Parallelization of Searching and Mining Time Series Data using Dynamic Time Warping

Ahmed Shabib, Anish Narang, Chaitra Prasad Niddodi, Madhura Das, Rachita Pradeep, Varun Shenoy, Prafullata Auradkar, Vignesh TS, Dinkar Sitaram
Dept. of Computer Science and Engineering
PES Institute of Technology
Bangalore, India.
dinkars@pes.edu, prafullatak@pes.edu

Abstract—**Among the various algorithms present for data mining, the UCR Dynamic Time Warping (DTW) suite provided a solution to search and mine large data sets of time series data more efficiently as compared to the previously existing method of using Euclidean Distance. The UCR DTW algorithm was developed for a single CPU core. In this paper, we consider 2 methods of parallelizing the DTW algorithm. First, we consider a multi-core implementation, followed by a cluster implementation using Spark. From the multi-core implementation, we achieve nearly linear speedup. In the Spark implementation, we find that a straightforward implementation of DTW does not perform well. This is because; a major step in DTW is parallel computation of a lower bound. This paradigm is not supported well by Spark, which supports (i) *broadcast variables* that are broadcasts of read-only variables (ii) *accumulation variables* that represent distributed sums. We show how to compute distributed lower bounds efficiently in Spark and achieve nearly linear speedup with DTW in a Spark computation as well.**

Keywords—Time series; Dynamic time warping; Multicore; Spark

I. INTRODUCTION

There has been a tremendous growth of interest in applications that deal with querying and mining of time series data. Time series data is widely used in various fields such as medicine, finance, speech recognition and science, and thus development of an algorithm to mine such time series data is a necessity. There is increasing evidence that the classic DTW measure is the best measure in most domains [1]. In time series analysis, DTW is an algorithm for measuring similarity between two temporal sequences that may vary in time. Indeed, any data that can be turned into a linear sequence can be analyzed with DTW. In general, DTW is a method that calculates an optimal match between two given sequences (e.g. time series). This is achieved by aligning the two sequences and then creating a *warping matrix* for the <*candidate, query*> pair to obtain a warping path between the two. The warping path is used to find the similarity between the sequences.

As DTW is an important algorithm, and new parallel systems are becoming available, we have developed a parallel version of DTW algorithm to get a further speedup. In the present paper, we describe two alternative parallel implementations of the UCR suite, using multi core CPU and an Apache Spark cluster. We choose the multi core implementation because most modern CPUs come with multiple cores. We also choose the Apache Spark cluster implementation due to the emergence of high performance computing clusters.

II. RELATED WORKS

Dynamic Time warping or DTW is considered to be the best method for finding a subsequence *S* of a time series *T* that best matches a query pattern *Q*. This form of a subsequence search is used widely in data mining tasks such as classification, motif discovery, anomaly detection etc. Given that the DTW technique is one of the fundamental procedures for other tasks, development of speedup methods to reduce the computation time would be beneficial, allowing DTW to be used on larger datasets under various domains. The classic DTW is still slow since it has a quadratic time and space complexity [2]. The UCR suite has optimised the DTW algorithm by pruning out most of the sub-sequences that need to be considered [2]. The classical DTW algorithm is applied only to the sub-sequences that survive pruning. We observed that the UCR suite pruned out more than 90% of the sub-sequences that need to be considered less than 10% of the candidate sub-sequences need to be considered for a match. The UCR suite is therefore much faster than classical DTW and hence our focus is on parallelizing the UCR suite.

There have been earlier attempts to parallelize classical DTW such as presented in [5]. Hardware acceleration of the DTW algorithm has been achieved using GPU (Graphics Processing Unit) and FPGA (Field Programmable Gate Array). The aforementioned work makes use of the NVIDIA CUDA architecture, in which multiple threads running on multiple processing cores can work on the same data. In this way, parallel DTW distance computations take place on different segments of the time series. While parallelizing the UCR suite, it is important to efficiently parallelize the pruning algorithms, which are important to the efficiency of the UCR suite, since the speed of the UCR suite arises from its efficient pruning.

In this paper, we present techniques for parallelizing the UCR DTW suite using multi-core processors and commodity clusters. For commodity clusters, we have used Apache Spark, rather than MapReduce, which is the classic technique for analyzing very large datasets using clusters [3]. A problem with MapReduce is that it is not well suited to iterative computations, as the data is processed only once. Spark has features that allow it to perform iterative computations more efficiently [4]. Since DTW requires iterative computation, we have used Spark.

