TPICDS: A Two-Phase Parallel Approach for Incremental Clustering of Data Streams

Ammar Al Abd Alazeez(✉), Sabah Jassim, and Hongbo Du

Department of Applied Computing, The University of Buckingham, Buckingham MK18 1EG, UK
{1405097, sabah.jassim, hongbo.du}@buckingham.ac.uk

Abstract. Parallel and distributed solutions are essential for clustering data streams due to the large volumes of data. This paper first examines a direct adaptation of a recently developed prototype-based algorithm into three existing parallel frameworks. Based on the evaluation of performance, the paper then presents a customised pipeline framework that combines incremental and two-phase learning into a balanced approach that dynamically allocates the available processing resources. This new framework is evaluated on a collection of synthetic datasets. The experimental results reveal that the framework not only produces correct final clusters on the one hand, but also significantly improves the clustering efficiency.

Keywords: Big data · Data stream clustering algorithms
Distributed and parallel frameworks

1 Introduction

Recent advances in information and networking technologies and their applications in almost every sector of life have led to a rapid growth of the massive amount of data known as *Big Data* [1]. One of the most important characteristics of big data is its *velocity*, which means that data may arrive and require processing at different speeds. While for some applications, the arrival and processing of data can be performed in an offline batch processing style, others require continuous and real-time analysis of collections of incoming data (known as data chunks) ([2–4]). Data stream clustering is defined as a grouping of data in light of frequently arriving new data chunks for understanding the underlying group patterns that may change over time [5].

It is the sheer volume of data arriving at high and variable speeds of accumulation that deems normal clustering algorithms inefficient and incapable of dealing with the demand [6]. Therefore, distributed and parallel algorithms are the ultimate solution for analysing big data streams in reality, which is evident in the more recent research work ([4, 7, 8]). Distributed and parallel solutions offer several benefits such as reduction of the overall response time, improved scalability of solutions and suitability for applications of distributed nature such as sensor networks, social media, Internet of Things (IoT), etc. [9].

Multi-core processor commodity computers are widely used nowadays. At a higher but affordable price, a computer can have up to 72 core processors. As the computer hardware technology advances, cheaper and more core processors will become available. The question then is how to utilise the available processing resources on board of a local machine. In this paper, we argue that algorithms for data stream clustering should be first implemented on a multi-core parallel processing framework by making the best use of the available processors in a local machine before running the algorithms in a distributed network of computers.

This paper is therefore concerned with how to parallelise most recent techniques for clustering data streams. In general, the paper promotes a two-phase parallel approach for incrementally clustering data streams (TPICDS) where processors will incrementally maintain local clustering models in parallel at the online phase, and local cluster models can be merged into a global cluster model at the offline phase. In particular, the paper investigates the parallelisation of a recent algorithm EINCKM [10] in the TPICDS framework because of the algorithm's modular structure and performance over other existing algorithms. The work consists of two parts. In the first part, the paper investigates how the EINCKM algorithm adapts three typical parallelisms in existence. Based on a performance evaluation of the adapted parallelisms inside the algorithm, the paper further proposes a parallel pipeline with optimised and dynamic allocations of processing resources. Experimental results show that the proposed solution not only produces correct final clusters, but also significantly improves the efficiency.

The rest of this paper is organised as follows. Section 2 explains the related work on distributed and parallel data stream clustering algorithms in the literature, and propose the TPICDS approach at the end. Section 3 explains the EINCKM algorithm adaptation of the existing parallelisms. Section 4 presents the proposed optimised and dynamic parallel pipelines. Section 5 concludes the work and outlines possible future directions of this research.

2 Related Work

2.1 Computational Approaches

Two approaches for mining data streams are in existence: incremental and two-phase learning. With the incremental methods (e.g. STREAM [11]), a global model of clusters is iteratively developed to reflect current modifications made by incoming data chunks. The two-phase approach (e.g. CluStream [12]) divides the clustering process into two phases, i.e. an online phase where the data records are summarised into small intermediate *micro-clusters*, and an offline phase where the micro-clusters are processed into final clusters at a query point [13]. While the incremental algorithms always provide an accumulated view of global clusters at the arrival point of an incoming data chunk at the expense of continuous clustering, the two-phase algorithms provide such a view of clusters at the point of the query without constantly finding final clusters. Therefore, it can be argued that incremental algorithms are more suited for real-time response systems [4].

