algorithms

MDPI

Article

A Multithreaded Algorithm for the Computation of Sample Entropy

George Manis [1,*], Dimitrios Bakalis [1] and Roberto Sassi [2]

[1] School of Engineering, Department of Electrical and Computer Engineering, University of Ioannina, 45110 Ioannina, Greece; d.bakalis@uoi.gr
[2] Dipartimento di Informatica, Università degli Studi di Milano, 20133 Milano, Italy; roberto.sassi@unimi.it
* Correspondence: manis@cs.uoi.gr

Abstract: Many popular entropy definitions for signals, including approximate and sample entropy, are based on the idea of embedding the time series into an m-dimensional space, aiming to detect complex, deeper and more informative relationships among samples. However, for both approximate and sample entropy, the high computational cost is a severe limitation. Especially when large amounts of data are processed, or when parameter tuning is employed premising a large number of executions, the necessity of fast computation algorithms becomes urgent. In the past, our research team proposed fast algorithms for sample, approximate and bubble entropy. In the general case, the bucket-assisted algorithm was the one presenting the lowest execution times. In this paper, we exploit the opportunities given by the multithreading technology to further reduce the computation time. Without special requirements in hardware, since today even our cost-effective home computers support multithreading, the computation of entropy definitions can be significantly accelerated. The aim of this paper is threefold: (a) to extend the bucket-assisted algorithm for multithreaded processors, (b) to present updated execution times for the bucket-assisted algorithm since the achievements in hardware and compiler technology affect both execution times and gain, and (c) to provide a Python library which wraps fast C implementations capable of running in parallel on multithreaded processors.

Keywords: entropy; sample entropy; fast algorithm; parallel algorithm; bucket-assisted algorithm

Citation: Manis, G.; Bakalis, D.; Sassi, R. A Multithreaded Algorithm for the Computation of Sample Entropy. *Algorithms* **2023**, *16*, 299. https://doi.org/10.3390/a16060299

Academic Editors: Charalampos Konstantopoulos and Grammati Pantziou

Received: 30 April 2023
Revised: 2 June 2023
Accepted: 9 June 2023
Published: 15 June 2023

1. Introduction

Despite the rapid evolution of hardware technology, high-performance algorithms continue to pose significant challenges when it comes to achieving speed. Exploiting the capabilities of underlying hardware, reducing computational complexity, exploring novel algorithmic approaches and parallelizing computing tasks have all proven to be critical factors in high performance computing [1,2].

Sample entropy [3,4] is a definition of entropy which presents increased computational complexity and, therefore, slow execution times. Since the fast computation of sample entropy is crucial when a large amount of data is to be analyzed, we investigate parallel algorithms to achieve faster executions. Sample entropy has emerged as an improvement over approximate entropy [5], a similar measure of system complexity. They both assess the probability of a pattern in a time series being repeated at two consecutive scales, namely m and $m+1$. A higher value of entropy indicates a more irregular or complex time series.

Sample entropy finds its application in various fields, including biomedical signal analysis, financial time series analysis and seismographic data interpretation. For instance, in biomedical signal analysis, sample entropy has been employed to study the dynamics of physiological processes by examining the complexity and irregularity of heart rate variability [4,6,7]. It has also been used in several other fields of biomedical engineering, where it finds often application on EEG analysis, for example, in automatic epileptic seizure

detection [8], or in arrhythmia detection, for example, in paroxysmal atrial fibrillation detection [9], or even to analyze human postural sway data [10]. Financial time series analysis has also benefited from the use of sample entropy algorithms, enabling the assessment of system complexity and the detection of patterns in financial data [11]. Moreover, seismographic data analysis has utilized sample entropy to uncover correlations and variability in electrical signals related to earthquake activity, contributing to a deeper understanding of complex phenomena underlying seismic events [12].

Given the increasing volume of data being generated across various domains, efficient computation becomes crucial. To address this challenge, researchers have explored parallel algorithms to achieve faster execution times. Especially for sample entropy, slow execution times due to the inherent computational complexity can hinder the timely analysis of large data sets. Leveraging parallel computing techniques and utilizing the computational resources available in modern hardware architectures can significantly accelerate the computation of sample entropy, enabling the efficient analysis of large data sets within reasonable time frames.

