$	
	2000 | 17,279,800	400 | 100,000,000
STAMP (estimated)	36.5 weeks	25.5 years
STOMP (estimated)	8.4 weeks	5.4 years
GPU-STOMP (actual)	9.27 h	12.13 days

5.1 STAMP versus STOMP

We begin by demonstrating that STOMP is faster than STAMP, and also that this difference grows as we consider increasingly large datasets. Furthermore, we measure the gains made by using GPU-STOMP. In Table 3, we measure the performance of the three algorithms on increasingly long random walk time series with a fixed subsequence length 256.

Note that we choose m's length as a power-of-two only to offer the best case for (the FFT-based) STAMP; our algorithm is agnostic to such issues.

A recent paper on finding motifs in seismograph datasets also considers a dataset of about 2^{19} in length and reports taking 1.6 h, which is approximately the same time it takes STOMP [36]. However, their method is probabilistic and allows false negatives (twelve of which were actually observed, after checking against the results of a 9.5 day brute-force search [36]). Moreover, it requires careful tuning of several parameters, and it does not lend itself to GPU implementation.

We wish to consider the scalability of even larger datasets with GPU-STOMP. However, in order to do so, we must estimate the time it takes the other two other algorithms. Fortunately, both of the other algorithms allow for an approximate prediction of the time needed, given the data length n. To obtain the estimated time, we evaluated only the first 100 distance profiles of both STAMP and STOMP and multiplied the time used by $(n-m+1)/100$. In Table 4, we consider even larger datasets, one of which reflects the data used in a case study in Sect. 5.4.

Note that the 100-million-length dataset is one hundred times larger than the largest motif search in the literature [14].

In all three algorithms under consideration, the time required is independent of the subsequence length m, which is desirable. This is demonstrated in Table 5, where we measure the time required with n fixed to 2^{17}, for increasing m.

Note that the time required for the longer subsequences is slightly shorter. This is true since the number of pairs that must be considered for a time series join [35] is $(n - m + 1)^2$, so as m becomes larger, the number of comparisons becomes slightly smaller.

Table 5 Time required for motif discovery with $n = 2^{17}$, varying m, for the three algorithms under consideration

Algorithm	Value of m				
	64	128	256	512	1024
STAMP	15.1 min	15.1 min	15.1 min	15.0 min	14.5 min
STOMP	4.23 min	4.33 min	4.21 min	4.23 min	2.92 min
GPU-STOMP	10 s	10 s	10 s	10 s	10 s

5.2 GPU-STOMP$_{\text{OPT}}$ breaks the ten quadrillion pairwise comparison barrier

In Table 6, we measure the performance of STAMP, STOMP, GPU-STOMP, and GPU-STOMP$_{\text{OPT}}$ on increasingly long random walk time series with a fixed subsequence length 256. The italic values are duplicated from Table 3, but they are included for comparison. Note that while some numbers are estimated, as explained in the next section, we can predict the time and memory requirement of STAMP and STOMP very precisely (with less than 5% error) for large datasets.

We note in passing that this experiment on a time series of length 143 million is the largest time series ever searched for exact motifs. Moreover, we are confident that this is the first time ten quadrillion pairwise comparisons have been made on a single dataset, in any context.

The time required for GPU-based algorithms can be divided into two parts. The first part includes the data reading time and computation time; the second part includes the time needed for kernel launch, data synchronization and memory writes. GPU-STOMP and GPU-STOMP$_{\text{OPT}}$ spent the same time in the first part. To further compare the effectiveness of the two methods, in Table 7 we measure their run time in the second part.

We can wee that GPU-STOMP$_{\text{OPT}}$ achieved more than 4X speedup over GPU-STOMP in the second part for large datasets.

5.3 STOMP versus state-of-the-art motif discovery algorithms

Beyond the independence of the subsequence length demonstrated in Table 5, all three matrix profile-based algorithms are also independent of the intrinsic dimensionality of the *data*, which is also desirable. To demonstrate this, we will compare the recently introduced Quick-Motif framework [14] and the more widely known MK algorithm [19]. The Quick-Motif method was the first technique to perform an exact motif search on one million subsequences.

To level the playing field, we do not avail of GPU acceleration, but instead, we use the identical hardware (a PC with Intel i7-2600@3.40 GHz) and programming languages for all algorithms. Note that for a fair comparison with STAMP [35], which is written in MATLAB, in Sect. 5.1, we measured the performance of STOMP based on its MATLAB implementation. However, because the two rival methods in this section (Quick-Motif and MK) are written in C/C++, here we measure the runtime of (the CPU version of) STOMP based on its C++ implementation.