2.2 Data Stream Clustering Algorithm EINCKM

EINCKM is a prototype-based algorithm for clustering data streams and identifying outlier objects [10]. Taking the incremental learning approach, the algorithm divides the clustering process into three sequential steps: *Build Clusters*, *Merge*, and *Prune*. *Build Clusters* uses the K-Means method to find the clusters in the input data chunk. *Merge* integrates the newly formed clusters with existing ones. *Prune* detects outliers and checks the concept drift using a fading function. The algorithm applies a heuristic-based method to estimate the number of clusters, a radius-based technique to merge overlapped clusters, and a variance-based mechanism to prune outliers. The algorithm is modular and adaptable to further improvements. However, the algorithm is a sequential algorithm where the three key operations must be performed in order. It is, therefore, useful to explore how to parallelise the algorithm.

2.3 Distributed and Parallel Frameworks

Depending on how the input data is organised, two categories of distributed and parallel data stream clustering algorithms exist: object-based where the data record is a complete data object and attribute-based where each data item is an attribute value. Each category may take either incremental or two-phase learning approach.

For the incremental learning of object-based clusters, the central site receives the input data streams, divides it into chunks and sends them to the remote sites. Upon receiving the local clustering models from the remote sites, the central site produces the final output clusters. Bandyopadhyay *et al.* have used this approach for clustering data streams in a peer-to-peer environment [14]. Gao *et al.* showed an enhanced Apache Storm framework for clustering social media data, by adding another process between the central site and the participant remote sites for synchronising changes to the local models to avoid a bottleneck in communications [15]. Incremental learning of clusters from attribute streams is similar. The only differences are that each remote site receives an input attribute stream directly without the central site to distribute the data and that upon receiving the local cluster models, the central site integrates the local attribute models into a global object-based cluster model. Rodrigues *et al.* used this approach in the ODAC algorithm to cluster attribute streams [16]. The incremental learning is simple, easy to implement and efficient. However, extensive communication with the central site can result in bottlenecks. Besides, integrating all attribute clusters with a central site becomes infeasible when the dimensionality of data streams are high.

In the two-phase learning of object-based clusters, the local models are saved in a local buffer memory on each remote site, and are sent to the central site when there is a query from the user or there are significant changes in the local models [17, 18]. However, heavy computation is needed with the central site to obtain the final output clusters due to the large number of micro-clusters. Guerrieri and Montresor presented an improvement by making the remote sites communicate to reduce the number of micro-clusters [19]. Karunaratne *et al.* also made an improvement using Apache Storm where the remote sites save their local clustering models in a globally shared memory so that the designated second central site processes the local clusters into the final clusters [19]. The two-phase learning of clusters from attribute streams is similar to

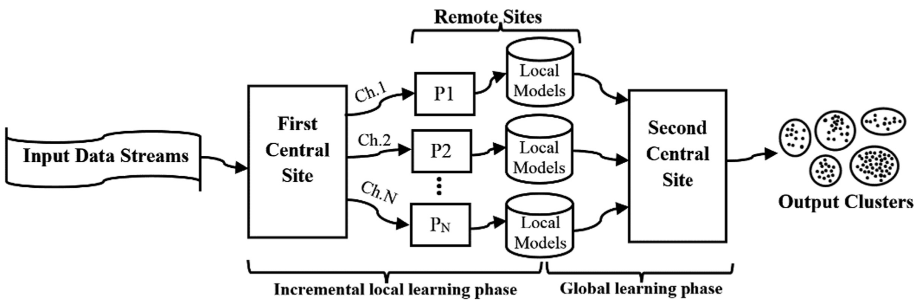


Fig. 1. Proposed TPICDS framework

incremental learning by using buffers for local models. Gama *et al.* adopted this approach in the DGClust algorithm to cluster data streams in sensor networks [20]. The algorithm reduces data dimensions and hence communications needed.