In our past research, we proposed two sequential fast algorithms for the rapid computation of sample entropy [13]. The first one, called the *bucket-assisted algorithm*, is still, to the best of our knowledge, the fastest way to compute sample entropy in the general case. In the special case, in which $m = 1$, the second algorithm proposed in the same paper, the *lightweight* algorithm, presents lower execution times. Earlier, a fast algorithm for the computation of approximate entropy was also presented by our team [14].

In this paper, parallel approaches to the bucket-assisted algorithm are investigated. We focus on the multithreading technology, which is now supported by every processor (even the cost-effective ones used in our home computers). Multithreaded software becomes more and more important, and the development of libraries targeting high performance for cost-effective hardware emerges. A multithreaded Python library, based on underlying C code, implementing the fast, parallel algorithms examined here, is publicly available (https://cs.uoi.gr/~manis/entropy_threads/, accessed on 2 June 2023).

Taking this opportunity, we also present new execution times and comparisons with the straightforward implementation for the bucket-assisted algorithm. Having focused on loops, compiler and hardware technology has affected both speed and potential gain. We will show that the bucket-assisted algorithm still presents very fast execution times and that the bucket-assisted algorithm presents remarkable parallelization rates and low execution times when only employing the available processor threads.

Similar to other entropy definitions [5,15,16], sample entropy embeds the time series, composed of n samples, into an m-dimensional space. It checks every pair of vectors in the embedding space for similarity. This is an $O(n^2)$ effort. The number of comparisons is quite large for large signals and each comparison is also expensive. The key idea behind the bucket-assisted algorithm is to reduce the number of comparisons by excluding those similarity checks, which we know a priori are going to fail. Specifically, points for which we know that, due to their relative position in the embedding space, the distance between them, in at least one or more threshold distances, is excluded from the similarity test. The algorithm is still $O(n^2)$, since in the worst case, all possible pairs will be checked, but the number of comparisons performed in practice is now much smaller.

The contribution of the paper is threefold: (a) to propose a new parallel algorithm for the computation of sample entropy, which runs in parallel even on cost-effective home computers; (b) to propose and publish a new library, (C code in a Python wrapper) in order to disseminate code and make the suggested algorithm available for everyone to use, without having to deal with C or the complicated implementation details of bucket-assisted algorithm; and (c) to provide new execution times and speedups for the bucket-assisted algorithm, since the new achievements of the compiler and hardware technology and in loops onto hardware have influenced the already published results.

The rest of the paper is organized as follows. The following section summarizes the most important and relative research papers in the field. Section 3 describes the

bucket-assisted algorithm. Section 4 discusses the parallel approaches to sample entropy computation based on the bucket-assisted algorithm. In Section 5, execution times for both sequential and parallel variants of bucket-assisted algorithm, as well as speedups, are presented. After the discussion (Section 6), the last section (Section 7) concludes this work.

2. Similar Work

Traveling back to the past, Grassberger in 1990 [17] proposed a fast algorithm for fractal dimension estimation, in which the distance of neighboring points were checked in an m-dimensional space by separating the m-space into orthogonal subspaces and mapping m-dimensional points onto these subspaces, limiting in this way the number of performed comparisons. The amount of necessary memory was the drawback of this algorithm. Grassberger also compared his algorithm with a previously published one, which used kd-trees for the same purpose [18], concluding that his algorithm was faster.

Papers examining efficient algorithms for the computation of sample entropy include [19–21]. In [19,20], range search on kd-trees is used to exclude unnecessary comparisons. Even though the algorithmic complexity is reduced, the algorithm remains expensive, at least for typical inputs. This is due to the overheads and to the large number of operations necessary to locate and retrieve data for each comparison from the tree structure. We have shown in [13] that, in practice, the bucket-assisted algorithm is faster than kd-tree based algorithms. This is also confirmed by [22], which presents faster execution times for the lightweight algorithm compared to the kd-tree based ones. Fast algorithms for entropy computation are also presented in [21], although the values computed there are only approximations.