We use the original author's executables [23] to evaluate the runtime of both MK and Quick-Motif. The reader may wonder why the experiments here are less ambitious than in the previous sections. The reason is that beyond *time* considerations, the rival methods have severe *memory* requirements. For example, for a seismology data with $m = 200$, $n = 2^{18}$, we measured the Quick-Motif memory footprint as large as 1.42 GB. In contrast, STOMP

Table 6 Time required for motif discovery with $m = 256$, varying n, for the four algorithms under consideration

Algorithm	Value of n							
	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	17,279,800	100,000,000	143,000,000
STAMP	*15.1 min*	*1.17 h*	*5.4 h*	*24.4 h*	*4.2 days*	36.5 weeks (estimated)	25.5 years (estimated)	51.2 years (estimated)
STOMP	*4.21 min*	*0.3 h*	*1.26 h*	*5.22 h*	*0.87 days*	8.4 weeks (estimated)	5.4 years (estimated)	10.9 years (estimated)
GPU-STOMP	*10 s*	*18 s*	*46 s*	*2.5 min*	*9.25 min*	9.27 h	12.13 days	24.5 days (estimated)
GPU-STOMP$_{\text{opt}}$	8 s	9 s	17 s	49 s	2.93 min	3.29 h	4.51 days	9.33 days

Table 7 Time required for kernel launch, data synchronization and memory writes with $m = 256$, varying n, for the two GPU-based algorithms

Algorithm	Value of n					
	2^{18}	2^{19}	2^{20}	2^{21}	17,279,800	100,000,000
GPU-STOMP	17 s	41 s	2.2 min	8.17 min	8.04 h	10.39 days
GPU-STOMP$_{\text{opt}}$	8 s	12 s	32 s	1.85 min	2.06 h	2.77 days

Table 8 Time required for motif discovery with $n = 2^{18}$, varying m, for various algorithms

Algorithm	m			
	512	1024	2048	4096
STOMP	501 s (14 MB)	506 s (14 MB)	490 s (14 MB)	490 s (14 MB)
Quick-Motif	27 s (65 MB)	151 s (90 MB)	630 s (295 MB)	695 s (101 MB)
MK	2040 s (1.1 GB)	N/A (> 2 GB)	N/A (> 2 GB)	N/A (> 2 GB)

requires only 14 MB memory for the same data, which is less than 1/100 of this. If this ratio linearly interpolates, Quick-Motif would need more than 1/2 terabyte of main memory to tackle the one hundred million benchmark, which is infeasible. Moreover, for Quick-Motif, it is possible that a different dataset of the same size could require a larger or smaller footprint. In contrast, the space required for STOMP is independent of both the structure of data and the subsequence length.

This severe memory requirement makes it impossible to compare the STOMP algorithm with Quick-Motif on the seismology data, since Quick-Motif often crashed with an *out-of-memory* error as we varied the value of m. However, we noticed that the memory footprint for Quick-Motif tends to be much smaller with smooth data. Therefore, instead of comparing performance of the algorithms on seismology data, in Table 8, we utilized the much smoother ECG dataset (used in Rakthanmanon et al. [24]), which is an ideal dataset for both MK and Quick-Motif to achieve their *best* performance.

Clearly, both the runtime and memory requirement for STOMP are independent of the subsequence length. In contrast, Quick-Motif and MK both poorly scale in subsequence length in both runtime and memory usage. Note that the memory requirement of Quick-Motif is not monotonic in m, as reducing m from 4096 to 2048 requires three times as much memory. This is not a flaw in implementation (we used the author's own code), but a property of the algorithm itself.

As indicated in Fig. 2, the Quick-Motif algorithm [14], the MK algorithm [19], and the original motif discovery by *projection* algorithm [8] can all be fast in the *best* case. For example, if there happens to be a perfect (zero Euclidean distance) motif in the dataset, they will all discover it with O(n) work (with high constants), and all algorithms can use this zero-valued best-so-far to prune all other possibilities for motif pairs. While we generally do not expect to have a zero-distance motif in real-valued data, a very close motif pair in a dataset with low intrinsic dimensionality (recall Fig. 2) can offer similar speedups. However, that describes the *best* case for all three algorithms. Consider instead the *worst* case (for example, the input signal is white noise, and all subsequences are effectively equidistant from each other), all three rival algorithms degenerate to O(mn^2) (again, with high constants). In contrast, STOMP is unique in that its best case and worse case are identical, just O(n^2).

Because m can be as large as 2000 (see Fig. 12), this can produce a significant speedup. Moreover, as we will show in the next two sections, STOMP computes much more useful information than the two rival methods.

Before demonstrating this, we show that the experiments in the previous table were spurious for STOMP. We do not need to *measure* its time or memory footprint, because we can predict it precisely. To the best of our knowledge, this property is unique among all motif discovery algorithms proposed in the literature [8,14,19].

For STOMP (assuming only that $m \ll n$), given only n, we can predict how long the algorithm will take to terminate and how much memory it will consume, which is completely independent of the value of m and the data.

To do this, we need to do a single calibration run on the machine in question. With a time series of length n, we measure T, the time taken to compute the matrix profile, and M, the (maximum) amount of memory consumed. Then, for any new length n_{new}, we can compute T_{required}, the time needed as the following:

$$T_{\text{required}} = \frac{T}{n^2} \times n_{\text{new}}^2 \tag{5}$$

and we can compute M_{required} the memory needed as the following:

$$M_{\text{required}} = \frac{M}{n} \times n_{\text{new}} \tag{6}$$

As long as we avoid trivial cases, such as $m \sim n$, n_{new} is very small or n is very small, and this formula will predict the resources needed with an error of less than 5%. To demonstrate this, we performed the following experiment. On our machine (a PC with Intel i7-2600@3.40 GHz) we ran STOMP (MATLAB version) on a random walk dataset of size 2^{18}, measuring the resources consumed. Then, as shown in Table 9, we use the formulas above to predict the resources needed to compute the matrix profile for datasets of size $\{2^{18}, 2^{19}, 2^{20}, 2^{21}\}$. Then we measured these values with actual experiments on random walk data. From Table 9, the agreement between our predictions and the observed values is clear.