2.4 The Proposed TPICDS Framework

The proposed framework TPICDS combines the two computational approaches in the two-phase hybrid system. At the online phase, the first central site receives the data streams, divides it into chunks, and sends then to the remote sites. Each remote site receives its own data chunk, creates and maintains its own local cluster models in the incremental fashion. At the offline phase, the second central site receives the local models from the remote sites and presents the final global clusters (see Fig. 1). Our research aims to embed the EINCKM algorithm within the proposed framework where the second central site uses the same merge strategy to form the global clustering model.

3 Adapting EINCKM to the Existing Parallelisms

In this section, we first briefly summarise three typical parallel frameworks that already exist. We then describe how the EINCKM algorithm can intuitively adapt to each framework. We then evaluate the performance of each adaptation empirically using synthesised datasets.

3.1 Existing Parallel Frameworks

Three typical parallel frameworks exist. The replication, hereby known as embarrassingly parallel or basic parallelism (BP), simply makes multiple copies of the entire algorithm and then runs each copy on each processor [21]. Each processor must complete all operations of the algorithm before receiving next new data inputs. The framework has the pros of being simple, and directly employs the principle of divide and conquer by sharing the processing of input data by the available processors. We use this framework as a basic benchmark for performance evaluations later.

The parallel pipeline (PP) is an improvement of the basic parallelism to streamline several processors in a pipeline [22], which works as follows. A present processor receives the output of a previous processor as its input, processes the data, generates the output, and passes it on to the next processor. Each processor has a certain degree of independence so that when the next processor processes its output, the present processor takes and processes its next data input. This framework not only divides the workload among different processors but also further improve the degree of parallelism within a sequential pipeline.

MapReduce parallel pipeline (MRPP) is a further modification of PP [23]. First, data stored in memory are divided into partitions, and each partition is sent to a different processor (Mapping). Each processor processes the data within the partition and the processed outputs are then hashed to fewer processors on another layer to further process them (Reducing). The hashing can be determined by the relevance of the outputs.

3.2 Algorithm Adaptation

All adaptations of the algorithm to the existing frameworks mentioned above require dividing the available processing resources into central and remote sites (in this context, a site is a single core processor on the same computer). The adaptation of the algorithm to the basic parallelism is straightforward: each remote site finds the clusters from the incoming chunk, merges them with its own existing clusters, prunes them, and saves the resulting clusters into its own local buffer memory.

The adaptation of the algorithm to the PP framework is done as follows. We first divide all the available remote sites into groups of *three* sites, and then arrange the three sites into a pipeline. We then designate the first of the three sites for the *Build Clusters* function, the second for the *Merge* function, and the third for the *Prune* function. When the *Build Cluster* site finishes the current chunk, it sends the clusters to the *Merge* site. When the *Merge* site merges the clusters from the chunk with the existing ones, the *Build Cluster* site starts discovering clusters from the next chunk. When the *Merge* site finishes its task of merging clusters, it sends the results to the *Prune* site, and then starts working on the new clusters from the *Build Cluster* site. The *Prune* site works in a similar fashion.

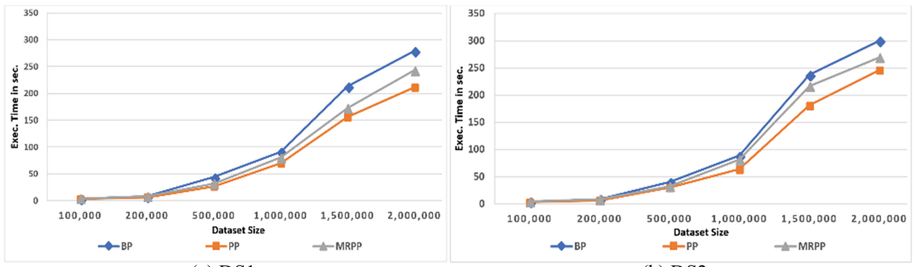
For the adaptation of the MRPP framework, the first central site in TPICDS performs the mapping operation. The remote sites for the *Build Cluster* function is modified to include a further function for the hashing, i.e. assigning similar local clusters to a specific *Merge* site. More precisely, a *Build Cluster* site checks the closest cluster's centroids and send them to the same merger site, and a *Merge* site receives clusters from different *Build Cluster* sites to build a *regional model* of clustering. After that, the *Prune* site conducts the pruning of regional models and sends them to the second central site in TPICDS. The difference between the PP adaptation and the MRPP adaptation is that the *Merge* site in the PP framework receives clusters only from the *Build Cluster* site within the same pipeline, whereas the *Merge* site in the MRPP merges clusters from more than one *Build Cluster* site in different pipelines.