Parallel computation of sample entropy has garnered attention in the field, as evidenced by the investigation conducted in [22]. In their paper, the researchers proposed algorithms based on OpenCL, an open standard widely supported by a majority of graphics processing units (GPUs) and operating systems. The aim was to enhance the efficiency of sample entropy calculations, particularly for data sets with significant sizes. Remarkably, the results presented showcased a remarkable reduction in execution time, reaching a speedup of approximately 1/75th compared to the base algorithm, specifically for signal lengths exceeding 60,000 points. This achievement was accomplished through the parallelization of the computation process, leveraging both the serial kd-tree based algorithm and a lightweight variant. Notably, the authors did not propose a parallelization scheme for the bucket-assisted algorithm, which is widely acknowledged as the fastest serial algorithm in general cases.

Building upon [22], we delve into further exploration of parallelization opportunities for the bucket-assisted algorithm. By investigating potential strategies for parallelizing this algorithm, we aim to unlock its inherent speed and scalability, thereby enabling faster computation of sample entropy for large-scale data sets. The investigation into parallelization schemes for the bucket-assisted algorithm is a significant contribution to the field, as it addresses a crucial aspect that was not covered in the previous works.

3. The Bucket-Assisted Algorithm

The bucket-assisted algorithm was initially proposed for approximate entropy in [14], and adapted for sample entropy in [13]. We will give a short description, before studying the multithreaded approaches.

The key point in the algorithm is the early exclusion of those similarity checks, which we know a priori are going to fail. Two vectors $\vec{x}_i, \vec{x}_j$ of size m are similar when $||\vec{x}_i - \vec{x}_j||_m \leq r$, where r is the threshold distance and $||.||_m$ the infinity norm in m-dimensional spaces. These vectors can not be similar if the sum of their m elements differ more than mr, since they will differ more than r in at least one dimension. This observation leads to a remarkable reduction of the number of necessary comparisons. The description of the algorithm follows.

After embedding the time series into the m-dimensional space, we compute what we call the *integrated time series* (or *integrated signal*) X. Each point X_i of the integrated series X is computed by the sum of the elements of the corresponding vector in the m-dimensional space. As a result of the previous observation, two m-dimensional vectors cannot be similar if their corresponding points in the integrated time series differ more than mr. Please see [14] for a mathematical proof.

We place the points of the integrated time series into a number of N_b buckets of size r. The point X_i goes to the bucket $\lceil X_i/r \rceil$. All points in a bucket must be checked for similarity with all points in the same bucket or in one of the previous m buckets. As an example, please see Figure 1, where an integrated signal is depicted. Points between the solid lines B and C, i.e., in the bucket BC, can be similar only to the points between the dashed lines A and D since all other points differ with the points between lines B and C more than mr. Taking a closer look, one can notice that points between lines B and C should be checked for similarity only with points between the lines A and B; otherwise, each pair of points located in different buckets would have been checked twice, once as the pair $\vec{x}_i, \vec{x}_j$ and once again as the pair $\vec{x}_j, \vec{x}_i$.

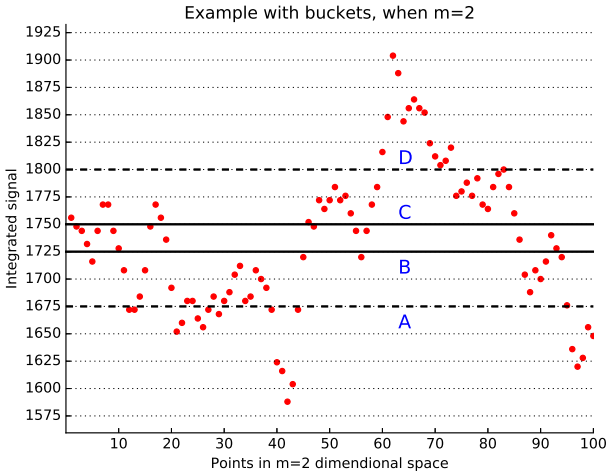

Figure 1. Example of the bucket-assisted algorithm. An example of an integrated signal is depicted. The value of m is 2 and the distance between the lines is r. Points between the solid lines B and C can be similar only to points between the dashed lines A and D. Similarity checks for points between lines B and C should be limited only to points between lines A and C to avoid double checking. Figure from [13].

