Procedia Computer Science

Volume 66, 2015, Pages 63–72

YSC 2015. 4th International Young Scientists Conference on Computational Science

Parallel Algorithm for Local-best-match Time Series Subsequence Similarity Search on the Intel MIC Architecture

Aleksander V. Movchan and Mikhail L. Zymbler

South Ural State University, Chelyabinsk, Russia
movchanav@susu.ru, mzym@susu.ru

Abstract

The paper touches upon the problem of local-best-match time series subsequence similarity search. The problem assumes that a query sequence and a longer time series are given, and the task is to find all the subsequences whose distance from the query is the minimal among their neighboring subsequences and distance from the query is under specified threshold. The Dynamic Time Warping (DTW) is used as a distance metric, which currently is recognized as the best similarity measure for most time series applications. However, computation of DTW costs too much despite the existing sophisticated software approaches. Existing hardware approaches to DTW computation involve GPU and FPGA and pay no regard to the Intel Many Integrated Core architecture. The paper proposes a parallel algorithm for solving this problem using both CPU and the Intel Xeon Phi many-core coprocessor. The implementation is based on the OpenMP parallel programming technology and offload execution mode, where part of the code and data is transmitted to the coprocessor. The algorithm utilizes a queue of subsequences on the processor side, which are uploaded to the coprocessor for the DTW computations. The results of experiments confirm the effectiveness of the algorithm.

Keywords: time series data mining, subsequence similarity, local-best-match search, Dynamic Time Warping, OpenMP, Intel Many Integrated Core architecture, Intel Xeon Phi coprocessor

1 Introduction

Subsequence similarity search is one of the topical issues of time series data mining in many applications, e.g. weather forecasting [1], finance analytics [2], medical monitoring [3], etc. Local-best-match search assumes that a query sequence and a longer time series are given, and the task is to find all the subsequences whose distance from the query is the minimal among their neighboring subsequences and distance from the query is under specified threshold.

Nowadays the Dynamic Time Warping (DTW) [4] is the most popular similarity measure in many applications [5]. DTW is computationally expensive and there are many approaches

Selection and peer-review under responsibility of the Scientific Programme Committee of YSC 2015

doi:10.1016/j.procs.2015.11.009

that have been proposed to accelerate it, e.g. lower bounding [5], computation reusing [6], data indexing [7], early abandoning [8], etc. However, DTW still takes a large part of the total application runtime. That is why there are efforts to accelerate subsequence similarity search using parallel hardware, e.g. computer-cluster [9], multi-core [10], FPGA and GPU [6, 11, 12].

This paper presents a parallel algorithm for local-best-match subsequence search based on DTW distance for a central processor unit (CPU) accompanied with the Intel Xeon Phi many-core coprocessor [13]. The rest of the paper is organized as follows. Section 2 contains formal definition of the problem, briefly describes Intel Xeon Phi architecture and programming model and discusses related work. The suggested algorithm is described in section 3. The results of experimental evaluation of the algorithm are presented in section 4. Section 5 contains concluding remarks and directions for future research.

2 Background and Related Work

2.1 Formal Definitions

A *time series* T is an ordered sequence $t_1, t_2, \ldots, t_N$ of real data points, measured chronologically, where N is a length of the sequence.

Dynamic Time Warping (DTW) is a similarity measure between two time series X and Y, where $X = x_1, x_2, ..., x_N$ and $Y = y_1, y_2, ..., y_N$, is defined as follows.

$$DTW(X, Y) = d(N, N), \text{ where}$$

$$d(i, j) = |x_i - y_j| + min \begin{cases} d(i-1, j) \\ d(i, j-1) \\ d(i-1, j-1), \end{cases}$$

$$d(0, 0) = 0; d(i, 0) = d(0, j) = \infty;\ i = j = 1, 2, \ldots, N.$$

A *subsequence* $T_{i\,m}$ of time series T is its continuous subset starting from i-th position and consisting of m data points, i.e. $T_{i\,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$, where $1 \leq i \leq N$ and $i + m \leq N$.

A *query* Q is a certain subsequence to be found in T. Let n is a length of the query, $n \ll N$.