3.3 Empirical Evaluation

In order to evaluate empirically the performance of each adaptation, we created two collections of synthetic datasets, DS1 and DS2. Each collection include six datasets of different sizes, i.e. 100,000, 200,000, 500,000, 1,000,000, 1,500,000 and 2,000,000 data points of two dimensions. To simulate various sizes, shapes and numbers of clusters, we used Gaussian distributions to randomly generate spherical shape clusters with different means, variances and number of members. DS1 and DS2 respectively have four and thirty clusters. We acknowledge the limitations of synthesised datasets in expressing the characteristics of data in reality, but synthesised datasets do allow us to check the correctness of clustering by comparing the resulting clusters to known clusters.

A computer with 12 2.8 GHz core processors and 16 GB memory under Microsoft Windows7 was used to conduct the experiment. MATLAB 2017a was used to implement the adapted algorithms and program the experiment scripts. For each experiment, we randomly selected data points from the dataset to form data chunks of 1000 data points. The random selection simulates the situation where there is no control on the order of the arriving data points. To minimise the random effect of the selected data points to the performance of the algorithms for a specific experiment, we repeated each experiment 100 times, and then take the average of the speeds of execution in seconds. The processors on the machine are configured as follows. For the BP framework, we allocate three processors each of which has the entire EINCKM algorithm. For the PP and MRPP frameworks, we allocate three processors (one for *Build Cluster*, one for *Merge*, and one for *Prune*) to form one parallel stream. We allocate two processors serving as the two central sites in TPICDS. Figure 2 shows the performance of the adapted EINCKM algorithms in terms of execution time.

Among the adapted algorithms, the BP adaptation is slower than the PP and MRPP. The two pipeline adaptations show consistent faster speeds due to the additional parallelism gained from the pipeline frameworks. However, the MRPP adaptation consumes more time than the PP adaptation in mapping and hashing similar local clusters to a merger. The PP adaptation, however, may still have potential delays because the processor configuration on each pipeline was fixed, and some processors within the pipeline may have to wait for the outputs from other processors. Therefore, optimised and dynamic allocations of processors to the needed steps should be the right way to further exploit the parallelism.



(a) DS1 (b) DS2

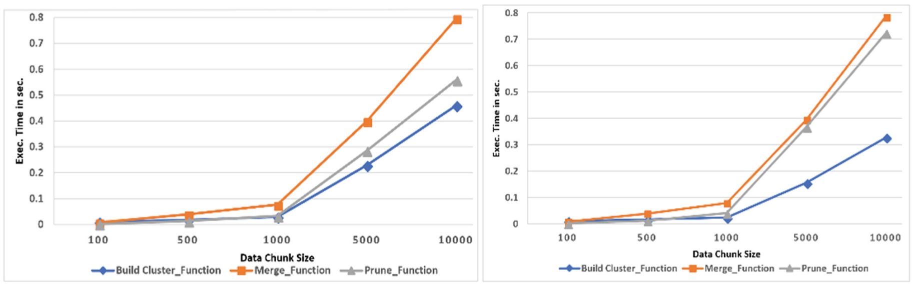
Fig. 2. Adapted algorithms performance in execution time