III. PARALLELIZED DTW

In this paper, we have implemented parallel computations of the UCR DTW suite. In the following sections we have described working of UCR suite and our parallel implementations on multi-core and Spark. Under multi-core, we have two implementations. The first implementation is parallelization of the entire UCR suite computations. The second implementation is parallelization of I/O read with computations

A. UCR Suite Implementation - Without any Parallelization

In order to find the best match between the candidate and the query the UCR suite [2] suggests the following steps. The query of size *m* is read from a query file and candidate is read from the data file (*EPOCH* number of points at a time). It is necessary to normalize these time series in similarity search to reduce the error rate as shown in [2]. This is done by Z-normalization or Z-score technique, performed in an incremental manner, in which the data values are normalized using the mean and standard deviation of the original data values (series). This method is also called Zero-Mean normalization because after this normalization the mean of normalized series becomes zero.

The query and a chunk of *m* points are considered from the candidate. LBKim, LBKeogh and DTW distance calculation are performed on them. LBKim and LBKeogh are lower bounding techniques that the UCR Suite introduces as optimizations [2]. These techniques help predicting and prune off unpromising candidates, thus reducing the number of candidates that go through till the DTW calculation. LBKim and LBKeogh use the *best-so-far* in order to eliminate candidates. LBKim compares the distance between the extremities of the two series with the *best-so-far* for pruning off candidates. LBKeogh compares the distance between one series and envelope created around the other series with the *best-so-far* to prune of candidates. The *best-so-far* is a measure which holds the best computed DTW distance at any point of time. If the lower bound exceeds the *best-so-far*, then the candidate is pruned off. If the candidate is not pruned off by the lower bound calculation then DTW distance calculation is done between the two time series. If the distance at any point of time exceeds the *best-so-far*, the remaining calculation is abandoned and the candidate is pruned off. Otherwise the *best-so-far* is updated with the most recent computed *best-so-far*.

B. Single Node Implementation - Parallelized DTW Computation using ThreadPool

The candidate is first divided into chunks of size of epoch. With every chunk of epoch points, the standard deviation and mean are also reset (required for Z-Normalization); this allows each chunk to be independent. The candidate is divided such that there is an overlap of size of the query along with the consecutively divided chunks.

Each CPU bound thread picks up a chunk and then works on it to get the best match. The steps performed by the threads are shown in the flowchart in **Fig. 1.** [6]

To find the best DTW match, we use the UCR DTW suite [2]. To attain maximum performance, each best match computed by each thread is communicated via a shared variable. To avoid races this shared variable has to be protected under a mutex. Therefore, every time a thread finds a better match, it updates this shared variable under the mutex.

In order to maximize performance, the read of the shared variable is not under a mutex. Due to this, when a read is performed the thread might read an old value. It is sometimes possible that a thread might end up updating the shared variable thinking that it has found a better match but the shared variable may have already been updated by a different thread to an even better match. To avoid this inconsistency, the shared variable is checked again after acquiring the mutex to ensure that the latest value in the shared variable is indeed the best.

We have reused threads and memory to prevent the overhead of object creation each time a new thread needs to work.

To maximize the performance, the data read operation is performed in parallel to the computations. For this purpose a thread is created that simply reads data from the disk and pushes it onto the work queue. A thread that has finished execution or a new thread on creation must pick up a data chunk from this queue and should continue to perform the computations. The multi-core implementation is done using a ThreadPool library.

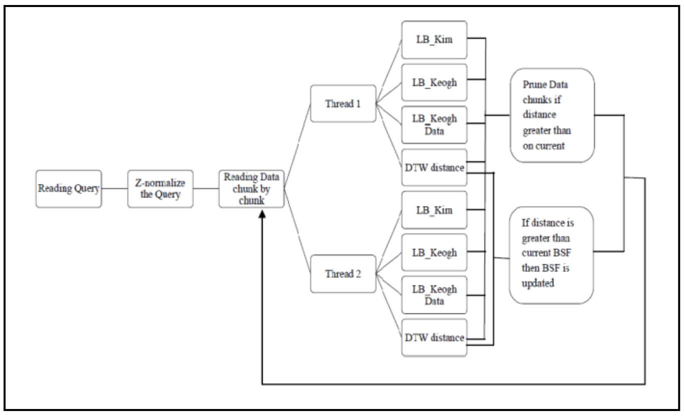
Reading Query
Z-normalize the Query
Reading Data chunk by chunk
Thread 1
Thread 2
LB_Kim
LB_Keogh
LB_Keogh Data
DTW distance
LB_Kim
LB_Keogh
LB_Keogh Data
DTW distance
Prune Data chunks if distance greater than on current
If distance is greater than current BSF then BSF is updated

Fig. 1. Flowchart for Parallelized DTW computation

ThreadPool is a cross platform C++ library. It is highly portable and runs on any platform that supports pthreads.