In the above description, the size of each bucket has been considered as r. However, there is no evidence that r is the optimal value for the size for the buckets. Thus, the bucket-assisted algorithm introduces a new parameter, r_{split}, an integer value, which indicates into how many buckets each bucket of size r will be split. The size of each new bucket is r/r_{split}. The load balancing between the number of the points in a bucket and the total number of buckets can be determined only experimentally, and typical values are small integers ($r_{split} < 10$).

In order to further reduce the number of comparisons, we can exploit binary search, which is the fastest way to locate an element in a sorted series. We sort the vectors in each bucket according to their first element. During similarity check of the vector $\vec{x}_i$ with the vectors in the bucket b, we can exclude from comparison all vectors in b with their first elements not belonging in the interval $[x_i - r, x_i + r]$. Due to the $O(log(n))$ complexity, the massive exclusion of a large number of points in b in rapid.

Finally, similarity checks in $(m+1)$-dimensional space is performed only when the corresponding similarity check is successful in m-dimensional space. In this case, we only check the last elements of the vectors, since the first m have been already checked in the

m-dimensional space. For a more detailed description of the bucket-assisted algorithm please refer to [13].

In the following section, we will investigate parallel approaches to the bucket-assisted algorithm in an attempt to detect the best parallelization policy.

4. Parallel Approaches

In order to investigate parallel approaches for the bucket-assisted algorithm, we examined opportunities for parallelization on a simple multithreaded home computer processor. A number of threads share the same address space. Each thread works on a different subset of points and is responsible for the similarity checks for those points. The similarity check is the computationally intensive part of the method, where parallelization emerges. The first part of the algorithm, i.e, the computation of the integrated signal and the distribution of the points in the buckets, is not expensive. We chose to execute this part sequentially, even though some parallelism could also be extracted here.

In the parallel part of the algorithm, each thread computes the local values for similarity counters A and B. After all threads complete their tasks, all local values of A and B are summed up, yielding the total number or similar points for the whole time series. Three parallelization policies were considered, each expecting to work better in different conditions.

- *Static load balancing*: In Static load balancing, buckets are cyclically assigned to the available threads. It is a low overhead policy and can lead to optimal speedup when the number of points are evenly distributed in the buckets or, at least, if neighboring buckets have almost equal numbers of points. Since we expect the number of points in neighboring buckets not to differ significantly, we investigated if we could gain from the low overhead of the policy.
- *Dynamic load balancing*: In the case in which static load balancing fails to achieve a satisfactory distribution of the work, dynamic load balancing could offer better parallelization opportunities. Each thread is assigned one bucket at a time. When the thread completes the similarity checks for this bucket, a new bucket is assigned to the thread until there are no more buckets left unassigned. The additional effort to distribute the load and synchronize the tasks increases the overhead, but it is expected this overhead not to be significant.
- *In-bucket-based balancing*: When points are not well balanced in the buckets, both previous approaches might be proved insufficient, especially the static balancing. We investigated an additional policy, in which points in each bucket are equally distributed to the number of the available threads, i.e., every thread works on every bucket and is assigned a part of the points in this bucket. This approach seems also promising, since theoretically it achieves the optimal load balancing, with each thread assigned almost equal number of points. However, the compiler technology avails large loops and splitting large loops into smaller ones might lead to lower performance.

Which policy is better, and when, for the examined problem will be decided by the experimental analysis. In the next section, we will present and compare execution times and speedups from all the above policies.