This property has several desirable implications: We can carefully plan resources when performing analytics on large data archives; we can easily divide the work to parallel computing resources to finish our task in time; and we can show a perfectly accurate "progress bar" to a user who is using STOMP interactively.

5.4 Case studies in seismology: infrequent earthquake case

To allow confirmation of the correctness and utility of STOMP, we begin by considering a dataset for which we know the result from external sources. On April 30, 1996, there was an earthquake of magnitude 2.12 in Sonoma County, California.[2] Then on December 29, 2009, about 13.6 years later, there was another earthquake with a similar magnitude. We concatenated the two full days in question to create a single time series of length 17,279,800 (see Table 4 for timing results) and examined the top motifs with $m = 2000$ (20 s). Note that we are using the raw data as provided to us by the seismologists, we are not preprocessing it in anyway. As Fig. 12.*top* illustrates, the top motif here is *not* an earthquake but an unusual sensor artifact [11].

There are a handful of other such artifacts; however, as shown in Fig. 12.*bottom*, the fifth best motif *is* the two occurrences of the earthquake. These misleading sensor artifacts are

[2] A small earthquake of that magnitude would only be felt by attentive humans in the immediate vicinity of the epicenter.

Table 9 Time and memory required for STOMP, with $m = 256$, varying n

Resources	n			
	2^{18}	2^{19}	2^{20}	2^{21}
STOMP time (memory) measured	19.0 min (30 MB)	75.6 min (60 MB)	313.2 min (121 MB)	1252.8 min (242 MB)
STOMP time (memory) predicted	19.0 min (30 MB)	76.0 min (60 MB)	304 min (120 MB)	1216 min (240 MB)
Relative error	0% (0%)	0.5 % (0%)	3% (0.8%)	3% (0.8%)

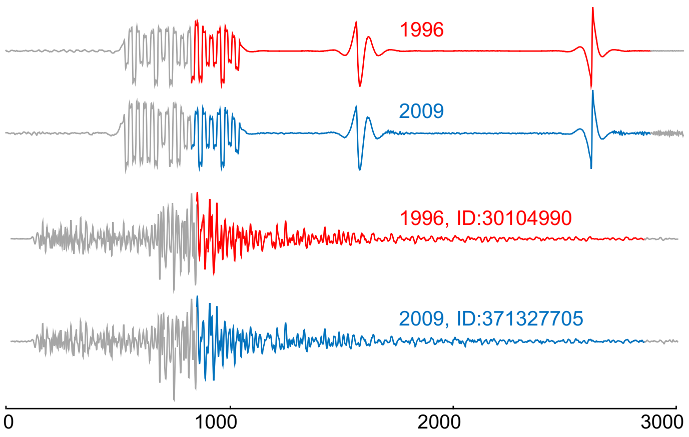


Fig. 12 Motifs (colored) shown in context (gray). Top The top motif discovered in the Sonoma County dataset is a sensor artifact, as are the next three motifs (not shown). Bottom The fifth motif is two true occurrences of an earthquake that happen 4992 days apart (color figure online)

common, but they could be eliminated easily [11]. For example, the sensors could have a zero crossing rate that is an order of magnitude lower than true earthquakes.

This example allows us to demonstrate yet another advantage of STOMP over rival methods. All the existing rival techniques can be expanded from top-1 motif discovery to top-k motif discovery; however, increasing k by a modest amount will significantly degrade their speed.

Furthermore, consider again the example in Fig. 12. It is not possible to have known the "magic" value of $k = 5$ beforehand. If k was set to a large value to "be on the safe side," say $k = 10$, then all existing techniques would slow down because the best-so-far lower bound that prunes unnecessary computations would be much looser. If we set k as a more conservative value, say $k = 3$, then we would miss the most valuable information in this seismology dataset. You might imagine that the rival methods could slowly increase from k to $k + 1$ based on the user's lack of satisfaction with the k motifs she has examined thus far; however, each adjustment of k will require all existing techniques to perform significant extra computation, *even* if they have cached the results of every calculation they have performed.

In contrast, the time needed for STOMP is completely independent of k. We only need to run STOMP once; as the matrix profile obtained already contains all necessary information, and it takes minimal additional effort to find the top k motif, no matter how large k is.

5.5 Parameter settings

As we have previously noted, STOMP (together with STAMP) is unique among motif discovery algorithms because it is parameter-free. In contrast, Random Projection [8] has four parameters, Quick-Motif [14] has three parameters, Tree-Motif has four parameters [33], MK [19] has one parameter, and FAST has three parameters [36].

That being said, the reader may wonder about the only input value besides the time series of interest: the subsequence length m. Note that this is also a required input for all the other existing techniques. However, we do not consider m to be a true parameter, as it is a *user choice,* reflecting her prior knowledge of the domain. Nevertheless, it is interesting to ask how sensitive motif discovery is to this choice; at least in the seismology domain that motivates us.