This implementation creates a pool of threads. Each thread services a request. As soon as servicing the current request is done, the thread is reused. Thus, it minimizes the overhead due to thread creation.

C. Single Node Implementation - Parallelized IO and Computation using OPENMP

In order to further optimize the UCR DTW algorithm, we can perform an overlapping of I/O (read) and computation (lower bound and distance calculations). We have implemented this using OpenMP.

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in languages such as C, C++ and FORTRAN.

OpenMP is a library which has an implementation of multithreading. It is efficient and widely used in high performance computing. The master thread forks a number of slave threads to distribute the task among them. Once the run time environment assigns threads to different processors, the threads execute concurrently. On completion of the parallel section, the threads re-join the master thread, which continues the execution of rest of the program.

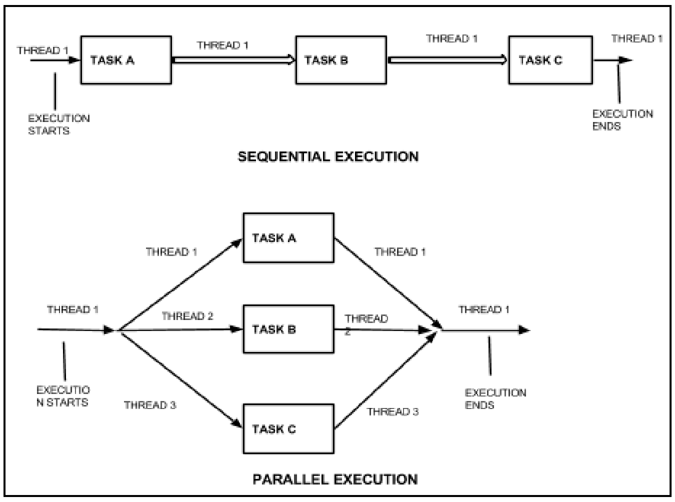

Fig. 2. Comparison of sequential vs. multithreaded parallel execution of three independent tasks.

The code is implemented using C++. First, the query file is read and all the data points are Z-normalized. After this, the program is divided among two sections that execute in parallel. Two threads are spawned for this purpose. One section performs the I/O - reading of the data file. The other performs the DTW computations in parallel.

However, the DTW computations themselves are performed sequentially within its thread. This section includes the optimizations such as LBKim, LBKeogh, LBKeogh2 and Z-normalization of data points from the data file.

The DTW computations section lags behind the file reading section by one iteration. This is because no DTW computation is performed when the first chunk of data points is read from the file during the first iteration.

Race conditions can possibly arise as the I/O section reads the next chunk of values when the DTW computations section processes the previous chunk. Hence the DTW computations section uses a copy of the values read by the I/O section in the previous iteration. The copies of required variables are created by a single thread at the end of each iteration. The variables *ep*, *buffer* filled by the I/O section in an iteration are being copied to variables *count*, *buffer1* and used in the DTW computations section in the forth-coming iteration.

Whenever a DTW distance lesser than the current *best-so-far* is obtained, the *best-so-far* and the corresponding location in the candidate where the *best-so-far* has been obtained are updated.

The process continues till all the data points in the candidate have been read chunk-wise. Finally, in the last iteration, no I/O operation is performed as all data points have already been read and only the last chunk of data points is left to be processed. Hence, only the DTW computations section executes in the last iteration.