4 Optimised and Dynamic Parallel Pipeline Frameworks

4.1 An Optimised Parallel Pipeline Framework

The idea behind optimised parallel pipeline (OPP) framework is to decide how many processors should be statically allocated for each step of the EINCKM algorithm by analysing the time complexity of the algorithm. The time complexity of the entire algorithm is estimated by the sum of the time complexity for each of the three key functions [10]. Let R represent the number of chunks, N the total number of data points in a chunk plus the outliers, K the number of clusters, I the number of iterations until the clusters converge, T the number of clusters of the previous iteration, n the maximum number of data points in a new/existing cluster, k the number of clusters from a new chunk, and S the number of output clusters of merge function. The time complexity is $O(NKI)$ for the *Build Cluster* function, $O\left((Tn+kn)^2\right)$ for the *Merge* function, and $O(Sn)$ for the *Prune* function. The expressions indicate that the *Merge* function takes the longest amount of time in the worst case. This is followed by the *Prune* function. The *Build Cluster* function needs relatively the minimum amount of time because the values for N, K and I are normally small. In order to confirm the results of the theoretical analysis, we tested each function separately on synthesised data chunks of different sizes. Figure 3 illustrates the execution time for each function at different chunk sizes for DS1 and DS2 datasets. The test results confirm the theoretical analysis results.

According to this understanding, we configure the 12-core machine in the following way: two processors for the *Build Cluster* function, four processors for the *Merge* function, and three processors for the *Prune* function (see Sect. 4.3 for performance test results), plus two processors serving as the two central sites.



(a) DS1 (b) DS2

Fig. 3. Comparison of execution time among the key functions of EINCKM

4.2 Dynamic Parallel Pipeline Framework

The worst-case measure of time complexity does not always reflect the reality. Dynamic scheduling of resources based on actual execution of each individual step makes more sense in deciding how many processors should be allocated to resolve the bottleneck at the time. Therefore, a dynamic parallel pipeline (DPP) framework is proposed. In this framework, a minimum number of processors are initially allocated as the *baseline* processors for performing the clustering task. One central processor is then designated to the role of *scheduler* by monitoring occupancy rates of the buffers being used by the existing processors. A number of spare processors are held in reserve. A spare processor can be assigned to join the baseline processors for a specific function by the scheduler according to the need for additional assistance as indicated by the level of free buffer memory.

We encountered two immediate problems: (a) how to select the right number of baseline processors, and (b) how to decide if there is a need for allocating additional resources. To solve the first problem, we allocate by default one processor for each of the three key functions of the algorithm. To solve the second problem, we monitor the size of the buffers, decide where the possible bottleneck may occur and then take a decision to add/move a processor one at a time. For each iteration, the scheduler checks the use of two buffers (Bf1 and Bf2). If the buffer use, i.e. use of the storage space of the buffer, is below a minimum threshold (*Min_Thr*), the buffer is about to become empty and hence more processors are needed by the function that outputs *into* the buffer. If the buffer usage is above a maximum threshold (*Max_Thr*), it means that the buffer is about to become full and more processors are needed for the function that inputs *from* the buffer. Figure 4 shows four decision rules for the two possible situations: (a) there are processors in the reserve (assigning a processor), and (b) there are no processors in the reserve (moving a processor).

```
Dynamic Parallel Pipeline Algorithm:
Algorithm Steps:
  $if\ size(Reserve) > 0\ then$ // (a) First situation when have reserved processors
    $if\ size(Bf1) < Min\_Thr\ \&\ size(Bf2) < Min\_Thr\ then\ add\ processor\ to\ Build\ Cluster_s$
    $elseif\ size(Bf1) < Min\_Thr\ \&\ size(Bf2) > Max\_Thr\ then\ add\ processor\ to\ Prune_s$
    $elseif\ size(Bf1) > Max\_Thr\ \&\ size(Bf2) < Min\_Thr\ then\ add\ processor\ to\ Merge_s$
    $elseif\ size(Bf1) > Max\_Thr\ \&\ size(Bf2) > Max\_Thr\ then\ add\ processor\ to\ Prune_s/Merge_s$
  $else$ // (b) Second situation when do not have reserved processors
    $if\ size(Bf1) < Min\_Thr\ \&\ size(Bf2) < Min\_Thr$
      $then\ take\ one\ processor\ from\ Prune_s\ /\ Merge_s\ and\ add\ it\ to\ Build\ Cluster_s$
    $if\ size(Bf1) < Min\_Thr\ \&\ size(Bf2) > Max\_Thr$
      $then\ take\ one\ processor\ from\ Merge_s\ and\ add\ it\ to\ Prune_s$
    $if\ size(Bf1) > Max\_Thr\ \&\ size(Bf2) < Min\_Thr$
      $then\ take\ one\ processor\ from\ Prune_s\ and\ add\ it\ to\ Merge_s$
    $if\ size(Bf1) > Max\_Thr\ \&\ size(Bf2) > Max\_Thr$
      $then\ take\ one\ processor\ from\ Build\ Cluster_s\ and\ add\ it\ to\ Prune_s\ /\ Merge_s$
```