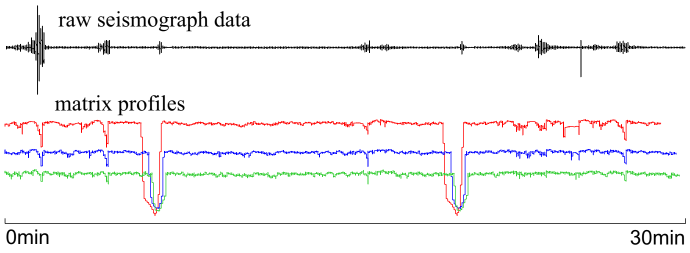


Fig. 13 Top Thirty minutes of seismograph data that has the two earthquakes from Fig. 12.*bottom* occur at 6 min 40 s and 20 min. Bottom The matrix profile computed if we use the suggested subsequence length 2000 (blue), or if we use twice the length (red), or half that length (green) (color figure online)

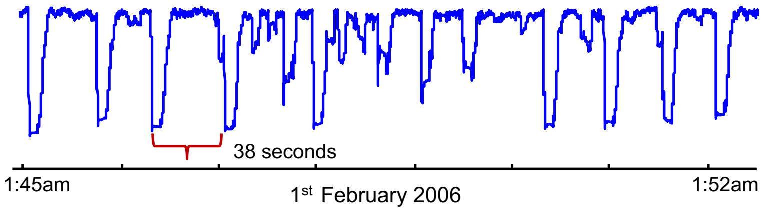


Fig. 14 The matrix profile of a seven-minute snippet from a seismograph recording at Mount St. Helens

To test this, we edited the data above such that the two earthquakes in Fig. 12.*bottom* happen exactly 13 min 20 s apart. We reran motif discovery with $m = 2000$ (20 s), with double that length ($m = 4000$), and with half that length ($m = 1000$). Figure 13 shows the result.

The results are reassuring. At least for earthquakes, motif discovery is not sensitive to the user input. Even a poor guess as to the best value for m, it will likely give accurate results.

5.6 Case studies in seismology: earthquake swarm case

In the previous section, we discovered a repeating earthquake source that has a frequency of about once per 13.6 years. Here, we consider earthquakes that are tens of millions of times more frequent.

Forecasting volcanic eruptions is of critical importance in many parts of the world [30]. For example, on May 18, 1980, Mount St. Helens had a paroxysmal eruption that killed 57 people [13]. It is conjectured that explosive eruptions are commonly preceded by elevated or accelerated gas emissions and seismicity; thus, seismology is a major tool for both monitoring and predicting such events.

In Fig. 14, we illustrate a short section of the matrix profile of a seismograph recording at Mount St Helens. It is important to restate that this is *not* the raw seismograph data, but it is the matrix profile that STOMP computed from it.

The image demonstrates a stunning regularity. Repeated earthquakes are occurring approximately once every 38 s. This is consistent with the findings of a team from the US Geological Survey who reported that the earthquakes, which accompanied a dome-building eruption, appeared "... *so regularly that we dubbed them 'drumbeats'. The period between successive drumbeats shifted slowly with time, but was 30–300 s*" [13].

This example shows a significant advantage of our approach that we share with STAMP but no other motif discovery algorithm. Instead of computing only O(1) distance values for the top k motifs, STOMP is computing *all* O(n) distances from *every* subsequence to their nearest neighbors. By plotting the entire matrix profile, gain unexpected insights by viewing the motifs *in context*. For example, in the example above, we can see both the surprising periodicity of the earthquakes, and by comparing the smallest values in the matrix profile with the mean or maximum values, we can get a sense of how well the motifs are conserved relative to "chance" occurrences. It could also potentially indicate whether there were changes to the earthquake source, reflecting changes in eruptive behavior over time.

A recent paper performed a similar analysis on the Mount Rainier volcano, making the interesting and unexpected discovery that the frequency of earthquakes is correlated with snowfall [2]. However, the paper bemoans at the number of ad hoc "hacks" that needed to make such an exploration tenable. For example, "*In order to save on computing time, we cut out detections that are unlikely to contain a repeating earthquake event by excluding events with a signal width*", and "*To save on computing time, we define that in order to be detected…*", etc. [2]. However, the results in Table 4 indicate that we could bypass these issues by spending a few hours computing the *full* exact answers. This would eliminate the risk that some speedup "trick" erases an interesting and unexpected pattern.

5.7 Case studies in seismology: detection of repeated low-frequency earthquakes

In the previous sections, we showed how STOMP could help us detect repeating earthquake sources by evaluating the matrix profile of a *single* seismograph recording time series. Here we show that by providing the matrix profiles of *multiple* seismograph recording time series, STOMP allows us to detect low-frequency earthquakes (LFEs). LFEs are of great importance to the seismology community, as they could "*potentially contribute to seismic hazard forecasting by providing a new means to monitor slow slip at depth*" [26]. LFEs recur episodically, often during bursts of tectonic "tremor," which are considered superpositions of many LFEs in a short period of elevated seismic activity [27]. One traditional approach known as "matched filtering" identifies repeated LFEs by evaluating the cross-correlation between continuous waveform data (time series) and a template waveform (subsequence) (e.g., [28]). However, this requires a suitable, carefully recorded template waveform of an LFE (an LFE subsequence) to have been identified in advance, which is very difficult or even impossible in many cases. In the face of this, similarity–join search through autocorrelation (e.g., [6]) has been used to detect LFEs in several studies. However, the traditional similarity–join search approach is computationally intensive (typically only one hour or less of continuously waveform data can be searched in feasible time), severely limiting the number and range of LFEs that can be detected.