5. Experimental Results

In order to compare the examined paralelization policies, we collected and compared execution times from parallel versions of the bucket-assisted algorithm, each one implementing a different paralelization policy. We performed the experiments on an Intel® Core™ i5-10400 CPU @ 2.90 GHz, with 6 threads and capability for 6 more virtual ones, and 7.6 GB of system memory. We developed the code in the C language and compiled it with gcc (v11.3) [23], in a Linux (Ubuntu 20.04.6 LTS) environment, with the standard optimization parameter *-O2*. POSIX threads (p-threads) [24] used as the parallelization infrastructure.

As input we used (a) a randomly generated time series of 100,000 samples with values uniformly distributed between 0 and 1 and (b) a data set of 54 Holter recordings from healthy people in sinus rhythm, publicly available on the internet [25] (each recording is approximately 24 h long, corresponding to approximately 100,000 beats). For the computation of sample entropy, we used the typical parameters $m = 2$, $r = 0.2$. No preprocessing (filtering) was applied to the Holter recordings.

Average times over 1000 executions for the random signals and over 100 times for the Holter recordings are summarized in Tables 1 and 2. In Table 1 times are per signal, while in Table 2 for the whole data set (54 recordings). Each table is separated into two parts. On the top part, the execution times of the two sequential algorithms are shown. The first one is a straightforward implementation based on the definition of the method. All possible pairs of vectors are checked, employing a loop with three nested levels. We give the exact structure of the loop here for clarity and in order to be used by researchers as a basis for future comparisons. See Algorithm 1.

Algorithm 1: Straightforward implementation.

```
1       for i=1 ... N−m:
2           for j=i+1 ... N−m:
3               for k=1 ... m:
4                   if abs(x_{i+k}−x_{j+k} > r):
5                       break;
6               if k==m:
7                   B++
8                   if abs(x_{i+m}−x_{j+m}) ≤ r:
9                       A++
```

The second algorithm is the bucket-assisted algorithm, as described earlier in Section 3 and analytically in [13].

Table 1. Execution times to compute sample entropy ($m = 2, r = 0.2$) for randomly generated signals.

Serial executions			
Straightforward algorithm: 9.153 s *Bucket-assisted algorithm*: 0.546 s			
Parallel executions			
	Static-load balancing	*In-bucket balancing*	*Dynamic-load balancing*
2 threads:	0.291 s	0.330 s	0.295 s
4 threads:	0.144 s	0.192 s	0.157 s
6 threads:	0.099 s	0.150 s	0.103 s
12 threads:	0.082 s	0.145 s	0.081 s

Table 2. Execution times to compute sample entropy ($m = 2, r = 0.2$) for 54 Holter (24 h) recordings.

Serial executions			
Straightforward algorithm: 517.58 s *Bucket-assisted algorithm*: 68.56 s			
Parallel executions			
	Static-load balancing	*In-bucket balancing*	*Dynamic-load balancing*
2 threads:	44.17 s	41.25 s	35.57 s
4 threads:	20.03 s	25.47 s	19.32 s
6 threads:	16.43 s	17.75 s	13.37 s
12 threads:	14.10 s	19.72 s	12.42 s

Figure 2 depicts the speedups achieved for each parallel execution compared to the execution time of the bucket-assisted algorithm. The panel on the left reports speedups for the randomly generated time series, while the one on the right shows speedups for the 54 Holter (24 h) recordings.

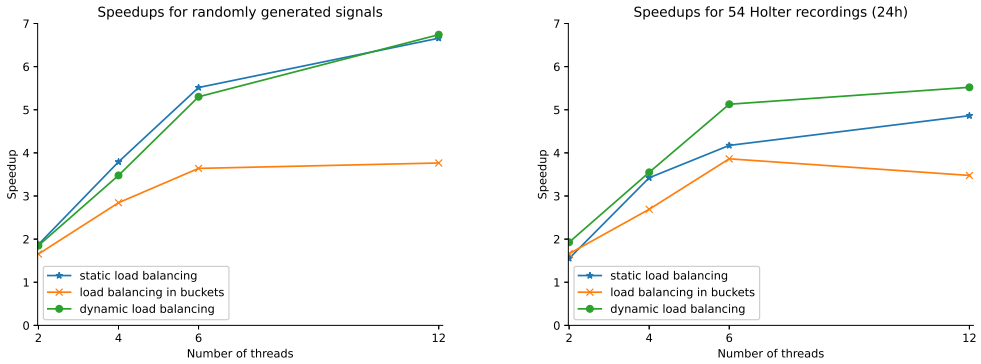

Figure 2. Left panel: speedups for randomly generated time series. **Right panel**: speedups for a set of 54 Holter recordings in sinus rhythm.