Local-best-match subsequence search [11] is defined as follows. Let $\mathcal{E} > 0$ is a similarity threshold and $\mathbb{L}$ denotes the resulting set of subsequences. Then $T_{i\,m} \in \mathbb{L} \Leftrightarrow T_{i\,m}$ satisfies the following conditions:

1. $m = n$;
2. $DTW(T_{i\,m}, Q) < \mathcal{E}$;
3. $i = \underset{j \in \{i-1, i, i+1\}}{\operatorname{argmin}} DTW(T_{j\,m}, Q)$.

2.2 The Intel Xeon Phi Architecture and Programming Model

The Intel Xeon Phi coprocessor is an x86 many-core coprocessor of 61 cores, connected by a high-performance on-die bidirectional interconnect where each core supports 4× hyperthreading and contains 512-bit wide vector processor unit (VPU). Each core has two levels of cache memory: a 32 Kb L1 data cache, a 32 Kb L1 instruction cache, and a core-private 512 Kb unified L2 cache. The Intel Xeon Phi coprocessor is to be connected to a host computer via a PCI Express system interface. PCI Express is used for data transfer between CPU and the coprocessor.

The Intel Xeon Phi coprocessor supports the same programming tools and models as a regular Intel Xeon processor because of its Intel x86 native architecture. The Intel Xeon Phi coprocessor supports three programming modes: native, offload and symmetric. In native mode the application runs independently, on the coprocessor only. In offload mode the application is running on the host and offloads computationally intensive part of work to the coprocessor. The symmetric mode allows the coprocessor to communicate with other devices by means of Message Passing Interface (MPI).

2.3 Related Work

Currently DTW is considered as best similarity measure for many applications [5], despite the fact that it is very time-consuming [14, 9]. Research devoted to acceleration of DTW computation includes the following.

The SPRING algorithm [15] uses computation-reuse technique. However, this technique squeezes the algorithm's applications because data-reuse supposes non-normalized sequence. In [7] indexing technique to speed up the search was used, which need to specify the query length in advance. Authors of [16] suggested multiple indices for various length queries. Lower bounding [17] allows one to discard unpromising subsequences using the lower bound of DTW distance estimated in a cheap way. The UCR-DTW algorithm [8] integrates all the possible existing speedup techniques and most likely it is the fastest of the existing subsequence matching algorithms.

All the aforementioned algorithms aim to decrease the number of calls of DTW subroutine, not accelerating DTW itself. However, because of its complexity, DTW still takes a large part of the total application runtime [12]. There are approaches exploiting the allocation of DTW computation of different subsequences into different processing elements.

In [10] subsequences starting from different positions of the time series are sent to different Intel Xeon processors, and each processor computes DTW. In [9] different queries are distributed onto different cores, and each subsequence is sent to different cores to be compared with different queries. GPU implementation [12] parallelize the generation of the warping matrix but still process the path search serially. GPU implementation proposed in [6] utilizes the same ideas as in [10]. FPGA implementation described in [6] focuses on the naive subsequence similarity search, and do not exploit any pre-processing techniques. It is generated by a C-to-VHDL tool and should be recompiled if length of query is changed. This algorithm supports 8-bit data precision and can not support queries longer than 2^{10}, so it can not be applied in big-scale tasks. To address these problems in [11] a stream oriented framework was proposed. It implements coarse-grained parallelism by reusing data of different DTW computations and uses a two-phase precision reduction technique to guarantee accuracy while reducing resource cost.

This paper suggests a parallel algorithm of the local-best-match time series subsequence similarity search under DTW for the Intel Many-integrated core accelerators where the UCR-DTW serial algorithm is used as a basis. Parallelization of the original algorithm was performed by means of OpenMP technology and adapted for the Intel Xeon Phi many-core coprocessor using our previous research [18].

3 Local-best-match Search on the Intel Xeon Phi

Development of the parallel algorithm for local-best-match subsequence search looks like the following. Firstly (3.1), we have implemented local-best-match serial algorithm using UCR-DTW serial algorithm [8]. Next (3.2), we have created a parallel version of our serial algorithm

by means of the OpenMP technology. Finally (3.3), we have adapted the algorithm developed at the previous step for the Intel Xeon Phi many-core coprocessor.