Consider an example of LFE detection along the central San Andreas fault near Parkfield, CA. We search for LFEs in waveform data from a tremor burst that occurred on October, 6, 2007, in which many LFEs were detected by matched filtering [28]. As before, note that we are using the raw data as provided to us by the seismologists, we are not preprocessing it in anyway. The LFE template (subsequence) in Shelly et al. [28] was found by careful visual examination of seismic recording from multiple temporary seismic stations located close to the source (the green triangles in Fig. 15; temporary stations were set up near a well-known earthquake source in this area) and subsequently also identified on more distant, permanent High Resolution Seismic Network (HRSN, the red triangles). Note that our task here is to detect all the LFEs automatically, and the only data available are those from the HRSN stations (the red triangles in Fig. 15), since in most applications we do not know the earthquake source location (thus the data from the temporary stations) until well after the event.

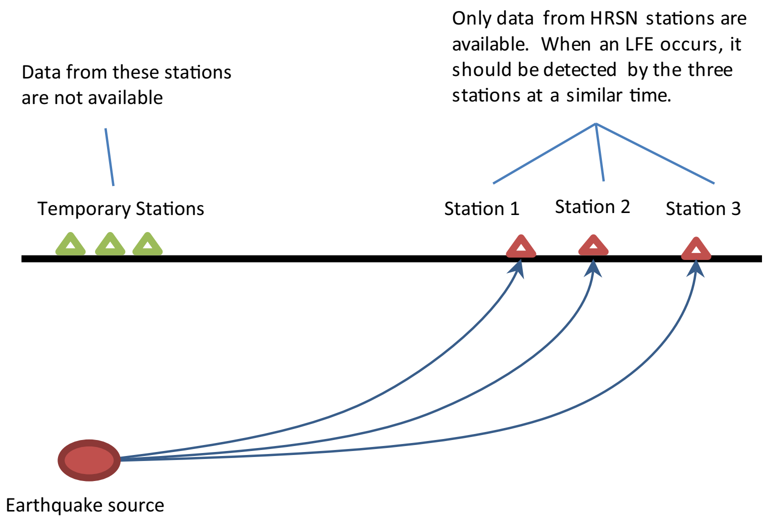

Only data from HRSN stations are available. When an LFE occurs, it should be detected by the three stations at a similar time.
Data from these stations are not available
Temporary Stations
Station 1
Station 2
Station 3
Earthquake source

Fig. 15 LFEs can be detected from the seismograph recording of HRSN stations

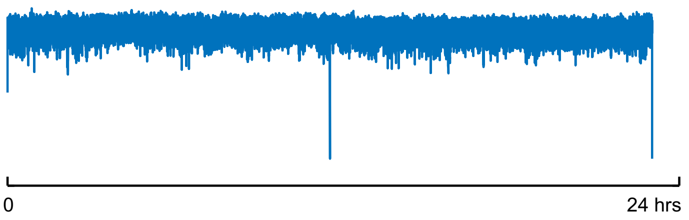


Fig. 16 The sum of three matrix profiles of the 24-hour seismograph recording at three HRSN stations near the central San Andreas fault

Apart from the lack of the temporary station data, what makes our task even more difficult is that the data from HRSN stations are noisy and many contain a lot of false positives. For example, the top 15 motifs (repeating templates) found from the data of an HRSN station near central San Andreas fault are either sensor artifacts (similar to Fig. 12) or instrument noise in the station itself. However, in spite of all these difficulties, we will demonstrate that STOMP allows us to detect LTEs from long seismic recordings.

We ran GPU-STOMP$_{\text{OPT}}$ on the seismic recording time series from three HRSN stations for a 24-hour period spanning the tremor burst. The three HRSN stations are located close to each other. The data were sampled at 20 Hz, for a total of ∼1.7 million samples per station time series. Figure 16 shows the sum of the three matrix profiles obtained.

The reader may wonder why we are summing the three matrix profiles here. This simple step greatly reduces the false positives in the data. As the three HRSN stations are located close to each other, when an LFE occurs, the stations should detect it at a similar time. As a result, the matrix profile values of the three stations should all be low at the occurrence of the LFE. The sum of the matrix profiles shows low values at such time instants, which strengthens the LFE signal and thus weakens the false positives, which are local to each sensor. We discovered that the top seven motifs identified in this way were either glitches in the waveform data (sensor artifacts, again, recall Fig. 12), or signals that could not be separated into individual LFEs; however, as shown in Fig. 17, the 8th best motif showed

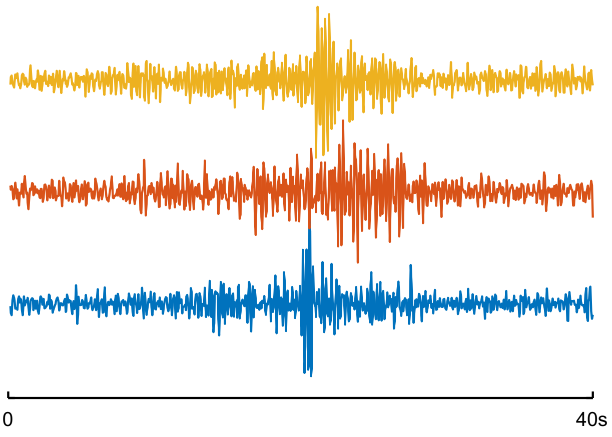


Fig. 17 The 40-second LFE snippet detected from the three HRSN station time series