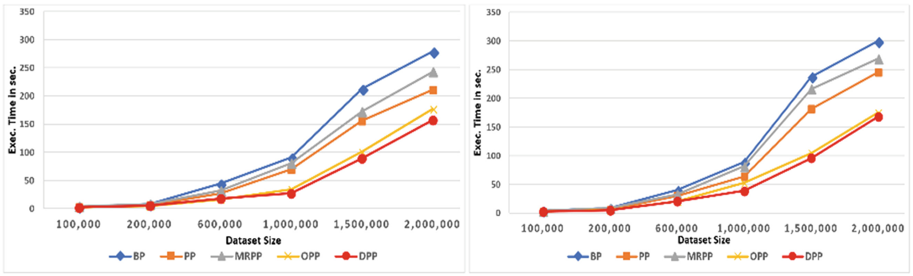
Fig. 4. Dynamic parallel pipeline framework

4.3 Experimental Results and Discussion

We used the two collections of datasets to test the performance of the OPP and the DPP frameworks. The results in Fig. 5 show that both OPP and DPP are consistently faster than the PP framework, confirming that optimised and dynamic allocations of processing resources are better than the even distribution of the resources among the processing steps. At the same time, the DPP framework performs better than the OPP one because the statically allocated processors in OPP does not reflect the dynamic reality.

One issue that affects the performance of the DPP framework is the setting of the two thresholds for the buffer use. For the tests presented in Fig. 5, we set *Min_Thr* = 20% and *Max_Thr* = 80%. Setting the range between the two thresholds too low means too many scheduling activities for additional resources. Setting the range too big means increasing the risk of the buffers being empty or full causing time delay in the process. Other factors such as the speed of data arrival and buffer sizes also play a role. A proper sensitivity study regarding the thresholds and the search for optimal thresholds certainly require further research.

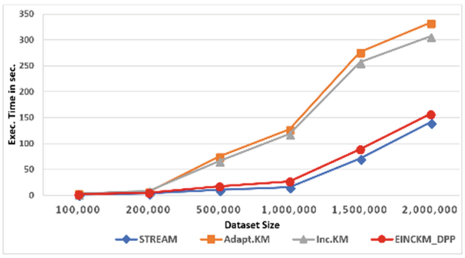
We also compare the DPP version of the EINCKM algorithm against the BP versions of three typical existing algorithms of the same category, i.e. STREAM, Adapt.KM [24], and Inc.KM [25]. The results show faster execution time by the EINCKM algorithm than that by the Adapt.KM and the Inc.KM algorithms due to the dynamic allocations of processing resources to the right place in the EINCKM algorithm. The EINCKM algorithm speed is close to that, but slower than that of the STREAM algorithm (see Fig. 6). This is mainly because the STREAM algorithm does not consider the concept drift issue and nor identify outliers as the EINCKM does.



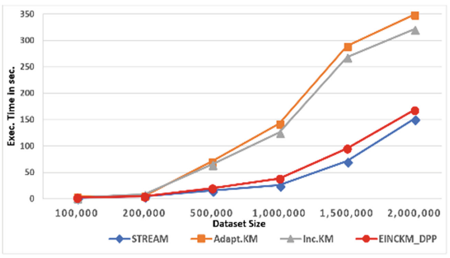
(a) DS1 (b) DS2

Fig. 5. The ratio between ideal time and the measured parallel frameworks

Regarding the correctness of the output clusters, we confirm that all the five versions of parallel EINCKM algorithm produce correct global clusters after the whole datasets are processed. We have evidence to demonstrate the correctness of the final global cluster models by comparing the output clusters by the algorithms against the ground truth clusters in terms of the correctness metrics such as purity, entropy, and the sum of square errors measurements. However, because of the constraints of the limited space, we are unable to present the evidence here.