3.1 Serial Algorithm

Our algorithm for local-best-match subsequence search is depicted in Fig. 1 (hereinafter lbm-UCR-DTW). The original algorithm is represented by the `Lower Bounding` subactivity, where a cascade of three lower bounding of DTW distance is used, namely LB_{Kim} [8], LB_{Keogh} [14] and $LB_{KeoghEC}$ [8].

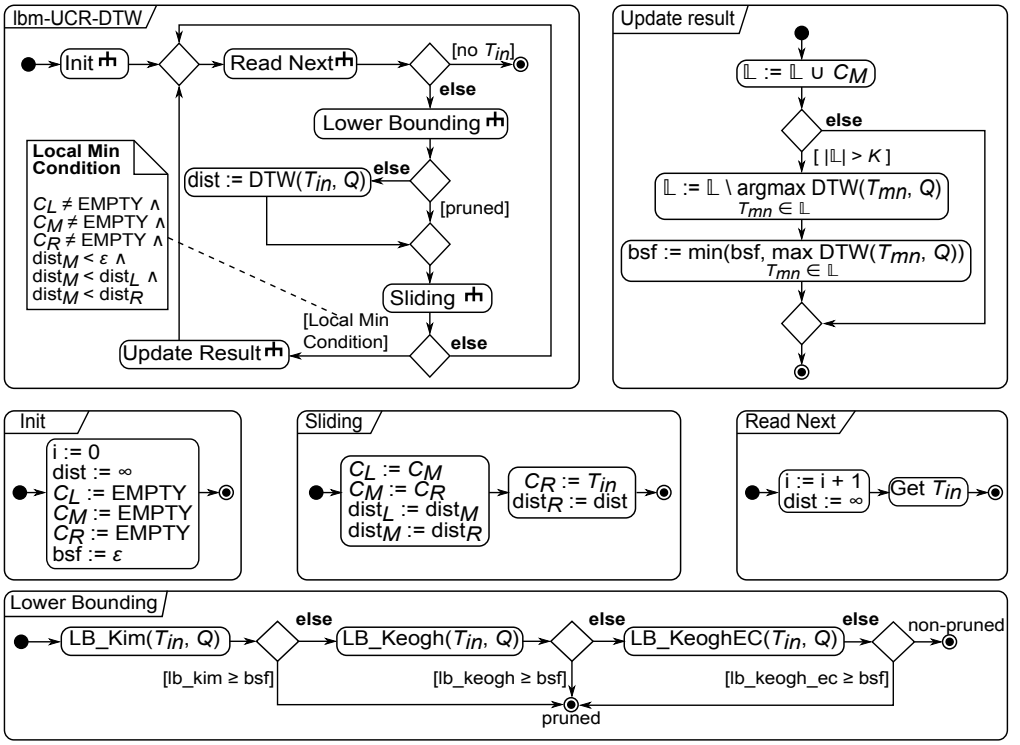


Figure 1: Serial algorithm

We assume that $|\mathbb{L}| \leq K$, i.e. resulting set contains no more than K subsequences, where K is specified threshold. This restriction is practically useful due to possible memory limitations to store resulting subsequences and does not lose generality because we can consider the case of $K = \infty$.

The `bsf` (best-so-far) variable stores the distance to the most distant subsequence among all the similar subsequences and is calculated as follows:

$$bsf = \begin{cases} \mathcal{E} & , |\mathbb{L}| < K \\ \min(\mathcal{E}, \max\limits_{T_{m\,n} \in \mathbb{L}} DTW(T_{m\,n}, Q)) & , else. \end{cases}$$

The algorithm scans every triple of neighboring subsequences where the current processed subsequence $T_{i\,n}$ is denoted as C_R and two previously processed subsequences $T_{i-1\,n}$ and $T_{i-2\,n}$ are denoted as C_M and C_L respectively. The distances from the C_R, C_M and C_L to the query

are denoted as $dist_R$, $dist_M$ and $dist_L$ respectively. The `Sliding` subactivity updates these variables during processing.