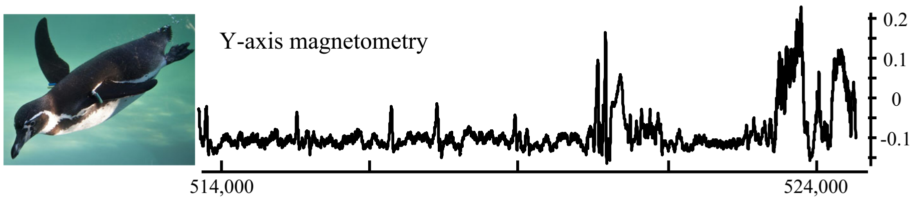


Fig. 18 Left The Magellanic penguin is a strong swimmer. Right A four-minute snippet of the full dataset reveals high levels of noise and no obvious structure

strong characteristics, in terms of frequency content, waveform shape and duration, of an LFE, and the origin time of this LFE is consistent with the results in Shelly et al. [28], which may be regarded as the ground truth.

In contrast to Shelly et al. [28], which detects the LFE pattern with *weeks* of enormous human effort, we are able to complete the same task automatically in approximately 3 min with GPU-STOMP$_{\mathrm{OPT}}$ on NVIDIA Tesla K80.

5.8 A case study in animal behavior

While seismology is the primary motivator for this work, nothing about our algorithm assumes anything about the data's structure, or precludes us from considering other datasets. To demonstrate this, in this section, we briefly consider telemetry collected from Magellanic penguins (*Spheniscus magellanicus*). Adult Magellanic penguins can regularly dive to depths of between 20 and 50m deep in order to forage for prey and may spend as long as fifteen minutes under water. The data were collected by attaching a small multi-channel data-logging device to the bird. The device recorded tri-axial acceleration, tri-axial magnetometry, pressure, etc. As shown in Fig. 18, for simplicity we consider only Y-axis magnetometry. Note that, as with the seismology, we are not preprocessing this data source in anyway, no smoothing, not downsampling, etc.

An observer with binoculars labels the data; thus, we have a coarse ground truth for the animal's behavior. The full data consist of 1,048,575 data points recorded at 40 Hz (about 7.5 h). We ran GPU-STOMP$_{\mathrm{OPT}}$ on this dataset, using a subsequence length of 2000. This

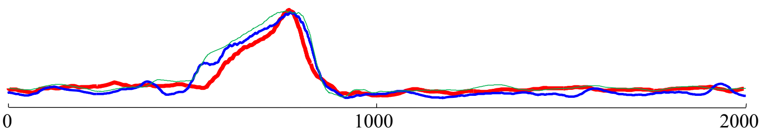


Fig. 19 The top motif of length 2000 discovered in the penguin dataset. Only three examples are shown for visual clarity, and there are eight such patterns. This behavior may be part of a "porpoise" maneuver

took our algorithm just 49 s to compute. As shown in Fig. 19, the top motif is a surprisingly well conserved "shark fin" like pattern.

What (if anything) does this pattern indicate? Suggestively, we observed this pattern does not occur in any of the regions labeled as *nesting*, *walking*, *washing*, etc., but only during regions labeled *foraging*. Could this motif be related to a diving (for food) behavior?

Fortunately, *diving* is the one behavior we can unambiguously determine from the data, as the *pressure* sensor reading increases by orders of magnitude when the penguin is under water. We discovered that the motif occurs moments before each dive, and nowhere else. This pattern appears to be part of a ritual behavior made by the bird before diving. It has been reported that "*The only time penguins are airborne is when they leap out of the water. Penguins will often do this to get a gulp of air before diving back down for fish*." Thus, we suspect this pattern in part of a "porpoise" behavior [29].

Generally speaking, we see this example as typical of the interactions that motif discovery supports. In most cases, motif discovery is not the *end* of analyses, but only the *beginning*. By correlating the observed motifs with other (internal or external) data, we can form hypotheses and open avenues for further research. Recall the previous section; this is rather similar to the team studying Mount Rainier's seismology discovered that its earthquakes are correlated with snowfall [2]. We believe that the STOMP algorithm may enable many such unexpected discoveries in a vast array of domains.

6 Conclusions

We introduced STOMP, a new algorithm for time series motif discovery, and showed that it is theoretically and empirically faster than its strongest rivals in the literature, STAMP [35], Quick-Motif [14] and MK [19]. In the limited domain of seismology, we showed that STOMP is at least as fast as the recently introduced FAST algorithm [36], but STOMP does not allow false negatives and does not need careful parameter tuning. Moreover, for datasets and subsequences lengths encountered in the real world, STOMP requires one to three orders of magnitude less memory than rival methods. Thus, even if we are willing to wait a longer period of time for the rival methods to search a large (ten million-plus) dataset, we will almost certainly run out of main memory. Given that these algorithms require random access to the data, disk-based implementations are infeasible. This is not a gap that is likely to be closed by a new implementation of these algorithms, because STOMP is unique among motif discovery algorithms in *not* extracting subsequences, but performing all the computations in situ.