(a) DS1



(b) DS2

Fig. 6. Comparison between algorithms

Next generation of online real-time systems required big data platforms to process a huge amount of continuously arriving data under computational constraints [26]. This kind of systems raises new issues regarding the current big data infrastructures. One of the main issues is that most current platforms are not intentionally built to consider real-time performance issues. Another main issue is the lack of clear computational models for processing big data that could be supported by the current frameworks [8]. Recent attempts to address these issues include a study of analysing patterns in data stream processing and associating the patterns with performance requirements [4], and an effort in improving the computational model for distributed stream processing and formalising the model through extensions to the Storm framework for real-time application [27]. Our proposed parallel frameworks can be considered as another attempt to address the infrastructure issue for real-time applications at least on the local individual machine level. The strengths and limitations of the proposed framework have not been, but can only be realistically evaluated within the context of a large-scale distributed processing environment.

5 Conclusion and Future Work

This paper made two main contributions: (a) adapting a newly developed data stream clustering algorithm EINCKM to existing parallel frameworks, and (b) developing static and dynamic allocation schemes for utilising available processors, both within a two-phase learning approach (TPICDS). The adaptation is made easier because the algorithm has a modular structure, making it easy to adapt pipeline frameworks. The evidence shows that the static and dynamic allocations of processing resources is more efficient than simple adaptations.

The understandings we take from our work are of two folds. Firstly, there is a room to utilising as much as possible the available resources within a single computer before we bring in a group of computers to share the workload distributedly. Secondly, the two learning approaches for data stream clustering are artificially separated. The paper shows that a hybrid way of merging them in a parallel pipeline is possible.

Future work includes an immediate sensitivity analysis for the buffer thresholds and more extensive testing of the proposed dynamic parallel pipeline version of the EINCKM algorithm, and further improvements to dynamic allocation of resources by using more sophisticated techniques including machine learning techniques. Reclaim of processors into the reserve should also be considered when the speed of incoming data arrival slows down and there is no need to use a large number of processors to share out a small amount of workload. Another important work is the integration and testing of the dynamic parallel pipeline on a single computer with a distributed network environment.

Acknowledgements. The first author wishes to thank the University of Mosul and Government of Iraq/Ministry of Higher Education and Research (MOHESR) for funding him to conduct this research at the University of Buckingham.

References

1. Liu, C., Ranjan, R., Zhang, X., Yang, C., Georgakopoulos, D., Chen, J.: Public auditing for big data storage in cloud computing – a survey. In: 2013 IEEE 16th International Conference on Computational Science and Engineering, pp. 1128–1135, December 2013
2. Olshannikova, E., Ometov, A., Koucheryavy, Y.: Towards big data visualization for augmented reality. In: 2014 IEEE 16th Conference on Business Informatics, pp. 33–37, July 2014
3. Kaur, N., Sood, S.K.: Efficient resource management system based on 4Vs of big data streams. J. Big Data Res. **9**, 98–106 (2017)
4. Basanta-Val, P., Fernandez-Garcia, N., Sanchez-Fernandez, L., Arias-Fisteus, J.: Patterns for real-time stream processing. IEEE Trans. Parallel Distrib. Syst. **28**(11), 1–91 (2017)
5. Yogita, Y., Toshniwal, D.: Clustering techniques for streaming data – a survey. In: 3rd IEEE International Advance Computing Conference (IACC), pp. 951–956 (2012)
6. Sliwinski, T.S., Kang, S.-L.: Applying parallel computing techniques to analyze terabyte atmospheric boundary layer model outputs. J. Big Data Res. **7**, 31–41 (2017)
7. Yusuf, I.I., Thomas, I.E., Spichkova, M., Schmidt, H.W.: Chiminey: connecting scientists to HPC, cloud and big data. J. Big Data Res. **8**, 39–49 (2017)
8. Lv, Z., Song, H., Basanta-val, P., Steed, A., Jo, M.: Next-generation big data analytics: state of the art, challenges, and future research topics. IEEE Trans. Industr. Inf. **13**(4), 1891–1899 (2017)
9. Aggarwal, C.C.: Data Streams: Models and Algorithms, Book. Yorktown Hieghts, NY 10598. Kluwer Academic Publishers, Boston/Dordrecht/London (2007)
10. Al Abd Alazeez, A., Jassim, S., Du, H.: EINCKM: an enhanced prototype-based method for clustering evolving data streams in big data. In: Proceedings of the 6th International Conference on Pattern Recognition Applications and Methods, ICPRAM, pp. 173–183 (2017)
11. Guha, S., Mishra, N., Motwani, R., O'Callaghan, L.: Clustering data streams. In: IEEE FOCS Conference, pp. 359–366 (2000)
12. Aggarwal, C., Han, J., Wang, J., Yu, P.: A framework for clustering evolving data streams. In: Proceedings of the 29th VLDB Conference, Germany, pp. 1–12 (2003)
13. Silva, J., Faria, E., Barros, R., Hruschka, E., Carvalho, A.: Data stream clustering: a survey. ACM Comput. Surv. (CSUR), 1–37 (2013)
14. Bandyopadhyay, S., Giannella, C., Maulik, U., Kargupta, H., Liu, K., Datta, S.: Clustering distributed data streams in peer-to-peer environments. J. Inf. Sci. **176**(14), 1952–1985 (2006)
15. Gao, X., Ferrara, E., Qiu, J.: Parallel clustering of high-dimensional social media data streams. arXiv, pp. 323–332 (2015)