If C_M subsequence represents a local minimum, then the `Update Result` subactivity includes C_M subsequence in $\mathbb{L}$ resulting set. If cardinality of the $\mathbb{L}$ set is greater than K then algorithm excludes from $\mathbb{L}$ a subsequence with greatest distance to the query and after that updates the `bsf` variable according to aforementioned formula. The algorithm stops after all the subsequences have been processed.

3.2 Parallel Algorithm for CPU

Fig. 2 depicts a parallel version of the lbm-UCR-DTW algorithm. Parallelization was performed through the OpenMP technology.

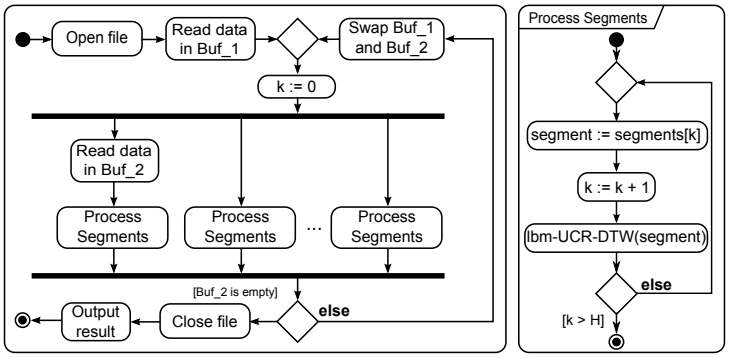


Figure 2: Parallel algorithm for CPU

The input time series T is partitioned into H equal-length segments. Let P denotes the number of OpenMP-threads, S denotes a maximum length of segment, then H is defined as

$$H = \lceil \frac{N}{P \cdot S} \rceil \cdot P$$

The number of segments H is divisible by the number of threads P for better load balancing. A k-th segment, $0 \leq k \leq H-1$, is defined as a subsequence $T_{s\,l}$, where

$$s = \begin{cases} 1 & , k = 0 \\ k \cdot \lfloor \frac{N}{H} \rfloor - n + 2 & , else \end{cases}$$

$$l = \begin{cases} \lfloor \frac{N}{H} \rfloor & , k = 0 \\ \lfloor \frac{N}{H} \rfloor + n - 1 + (N \ mod \ H) & , k = H-1 \\ \lfloor \frac{N}{H} \rfloor + n - 1 & , else \end{cases}$$

It means that the head part of every segment except first overlaps with the tail part of previous segment in $n-1$ data points, where n is length of the query. This prevents from the loss of possible resulting subsequences, which start at tail part of the previous segment.

The algorithm distributes segments across threads as follows. There is an integer `k` variable, which is shared among all threads and identifies first unprocessed segment. The `k` is initialized by 0 and while there are unprocessed segments (i.e. $k \leq H$), a thread gets k-th segment, increments `k` by 1 and processes the segment by means of the `lbm-UCR-DTW` subroutine, which implements the serial algorithm. To provide correct processing of shared data we use critical

section to prevent multiple threads from accessing the critical section's code at the same time, i.e. only one active thread can get k-th segment and update the `k` variable.

We use this way of distribution instead of the assigning of each thread its own segments before calculations because the latter could result in worse load balancing due to unpredictable amount of pruned and early abandoned subsequences for each thread. In other words, overhead costs to provide the critical section is a lesser evil than highly probable load imbalance.

In contrast with the serial version the `bsf` variable is shared among the threads. This allows each thread to prune off unpromising subsequence using lower bounding. Resulting set is shared among the threads as well and is write-protected by means of critical section. This allows to improve the value of the `bsf` variable (i.e. decrease it to prune more unpromising subsequences) faster than in case of using individual resulting subset for each thread.

3.3 Parallel Algorithm for the Intel Xeon Phi

The parallel algorithm for CPU and the Intel Xeon Phi is depicted in Fig. 3. The idea of the algorithm is that the coprocessor should be exploited only for DTW computations whereas CPU performs lower bounding, prepares subsequences for the coprocessor and computes DTW in case if it really does not have another job. CPU supports a queue of candidate subsequences and the coprocessor computes DTW for each candidate. Queue stores a tuple (i, A) corresponding a candidate subsequence $T_{i\,n}$, where A is an n-element array containing LB_{Keogh} lower bounds for each position of the subsequence which is used for early abandoning of DTW [8].