We further demonstrated optimizations that allow STOMP to take advantage of GPU architecture, opening an even greater performance gap and allowing the first exact motif search in a time series of length one hundred and forty-three-million.

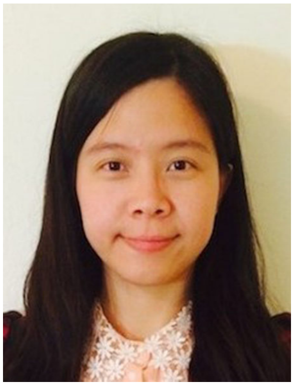
In future work, we plan to investigate multidimensional and incremental versions of our algorithms. The latter may have implications for real-time earthquake warning systems, which will reduce the probability of false alarms by quickly searching dictionaries of previously confirmed events [36].

Acknowledgements This research was funded by NSF IIS-1161997 II, NSF IIS-1510741, NSF CCF-1528181, NSF CCF-1527127 and USGS Earthquake Hazard Program Award G16AP00034. We gratefully acknowledge all the donors of the datasets.

References

1. Agrawal R, Faloutsos C, Swami A (1993) Efficient similarity search in sequence databases. Foundations of data organization and algorithmsm, 69–84
2. Allstadt K, Malone SD (2014) Swarms of repeating stick-slip icequakes triggered by snow loading at Mount Rainier volcano. J Geophys Res Earth Surf 119(5):1180–1203
3. Balasubramanian A, Wang J, Balakrishnan P (2016) Discovering multidimensional motifs in physiological signals for personalized healthcare. IEEE J Sel Top Signal Process 10(5):832–841
4. Bailis P, Gan E, Rong K et al (2017) Prioritizing attention in fast data: principles and promise. In: CIDR
5. Brown AEX, Yemini EI, Grundy LJ et al (2013) A dictionary of behavioral motifs reveals clusters of genes affecting caenorhabditis elegans locomotion. Proc Natl Acad Sci 110(2):791–796
6. Brown JR, Beroza GC, Shelly DR (2008) An autocorrelation method to detect low frequency earthquakes within tremor. Geophys Res Lett 35, L16305. https://doi.org/10.1029/2008GL034560
7. Chandola V, Banerjee A, Kumar V (2007) Anomaly detection: a survey. Technical report, University of Minnesota
8. Chiu B, Keogh E, Lonardi S (2003) Probabilistic discovery of time series motifs. In: SIGKDD, pp 493–498
9. Geller RJ, Mueller CS (1980) Four similar earthquakes in central California. Geophys Res Lett 7(10):821–824
10. Harris M (2007) Optimizing parallel reduction in CUDA. NVIDIA Developer Technology 2.4
11. Havskov J, Alguacil G (2004) Instrumentation in earthquake seismology, vol 358. Springer, Dordrecht
12. Igarashi T, Matsuzawa T, Hasegawa A (2003) Repeating earthquakes and interplate aseismic slip in the northeastern Japan subduction zone. J Geophys Res 108, 2249. https://doi.org/10.1029/2002JB001920.
13. Iverson RM, Dzurisin D, Gardner CA et al (2006) Dynamics of seismogenic volcanic extrusion at Mount St. Helens in 2004–2005. Nature 444(7118):439–443
14. Li Y, U LH, Yiu ML, Gong Z (2015) Quick-motif: An efficient and scalable framework for exact motif discovery. In: ICDE, IEEE, pp 579–590
15. Luo W, Tan H, Mao H et al (2012) Efficient similarity joins on massive high-dimensional datasets using mapreduce. In: MDM, IEEE, pp 1–10
16. McGovern A, Rosendahl D, Brown R et al (2011) Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. Data Min Knowl Discov 22(1):232–258
17. Meng X, Yu X, Peng Z et al (2012) Detecting earthquakes around salton sea following the 2010 mw7.2 El Mayor-Cucapah earthquake using GPU parallel computing. Procedia Comput Sci 9:937–946
18. Minnen D, Isbell CL, Essa I et al (2007) Discovering multivariate motifs using subsequence density estimation and greedy mixture learning. In: AAAI, pp 615–620
19. Mueen A, Keogh E, Zhu Q et al (2009) Exact discovery of time series motifs. In: SDM, pp 473–484
20. NVIDIA CUDA C Programming Guide (2016) Version 7.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
21. NVIDIA CUFFT Library User's Guide (2016) Version 7.5. http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf
22. Project Website (2017) http://www.cs.ucr.edu/~eamonn/MatrixProfile.html
23. Quick Motif (2015) http://degroup.cis.umac.mo/quickmotifs/
24. Rakthanmanon T, Campana B, Mueen A et al (2013) Addressing big data time series: mining trillions of time series subsequences under dynamic time warping. TKDD 7(3):10
25. Rong K, Bailis P (2017) ASAP: prioritizing attention via time series smoothing. VLDB Endowment 10(11):1358–1369
26. Shelly DR, Beroza GC, Ide S et al (2006) Low-frequency earthquakes in Shikoku, Japan, and their relationship to episodic tremor and slip. Nature 442(7099):188–191
27. Shelly DR, Beroza GC, Ide S (2017) Non-volcanic tremor and low-frequency earthquake swarms. Nature 446(7133):305–307