In the following section, we will further discuss the findings of this research and examine the factors that influence the performance of the sample entropy computation algorithms.

6. Discussion

The efficiency of the computation of sample entropy depends on several factors. The most important one is the size of the time series since the complexity of the problem is relatively high: $O(n^2)$. For a short recording, the computational time is not significant, but for a larger signal, for example, for a 24 h Holter recording, this time is not an amount which could be easily ignored, especially when more than one signal is to be processed or more than single values of m and r are to be investigated for each recording. Even though the typical values $m = 2$, $r = 0.2$ are almost always used, investigation of different values for those parameters requires additional executions.

The values of the parameters m and r both affect the execution time. The larger the size of the embedding space m, the larger the size of the vectors checked for similarity in each step. This is, however, a linear dependency. If we select to reflect the influence of m in the computational complexity, the complexity of the algorithm becomes $O(mn^2)$, with $m \ll n$. Execution time also depends on the threshold distance r. Since, according to the definition of sample entropy, the maximum (infinity) norm, i.e., $||x||_\infty = max(|x_1|, \ldots, |x_n|)$, is used for the similarity checks, the similarity can sometimes be decided early, even during the comparison of the first elements of the two vectors. In this case, the comparisons of the rest of the elements of the two vectors are not necessary (see lines 3 and 4 of the straightforward implementation in Section 5). The smaller the value of r, the larger the probability that a similarity check can be decided early and the smaller the average amount of time necessary for a similarity check.

Faster algorithms can be achieved in two ways. The first one is to design algorithms which compute an approximation of the final result. Especially in approximate and sample entropy computation, which by definition are estimations of the entropy of the system, this approximation does not necessarily significantly affect the discriminating power of each method. However, such an approach should be considered as a limitation from the algorithmic point of view. Such limitation does not exist for the bucket-assisted algorithm since it computes the exact value of sample entropy, returning exactly the same result of the straightforward implementation and according to the definition. Thus, no remarkable limitations of the bucket-assisted algorithm should be mentioned.

As a drawback, the complexity of the implementation can be considered. Not the algorithmic one, which is still $O(n^2)$, but the development effort and the complicated data structures and nested loops required for the implementation. Even though this is not a problem per se, recent compiler technology targets simple structures and uniform loops for speeding up applications and automatic extraction of parallelism. As we claimed in [13], the bucket-assisted algorithm is much faster than other similar algorithms. However, due to the implementation complexity, the bucket-assisted algorithm cannot expect to gain as much as the simpler implementations from future achievements in compiler and automatic code parallelization technology.

In this work, we proposed parallel implementations for the bucket-assisted algorithm, tested on common cost-effective home processors, employing multithreading technology. We performed experiments with (a) randomly generated time series and (b) 54 Holter recordings, approximately 24 h long each.

We first compared the performance of the bucket-assisted algorithm compared to a straightforward implementation based on the definition of sample entropy. The purpose of this comparison was: (a) to show the efficiency of bucket-assisted algorithm to efficiently compute the value of sample entropy and (b) to give the opportunity to researchers for future comparisons, by providing execution times and pseudo-code for this implementation. For randomly generated signals, the straightforward implementation required on average 9.153 s to complete the computation, while the bucket-assisted algorithm only 0.546 s, i.e., 16.76 times faster. For the set of 54 Holter recordings, the corresponding times were 517.58 s for the straightforward implementation and 68.56 s for the bucket-assisted algorithm, yielding a ratio of 7.55. Parallel implementations further reduced the execution times. For 12 threads, (6 plus 6 virtual) the average computation time for a random signal was reduced to 0.081 s, while for the 54 Holter recordings, it was reduced to 12.42 s.