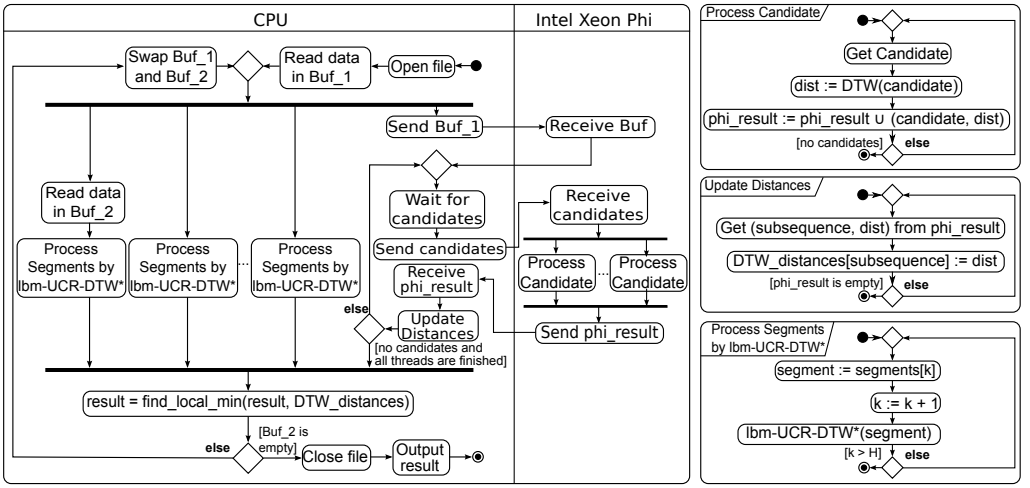


Figure 3: Parallel algorithm for CPU and the Intel Xeon Phi

To reduce the amount of data transferred to the coprocessor, CPU offloads current buffer of the time series once whereas queue is offloaded each time it is full. The number of elements in the queue is the algorithm's parameter and is calculated as $C \cdot h \cdot W$, where C is a number of cores of the coprocessor, h is a hyperthreading factor of the coprocessor and W is a number of candidates to be processed by a coprocessor's thread.

The algorithm could be described in the following way. One of the CPU threads is declared as a master and the rest as workers. At start master sends a buffer with the current segment of the time series to the coprocessor. If queue is full then master offloads it to the coprocessor to perform DTW computation for the corresponding subsequences by the coprocessor's threads.

Worker's activity is similar to activity of CPU threads in parallel algorithm described in 3.2. Each worker processes segments by means of the `lbm-UCR-DTW*` (see Fig. 4) subroutine, which calculates cascade of lower bounds for the subsequence. If the subsequence is dissimilar to the query then the worker prunes it off otherwise the subsequence is pushed to the queue. If the queue is full (and data previously transferred to the coprocessor have not been processed yet), the worker calculates DTW by itself. Worker stores results of DTW calculations in an array shared among all the workers.

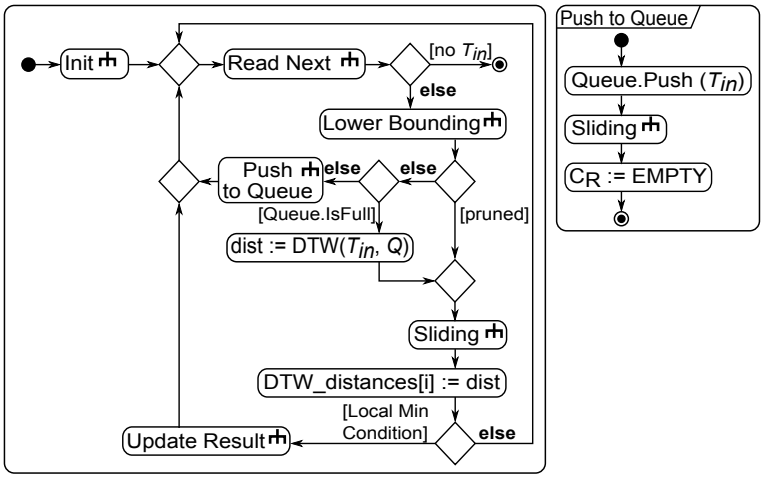


Figure 4: lbm-UCR-DTW* subroutine

After all the candidate subsequences are offloaded to the coprocessor DTW calculation is performed for every candidate subsequence and "index-distance" tuples are offloaded to CPU. Finally, algorithm searches local-best-match subsequences in the shared array filled earlier.