28. Shelly DR, Ellsworth WL, Ryberg T et al (2009) Precise location of San Andreas Fault tremors near Cholame, California using seismometer clusters: Slip on the deep extension of the fault? Geophys Res Lett 36, L01303. https://doi.org/10.1029/2008GL036367
29. Simeone A, Wilson RP (2003) In-depth studies of Magellanic penguin (Spheniscus magellanicus) foraging: Can we estimate prey consumption by perturbations in the dive profile? Mar Biol 143(4):825–831
30. Sparks RSJ (2003) Forecasting volcanic eruptions. Earth Planet Sci Lett 210(1):1–15
31. Tanaka Y, Iwamoto K, Uehara K (2005) Discovery of time-series motif from multi-dimensional data based on MDL principle. Mach Learn 58(2):269–300
32. Vahdatpour A, Amini N, Sarrafzadeh M (2009) Toward unsupervised activity discovery using multi-dimensional motif detection in time series. IJCAI 9:1261–1266
33. Wang L, Chng ES, Li H (2010) A tree-construction search approach for multivariate time series motifs discovery. Pattern Recognit Lett 31(9):869–875
34. Wang X, Mueen A, Ding H et al (2013) Comparison of representation methods and distance measures for time series data. Data Min Knowl Discov 26(2):275–309
35. Yeh CCM, Zhu Y, Ulanova L et al (2016) Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In: ICDM, IEEE, pp 579–588
36. Yoon CE, O'Reilly O, Bergen KJ et al (2015) Earthquake detection through computationally efficient similarity search. Sci Adv 1(11):e1501057



Yan Zhu is a Ph.D. student at the University of California, Riverside. She received her M.S. and B.S. degrees in 2013 and 2010, respectively, both from Shanghai Jiao Tong University, Shanghai, China. She has published papers in top data mining venues, including ICDM, ECML-PKDD, Data Mining and Knowledge Discovery, etc. Her research interests include data mining and machine learning.

Zachary Zimmerman is a Ph.D. student in computer science at the University of California, Riverside, where he also received his Bachelor's and Master's degrees in 2015 and 2017, respectively. His research interests include high-performance computing, data mining and machine learning. He is especially interested in scalable algorithms for real-time data.

Nader Shakibay Senobari received his B.S. in Physics from Sharif University of Technology, Iran in 2010. In 2013, He started his Ph.D. in Geophysics program at the University of California, Riverside. His research is mostly in the crossover between geodesy and seismology. He is using the data and methods from both of these subjects to study Earth's interior structure, fault behavior and the physics of earthquakes. He has received NASA Earth and Space Science Fellowship (NESSF) for three consecutive years (2015-17).

Chin-Chia Michael Yeh received his B.S. and M.S. degrees from Virginia Tech and University of California, Los Angeles in 2010 and 2011, respectively. From 2011 to 2014, he joined Research Center for Information Technology Innovation at Academia Sinica, Taiwan as a Research Assistant. He has been a Ph.D. Student at University of California, Riverside, since 2014. His research interests include time series analysis, data mining and machine learning.

Gareth Funning received a M.Sc. degree in Geophysics from the University of Durham in 2000 and a D.Phil. in Earth Sciences from Hertford College, Oxford in 2005. His doctoral research focused on methods for combining InSAR and teleseismic waveform data into joint inversions for earthquake source parameters. In 2005, he joined the Berkeley Seismological Laboratory as a postdoctoral researcher. Since 2007, he has been on the faculty of the University of California, Riverside, in the Department of Earth Sciences. His research interests include the study of fault creep using geodetic and seismic methods, reconciling earthquake centroid locations between InSAR and long-period seismology, detailed investigations of the earthquake source from geodetic data, and studies of anthropogenic deformation sources.

Abdullah Mueen is an assistant professor at the Department of Computer Science in the University of New Mexico. Earlier, he served as a scientist in Microsoft Corporation (2012–2013). Dr. Mueen is interested in analyzing large data to discover patterns and anomalies. He is the runner-up of ACM SIGKDD Doctoral Dissertation Award 2012 and his paper on mining trillion *subsequences* has won the best paper award in the ACM SIGKDD conference in 2012.

Philip Brisk received the B.S., M.S., and Ph.D. degrees, all in Computer Science, from UCLA in 2002, 2003 and 2006, respectively. From 2006–2009, he was a Postdoctoral Scholar in the Processor Architecture laboratory in the School of Computer and Communication Sciences at the École Polytechnique Fédérale de Lausanne, (EPFL), in Lausanne, Switzerland. Since 2009, he has been an Assistant Professor in the Department of Computer Science and Engineering at the University of California, Riverside and has been promoted to Associate Professor as of July 1, 2015. His work has received the Best Paper Award at CASES 2007 and FPL 2009, and been nominated for the Best Paper Award at DAC 2007 and HiPEAC 2010. Dr. Brisk has been the general (co-)chair of IEEE SIES 2009, IEEE SASP 2010 and IWLS 2011.

Eamonn Keogh a full professor at the University of California, Riverside, is a top-ten most prolific author in all three of the top ranked data mining conferences, SIGKDD, ICDM and SDM, and the most prolific author in the Data Mining and Knowledge Discovery Journal. Dr. Keogh has won best paper awards at ICDM, SIGKDD and SIGMOD. His H-index of 78 reflects the influence of time series representations and algorithms. He has given well-received tutorials at SIGKDD (five times), ICDM (four times), VLDB, SDM and CIKM.