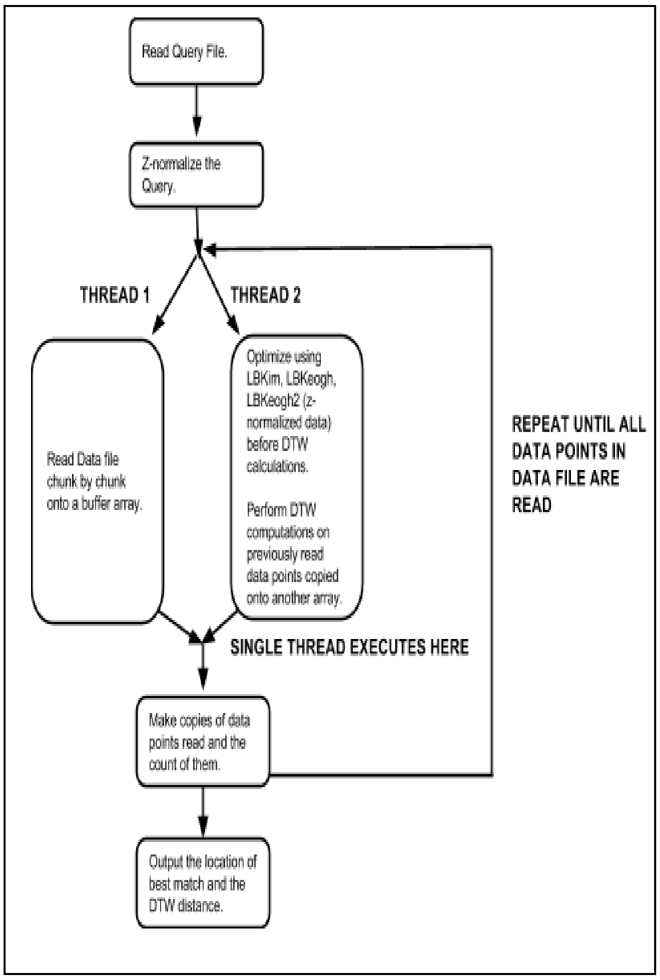
Read Query File.
Z-normalize the Query.
THREAD 1
THREAD 2
Read Data file chunk by chunk onto a buffer array.
Optimize using LBKim, LBKeogh, LBKeogh2 (z-normalized data) before DTW calculations.
Perform DTW computations on previously read data points copied onto another array.
REPEAT UNTIL ALL DATA POINTS IN DATA FILE ARE READ
SINGLE THREAD EXECUTES HERE
Make copies of data points read and the count of them.
Output the location of best match and the DTW distance.

Fig. 3. Flowchart for overlap of I/O and DTW computation.

D. Cluster DTW using Apache Spark

The sequential UCR suite algorithm cannot be directly implemented with Spark as the *best-so-far* needs to be computed in parallel. The main abstraction Spark provides is a Resilient Distributed Data set (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. These RDDs can be created from files present in Hadoop Distributed File System (HDFS). Applications on Spark are run as tasks coordinated by a master node in the cluster which has a driver program. Linear speedup is achieved by varying the number of nodes in the cluster.

Executors are started on each node of the cluster to perform the tasks. Spark API provides *broadcast variables* and *accumulators* in the form of shared variables which can be used to distribute data among the worker nodes, but these don't fit our requirement as they need to be overwritten with every calculation of the *best-so-far*. However, Spark does not provide a technique to use lower bounding.

The time series data is in the form of a text file and is split using a Python program. For example, to search through a time series data set of 10 million points with a query of length 128, the file is split into 100 parts, each consisting of 100128 points, where the extra 128 points are the last 128 points of the previous partition as a buffer to prevent loss of precision while querying. These files are stored into the Hadoop Distributed File System (HDFS) and replicated on all nodes to be computed on in parallel.

The UCR suite was implemented in Scala to be run on Spark as Scala is the recommended language for use with Apache Spark. An RDD is created from the files stored in HDFS. The Spark configuration is modified to set the parallelism to the number of cores present in the cluster to utilize the cluster to its maximum capacity. It is recommended that a maximum of 2-3 tasks are to be run on a CPU core at an instance of time. The number of tasks executed in parallel on a node is equal to the number of cores in the corresponding node. Spark automatically sets the number of map tasks to run on each file according to the number of partitions present. A partition is defined by the each file which is loaded from the HDFS. Using actions and transformations on RDDs, the location of the current *best-so-far* is maintained. A reduce or collect action needs to be necessarily executed for the driver to complete tasks from the nodes of the cluster. A reduce was used in this implementation. During the reduce phase, the least *best-so-far* and its corresponding location is returned to the driver program by the workers in the cluster to prevent executing another search among the returned tuples to find the optimal *best-so-far*. To speed up the process of serializing the tasks to be run, we have used the Kryo serializer [7].