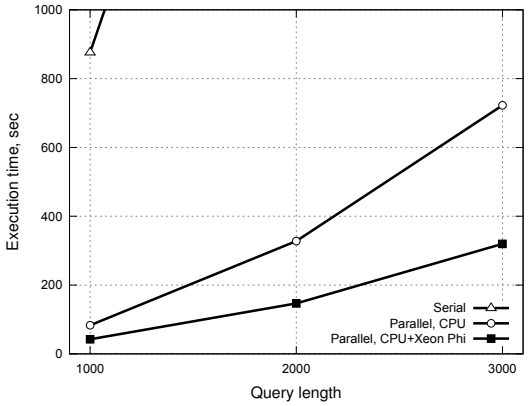
4 Experiments

Hardware platform of the experiments is described in Tab. 1.

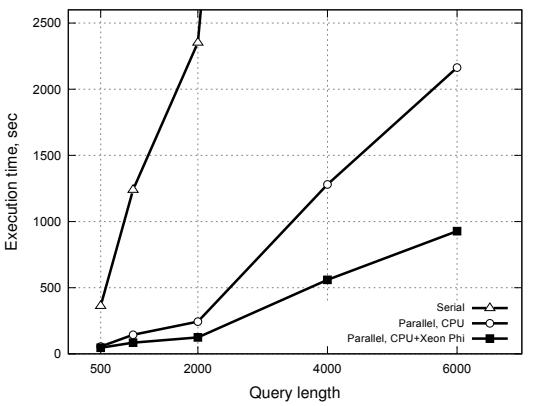
Table 1: Specifications of the hardware platform

Specifications	Processor	Coprocessor
Model	Intel Xeon X5680	Intel Xeon Phi SE10X
Cores	6	61
Frequency, GHz	3.33	1.1
Threads per core	2	4
Peak performance, TFLOPS	0.371	1.076
Memory, Gb	24	8

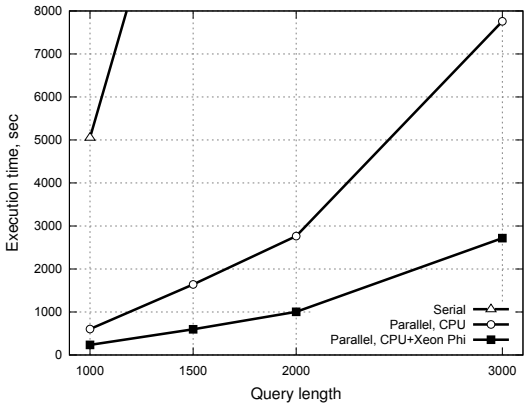
Experiments have been performed on three time series, namely PURE RANDOM, RANDOM WALK and ECG. The PURE RANDOM data set consists of 10^6 points generated by a random function. The RANDOM WALK data set is one-dimensional random walk time series consisting of 10^8 points. The ECG data set [8] consists of $2 \cdot 10^7$ points and represents approximately 22 hours of one electrocardiographic channel sampled at 250 Hz.