Three different parallelization policies were tested. Execution times are depicted in detail in Tables 1 and 2, while speedups are shown in Figure 2. The two data sets present different behaviors. In the first one, points are uniformly distributed, leading to uniform distribution of points in the buckets. In the second data set, a similar load balancing is more difficult to reach, since the number of points in the buckets may differ significantly. This in turn affected the parallelization rates, as it can be easily noticed in Figure 2, where speedups for the random time series are significantly higher than those for the Holter recordings.

Comparing the three examined parallelization policies, one can easily notice that the static policy behaves better in the random time series. This is due to the uniform distribution of points in the buckets as well as the low management and synchronization overhead. For the Holter recordings, where the distribution of the points in the buckets is not uniform, the dynamic load balancing policy becomes better.

The *in-bucket* load balancing is always worse than the two other policies, but the difference between the *in-bucket* and the other two policies is significantly smaller for the random time series. This is, again, due to the uniform distribution of points in the buckets. This policy theoretically gives the optimal distribution, since each thread is assigned almost exactly the same number of points. However, the additional management overhead and the reduction of the size of the loops lead to additional overhead. However, for large and relatively uniform time series, this policy might produce better parallelization rates than the other two.

In conclusion, the dynamic load balancing policy seems to be the most efficient one in the general case, even though there are cases in which the two other policies may result in lower execution times.

7. Conclusions

The main purpose of this paper was to investigate how the bucket-assisted algorithm can be parallelized to further speed up the execution of sample entropy. Even though the algorithm is parallel, it can be executed in any modern processor, since parallelization is based on the multithreading technology. A secondary aim of the paper was to report

updated execution times and speedups for the bucket-assisted algorithm, as affected by the recent advances on compiler and hardware technologies. In addition, a multithreaded Python library, wrapped on top of fast implementations in C, is also made available. Three parallelization policies were examined for the bucket-assisted algorithm, each of which was tested on signals with different sample distributions. The dynamic load balancing, however, proved to be the best one in the general case. The parallelization rates reported for the parallel version of bucket-assisted algorithm are approaching, with 6 threads on a cost-effective processor, the theoretically optimal speedup.