16. Rodrigues, P.P., Gama, J., Pedroso, J.P.: Hierarchical clustering of time-series data streams. IEEE Trans. Knowl. Data Eng. **20**(5), 615–627 (2008)
17. Zhou, A., Cao, F., Yan, Y., Sha, C., He, X.: Distributed data stream clustering : a fast EM-based approach. 1-4244-0803-2/07/$20.00 ©2007, pp. 736–745. IEEE (2007)
18. Yeh, M.Y., Dai, B.R., Chen, M.S.: Clustering over multiple evolving streams by events and correlations. IEEE Trans. Knowl. Data Eng. **19**(10), 1349–1362 (2007)
19. Guerrieri, A., Montresor, A.: DS-means: distributed data stream clustering. In: Kaklamanis, C., Papatheodorou, T., Spirakis, Paul G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 260–271. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32820-6_27
20. Gama, J., Rodrigues, P.P., Lopes, M.L.: Clustering distributed sensor data streams using local processing and reduced communication. Intell. Data Anal. **15**(1), 3–28 (2011)
21. Talistu, M., Moh, T.S., Moh, M.: Gossip-based spectral clustering of distributed data streams. In: 2015 International Conference on High Performance Computing Simulation (HPCS), pp. 325–333 (2015)
22. Fu, T.Z.J., Ding, J., Ma, R.T.B., Winslett, M., Yang, Y., Zhang, Z.: DRS: dynamic resource scheduling for real-time analytics over fast streams. In: Proceedings of International Conference on Distributed Computing Systems, pp. 411–420, July 2015
23. Jin, C., Patwary, M.A., Agrawal, A., Hendrix, W., Liao, W., Choudhary, A.: DiSC: a distributed single-linkage hierarchical clustering algorithm using MapReduce. In: Proceedings of the International SC Workshop on Data Intensive Computing in the Clouds (DataCloud), pp. 1–10 (2013)
24. Bhatia, S.K., Louis, S.: Adaptive K-Means clustering. Am. Assoc. Artif. Intell. 1–5 (2004)
25. Chakraborty, S., Nagwani, N.K.: Analysis and study of incremental K-means clustering algorithm. In: Mantri, A., Nandi, S., Kumar, G., Kumar, S. (eds.) HPAGC 2011. CCIS, vol. 169, pp. 338–341. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22577-2_46
26. Stoica, I.: Trends and challenges in big data processing. Proc. VLDB Endowment **9**(13), 1619–1622 (2016)
27. Basanta-Val, P., Fernández-García, N., Wellings, A.J., Audsley, N.C.: Improving the predictability of distributed stream processors. Future Gener. Comput. Syst. **52**, 22–36 (2015)