(a) PURE RANDOM data set



(b) RANDOM WALK data set



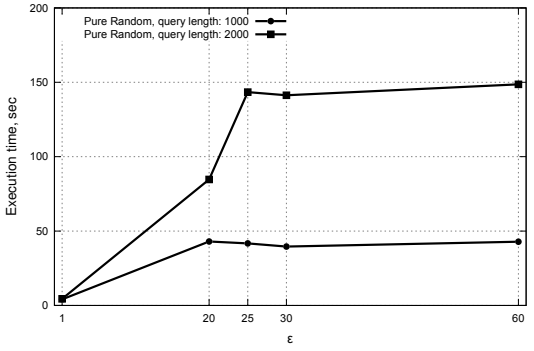
(c) ECG data set

Figure 5: Performance of the algorithm

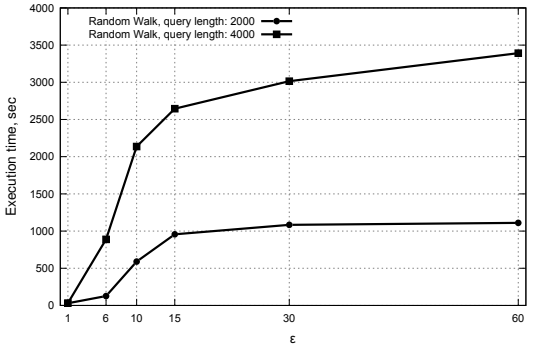
In the experiments we used $\mathcal{E} = 2 + D_{min}$ as a similarity threshold where D_{min} is a distance between a query and the most similar subsequence found beforehand using our algorithm [18]. As cardinality of the resulting set we used $K = 10^4$. We investigated performance of our algorithm varying query length and similarity threshold.

On the PURE RANDOM data set our algorithm shows (Fig. 5a) a two times higher performance than the parallel algorithm for CPU only. Experimental results on RANDOM WALK data set (Fig. 5b) show that our algorithm is more effective for longer queries. In case of shorter queries the algorithm has the same performance as parallel algorithm for CPU only. In the experiments on ECG data set our algorithm shows (Fig. 5c) almost three times higher performance than the parallel algorithm for CPU only.

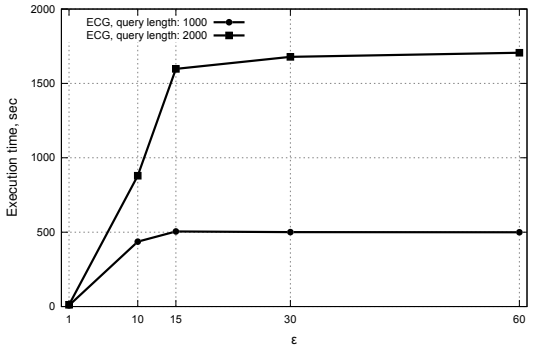
Impact of the $\mathcal{E}$ similarity threshold on the performance is depicted in Fig. 6. As expected algorithm's performance is inversely proportional to the similarity threshold. As we can see for each data set and length of query there exists a threshold t such that execution time stays almost constant for all values of threshold more than t. This behavior occurs because resulting set is updated with the same speed for all values of threshold more than t.



(a) PURE RANDOM data set



(b) RANDOM WALK data set



(c) ECG data set

Figure 6: Impact of similarity threshold on the performance of the algorithm

5 Conclusion

In this paper we have described the parallel algorithm for local-best-match time series subsequence search under DTW distance on the Intel Many Integrated Core architecture. The parallel algorithm combines capabilities of CPU and the Intel Xeon Phi coprocessor. The coprocessor is exploited only for DTW computations whereas CPU performs lower bounding, prepares subsequences for the coprocessor and computes DTW as a last resort. CPU supports a queue of candidate subsequences and the coprocessor computes DTW for every candidate. Experiments on synthetic and real data sets have shown that our algorithm outperforms serial algorithm and parallel algorithm that uses CPU only.

As future work we plan to extend our research for the case of cluster system based on nodes equipped with the Intel Xeon Phi coprocessors.

Acknowledgment

This work was financially supported by the Ministry of education and science of the Russian Federation (“Research and development on priority directions of scientific-technological complex of Russia for 2014–2020” Federal Program, contract No. 14.574.21.0035).

References

[1] Abdullaev, S., Lenskaya, O., Gayazova, A., Sobolev, D., Noskov, A., Ivanova, O., and Radchenko, G. *Bull. of South Ural State University. Series: Comput. Math. and Soft. Eng.* **3**(1), 17–32 (2014).