Author Contributions: Conceptualization, G.M. and R.S.; software, D.B.; writing—original draft preparation, D.B.; writing—review and editing, G.M. and R.S.; supervision, G.M. and R.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hager, G.; Wellein, G. *Introduction to High Performance Computing for Scientists and Engineers*; CRC Press, Inc.: Boca Raton, FL, USA, 2010.
2. Sterling, T.; Anderson, M.; Brodowicz, M. *High Performance Computing: Modern Systems and Practices*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2017.
3. Richman, J.S.; Moorman, J.R. Physiological time-series analysis using approximate entropy and sample entropy. *Am. J. Physiol.-Heart Circ. Physiol.* **2000**, *278*, H2039–H2049. [CrossRef] [PubMed]
4. Lake, D.; Richman, J.; Griffin, M.; Moorman, J. Sample entropy analysis of neonatal heart rate variability. *Am. J. Physiol. Regul. Integr. Comp. Physiol.* **2002**, *283*, R789–R797. [CrossRef] [PubMed]
5. Pincus, S.M. Approximate entropy as a measure of system complexity. *Proc. Natl. Acad. Sci. USA* **1991**, *88*, 2297–2301. [CrossRef] [PubMed]
6. Al-Angari, H.; Sahakian, A. Use of Sample Entropy Approach to Study Heart Rate Variability in Obstructive Sleep Apnea Syndrome. *IEEE Trans. Bio-Med. Eng.* **2007**, *54*, 1900–1904. [CrossRef] [PubMed]
7. Goya-Esteban, R.; Marques de Sa, J.; Rojo-Alvarez, J.; Barquero-Perez, O. Characterization of Heart Rate Variability loss with aging and heart failure using Sample Entropy. In Proceedings of the 2008 Computers in Cardiology, Bologna, Italy, 14–17 September 2008; pp. 41–44. [CrossRef]
8. Song, Y.; Crowcroft, J.; Zhang, J. Automatic epileptic seizure detection in EEGs based on optimized sample entropy and extreme learning machine. *J. Neurosci. Methods* **2012**, *210*, 132–146. [CrossRef] [PubMed]
9. Alcaraz, R.; Rieta, J.J. Sample entropy of the main atrial wave predicts spontaneous termination of paroxysmal atrial fibrillation. *Med. Eng. Phys.* **2009**, *31*, 917–922. [CrossRef] [PubMed]
10. Ramdani, S.; Seigle, B.; Lagarde, J.; Bouchara, F.; Bernard, P.L. On the use of sample entropy to analyze human postural sway data. *Med. Eng. Phys.* **2009**, *31*, 1023–1031. [CrossRef] [PubMed]
11. Olbrys, J.; Majewska, E. Approximate entropy and sample entropy algorithms in financial time series analyses. *Procedia Comput. Sci.* **2022**, *207*, 255–264. [CrossRef]
12. Guzmán-Vargas, L.; Ramírez-Rojas, A.; Hernández-Pérez, R.; Angulo-Brown, F. Correlations and variability in electrical signals related to earthquake activity. *Phys. A Stat. Mech. Its Appl.* **2009**, *388*, 4218–4228. [CrossRef]
13. Manis, G.; Aktaruzzaman, M.; Sassi, R. Low computational cost for sample entropy. *Entropy* **2018**, *20*, 61. [CrossRef] [PubMed]
14. Manis, G. Fast computation of approximate entropy. *Comput. Methods Programs Biomed.* **2008**, *91*, 48–54. [CrossRef] [PubMed]
15. Manis, G.; Aktaruzzaman, M.; Sassi, R. Bubble entropy: An entropy almost free of parameters. *IEEE Trans. Biomed. Eng.* **2017**, *64*, 17259870. [CrossRef]
16. Bandt, C.; Pompe, B. Permutation entropy: A natural complexity measure for time series. *Phys. Rev. Lett.* **2002**, *88*, 174102. [CrossRef] [PubMed]
17. Grassberger, P. An optimized box-assisted algorithm for fractal dimensions. *Phys. Lett. A* **1990**, *148*, 63–68. [CrossRef]
18. Bingham, S.; Kot, M. Multidimensional trees, range searching, and a correlation dimension algorithm of reduced complexity. *Phys. Lett. A* **1989**, *140*, 327–330. [CrossRef]
19. Pan, Y.H.; Wang, Y.H.; Liang, S.F.; Lee, K.T. Fast computation of sample entropy and approximate entropy in biomedicine. *Comput. Methods Programs Biomed.* **2011**, *104*, 382–396. [CrossRef] [PubMed]
20. Jiang, Y.; Mao, D.; Xu, Y. A fast algorithm for computing sample entropy. *Adv. Adapt. Data Anal.* **2011**, *3*, 167–186. [CrossRef]
21. Tomčala, J. New Fast ApEn and SampEn entropy algorithms implementation and their application to supercomputer power consumption. *Entropy* **2020**, *22*, 863. [CrossRef] [PubMed]
22. Dong, X.; Chen, C.; Geng, Q.; Zhang, W.; Zhang, X.D. Fast algorithm based on parallel computing for sample entropy calculation. *IEEE Access* **2021**, *9*, 20223–20234. [CrossRef]

23. Stallman, R.M.; The GCC Developer Community. GNU Compiler Collection (GCC) Internals. Available online: https://gcc.gnu.org/onlinedocs/gcc-11.1.0/gcc/ (accessed on 13 June 2023).
24. Lewis, B.; Berg, D.J.; Cunningham, R. *POSIX Threads Programming*; Addison-Wesley Professional: Boston, MA, USA, 1996.
25. Goldberger, A.L.; Amaral, L.A.N.; Glass, L.; Hausdorff, J.M.; Ivanov, P.C.; Mark, R.G.; Mietus, J.E.; Moody, G.B.; Peng, C.K.; Stanley, H.E. PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation* **2000**, *101*, e215–e220. [CrossRef] [PubMed]