However, speedup was low. A reason for this is due to the computation of the *best-so-far*. Re-computing the lower bound, which is the *best-so-far*, for every partition will lead to a decrease in speedup. The *best-so-far* needed to be stored in a location which could be accessible by all the nodes during the process of running the application. As an alternative, we split the data sets into an optimum number of partitions. A Python server runs on the master node to keep track of the *best-so-far* with the help of a socket connection. This design was found to be suitable for our implementation as computation of the *best-so-far* was not an overhead, as in the multi-core implementation. The *best-so-far* is initialized to infinity only at the beginning of the application. A task is run on each of the split files and the Python server receives a key-value pair of the *best-so-far* and location from the worker nodes at the end of each task from each partition. This *best-so-far* value is updated to the lowest value on the server and sent to the subsequent tasks in order to initialize their *best-so-far* to a lower value, thus helping in pruning off more of the unpromising candidates. This is a general method that can be implemented in Spark for parallel computation of a lower bound. The socket programming overhead is much lower and does not add much to the time consumed by the Spark application. The above implementation is illustrated in **Fig.4**.

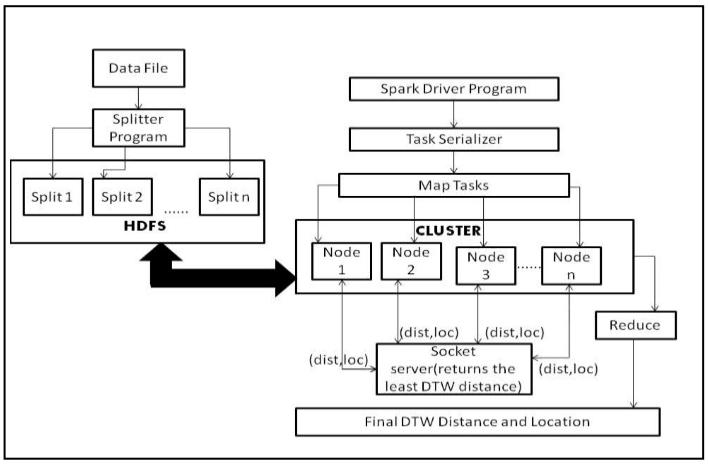

Fig. 4. Parallel DTW implementation on Spark Cluster

IV. Results

All machines used in the implementation have a 12GB RAM. The data sets have been generated from a python program which writes a Random Walk into a text file. All the implementations have been tested on common data sets for varying sizes. The query used is the same as the one being used by the UCR suite [4]. The query length is 128 points. The data is a random walk data set generated by a python script taking the number of data points as an input from the user. The algorithm used the warping window as 5% of the query length. The UCR results provided here are from the source code available in the UCR suite[2] run on Intel(R) Xeon(R) CPU E3-1200 V2 at 3.10GHz with a 12GB RAM.

A. Single Node Implementation Results

TABLE I. Parallelized IO and computation Dual and Quad Core machines with 12GB RAM and 2 threads

Data File Size(GB)	UCR implementation of DTW (seconds)	Parallelized I/O and Computation (seconds)	Speed Up
3	137	87	1.57
5	242	160	1.51

7	320	210	1.52
9	398	247	1.61
11	421	290	1.45

The overlap of I/O with computation uses only 2 threads, one for the I/O and the other for the computations. The computations are serialized within the thread executing it. Hence, increasing the number of threads does not give a speed up. The time of computation thus remains same for 2 and 4 cores. This gave a speedup of up to 1.6 times.

TABLE II. PARALLELIZED DTW COMPUTATION ON DUAL CORE MACHINES WITH 12GB RAM AND 5 THREADS

Data File Size(GB)	UCR implementation of DTW(seconds)	Parallel Computation(seconds)	Speed Up
3	137	38	3.60
5	242	73	3.31
7	320	115	2.78
9	398	154	2.58
11	421	171	2.46

TABLE III. PARALLELIZED DTW COMPUTATION ON QUAD CORE MACHINES WITH 12GB RAM AND 9 THREADS

Data File Size(GB)	UCR implementation of DTW(seconds)	Parallel Computation(seconds)	Speed Up
3	137	43	3.18
5	242	72	3.36
7	320	93	3.44
9	398	125	3.18
11	421	125	3.36

TABLE IV. COMPARISON OF THE SERIAL IMPLEMENTATION OF DTW VS. THE PARALLEL IMPLEMENTATION

Data File Size(GB)	UCR implementation of DTW(seconds)	Parallel Computation(seconds)	Parallelized I/O and Computation(seconds)
3	137	43	87
5	242	72	160
7	320	93	210
9	398	125	247
11	421	125	290

The parallelization of DTW computation was run on different number of cores like 2 cores and 4 cores each with 12 GB RAM. This code gave a speedup of about 3x when run on a four core and 2x when run on the 2 core. For bigger data sets the 4 cores and 2 cores gave significant difference in speed up but were almost negligible when it came to smaller data sets.