[2] Dyshaev, M. and Sokolinskaya, I. *Bull. of South Ural State University. Series: Comput. Math. and Soft. Eng.* **2**(4), 103–108 (2013).

[3] Epishev, V., Isaev, A., Miniakhmetov, R., Movchan, A., Smirnov, A., Sokolinsky, L., Zymbler, M., and Ehrlich, V. *Bull. of South Ural State University. Series: Comput. Math. and Soft. Eng.* **2**(1), 44–54 (2013).

[4] Berndt, D. J. and Clifford, J. In *KDD Workshop,* Fayyad, U. M. and Uthurusamy, R., editors, 359–370. AAAI Press, (1994).

[5] Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., and Keogh, E. J. *PVLDB* **1**(2), 1542–1552 (2008).

[6] Sart, D., Mueen, A., Najjar, W. A., Keogh, E. J., and Niennattrakul, V. In *ICDM,* Webb, G. I., Liu, B., Zhang, C., Gunopulos, D., and Wu, X., editors, 1001–1006. IEEE Computer Society, (2010).

[7] Lim, S., Park, H., and Kim, S. In *Database Systems for Advanced Applications, 11th International Conference, DASFAA 2006, Singapore, April 12-15, 2006, Proceedings,* Lee, M., Tan, K., and Wuwongse, V., editors, volume 3882 of *Lecture Notes in Computer Science*, 65–79. Springer, (2006).

[8] Rakthanmanon, T., Campana, B. J. L., Mueen, A., Batista, G. E. A. P. A., Westover, M. B., Zhu, Q., Zakaria, J., and Keogh, E. J. In *KDD,* Yang, Q., Agarwal, D., and Pei, J., editors, 262–270. ACM, (2012).

[9] Takahashi, N., Yoshihisa, T., Sakurai, Y., and Kanazawa, M. In *2009 International Conference on Complex, Intelligent and Software Intensive Systems, CISIS 2009, Fukuoka, Japan, March 16-19, 2009,* Barolli, L., Xhafa, F., and Hsu, H., editors, 1100–1105. IEEE Computer Society, (2009).

[10] Sharanyan, S., Arvind, K., and Rajeev, G. In *ICCCT,* of Computer Science, D. and Engineering, M. N. N. I. o. T., editors, 394–398. IEEE Computer Society, (2011).

[11] Wang, Z., Huang, S., Wang, L., Li, H., Wang, Y., and Yang, H. In *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013,* Hutchings, B. L. and Betz, V., editors, 53–62. ACM, (2013).

[12] Zhang, Y., Adl, K., and Glass, J. R. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2012, Kyoto, Japan, March 25-30, 2012*, 5173–5176. IEEE, (2012).

[13] Duran, A. and Klemm, M. In *HPCS,* Smari, W. W. and Zeljkovic, V., editors, 365–366. IEEE, (2012).

[14] Fu, A. W., Keogh, E. J., Lau, L. Y. H., Ratanamahatana, C. A., and Wong, R. C. *VLDB J.* **17**(4), 899–921 (2008).

[15] Sakurai, Y., Faloutsos, C., and Yamamuro, M. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007,* Chirkova, R., Dogac, A., Özsu, M. T., and Sellis, T. K., editors, 1046–1055. IEEE, (2007).

[16] Keogh, E. J., Wei, L., Xi, X., Vlachos, M., Lee, S., and Protopapas, P. *VLDB J.* **18**(3), 611–630 (2009).

[17] Fu, A. W., Keogh, E. J., Lau, L. Y. H., and Ratanamahatana, C. A. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005,* Böhm, K., Jensen, C. S., Haas, L. M., Kersten, M. L., Larson, P., and Ooi, B. C., editors, 649–660. ACM, (2005).

[18] Miniakhmetov, R., Movchan, A., and Zymbler, M. L. In *38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015, Opatija, Croatia, May 25-29, 2015,* Biljanovic, P., Butkovic, Z., Skala, K., Mikac, B., Cicin-Sain, M., Sruk, V., Ribaric, S., Gros, S., Vrdoljak, B., Mauher, M., and Sokolic, A., editors, 1399–1404. IEEE, (2015).