Out of the two multi-core implementations, we observe that by parallelizing the DTW computations, we obtain a better speedup than by parallelizing the I/O and the DTW computation. Hence, we can infer that the DTW computations take more time than the file reading (I/O).

B. *Results for Spark Iplementation*

The scaling of performance with varying data size is presented in Tables V, VI and VII. Each node in the cluster has 4 cores. The data was replicated across the HDFS by the same number of nodes present in the cluster.

The number of files that each dataset has been split to has also been mentioned. This is the optimum number of splits for the respective dataset sizes, to achieve maximum speed up.

TABLE V. RESULTS OF 2 NODE SPARK CLUSTER

Data size(GB)	Number of split files	Total number of data points	UCR timing (seconds)	Spark timing(seconds)	Speed up
3	300	220000000	137	46.5	2.94
5	450	400000000	242	85	2.84
7	500	520000000	320	129	2.48
9	600	660000000	398	210	1.90

TABLE VI. RESULTS OF 4 NODE SPARK CLUSTER

Data size(GB)	Number of split files	Total number of data points	UCR timing (seconds)	Spark timing(seconds)	Speed up
3	300	220000000	137	23	5.1
5	450	400000000	242	41	5.1
7	500	520000000	320	60	5.9
9	600	660000000	398	77	5.9

TABLE VII. RESULTS OF 6 NODE SPARK CLUSTER

Data size(GB)	Number of split files	Total number of data points	UCR timing (seconds)	Spark timing(seconds)	Speed up
3	300	220000000	137	32	4.28
5	450	400000000	242	39	6.21
7	500	520000000	320	53	6.03
9	600	660000000	398	51	7.80

With the Spark implementation we see that on an average, the 2 node Spark cluster gives a speedup of 2.54, the 4 node Spark cluster gives a speedup of 5.5 and a 6 node cluster gives us a speedup of 6.08. We can see that there is a substantial increase in the speedup which is directly proportional to the number of nodes in the cluster. In contrast, for smaller data sets, having a large number of nodes is unnecessary as starting up the cluster for a large number of nodes is time consuming and affects the overall speedup.

The number of files the candidate is split into affects the speed of computation. Sufficient care should be taken while splitting the candidate such that there is no loss of accuracy. The splitting of files above was found to be the optimum number for the corresponding data sizes. Having too many partitions will produce the wrong results. Having too few partitions will reduce the performance because the initial DTW distance computed which passes through all the lower bounding techniques behaves as a threshold lower bound for the remaining candidates. Automatically determining the split size for each node to process is a topic of future exploration.

V. Conclusion

Multi-core and cluster computing can be applied to problems like pattern matching. From the above results, we can conclude that parallelizing the sequential DTW code improves the performance of the algorithm while maintaining 100% accuracy to give a linear speedup in both cases. This is quite obvious as the throughput of processes increase on parallelization. Our research has improved the DTW algorithm by parallelization on the software level due to advancement of technology over the years by improving the speed in which the time series are matched.

Acknowledgment

We would like to thank General Electric Research for their support in the form of a research grant. We would like to thank Prof. Phalachandra H L, Prof. Sudheendra H, Vincent Kuri, Naveen Kunchagi, Vinay U S and Muralidhar Shetty for their support in maintaining the hardware setup.

References

[1] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. J. Keogh. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. PVLDB 1, 2.

[2] Rakthanmanon, Thanawin, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. "Searching and mining trillions of time series subsequences under dynamic time warping." In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 262-270. ACM, 2012.

[3] Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations Yuan Yu Pradeep Kumar Gunda Michael Isard

[4] Map-Reduce for Machine Learning on Multi-core. Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, Kunle Olukotun CS. Department, Stanford University 353 Serra Mall, Stanford University, Stanford CA .

[5] Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs. Doruk Sart, Abdullah Mueen, Walid Najjar, Eamonn Keogh, University of California, Riverside.

[6] Monitors: An Operating System Structuring Concept, C.A.R.Hoare

Gagne, Abraham Silberschatz, Peter Baer Galvin, Greg. Operating system concepts (9th ed.). Hoboken, N.J.: Wiley. pp. 181–182. ISBN 9781118063330.

[7] Notes Documentation Spark Release 1.0.0; Lian Cheng