Scalability of efficient parallel K-Means

Conference or Workshop Item

Published Version

Scalability of Efficient Parallel K-Means

David Pettinger and Giuseppe Di Fatta
School of Systems Engineering
The University of Reading
Whiteknights, Reading, Berkshire, RG6 6AY, UK
{D.G.Pettinger,G.DiFatta}@reading.ac.uk

Abstract

Clustering is defined as the grouping of similar items in a set, and is an important process within the field of data mining. As the amount of data for various applications continues to increase, in terms of its size and dimensionality, it is necessary to have efficient clustering methods. A popular clustering algorithm is K-Means, which adopts a greedy approach to produce a set of K-clusters with associated centres of mass, and uses a squared error distortion measure to determine convergence. Methods for improving the efficiency of K-Means have been largely explored in two main directions. The amount of computation can be significantly reduced by adopting a more efficient data structure, notably a multi-dimensional binary search tree (KD-Tree) to store either centroids or data points. A second direction is parallel processing, where data and computation loads are distributed over many processing nodes. However, little work has been done to provide a parallel formulation of the efficient sequential techniques based on KD-Trees. Such approaches are expected to have an irregular distribution of computation load and can suffer from load imbalance. This issue has so far limited the adoption of these efficient K-Means techniques in parallel computational environments. In this work, we provide a parallel formulation for the KD-Tree based K-Means algorithm and address its load balancing issues.

1. Introduction

Clustering [1] is a key process in the field of data mining, alongside classification, regression, association rule learning and others. Unlike classification, which arranges the data into predefined groups, clustering does not use predefined groups. The aim of clustering is to ascertain the most appropriate groupings of the input data. Furthermore, compared to classification, clustering is an unsupervised task, requiring no overseer, or overseeing process. Clustering works on the principle that in a given dataset, certain points can be considered similar, but different from other collections of points. Clustering can be defined by a need to minimise the intra-cluster distances, whilst simultaneously maximising inter-cluster distances.

One of the more popular methods of partitional clustering is K-Means. The K-Means algorithm is an iterative refinement process, where at each iteration the clustering assignments are updated, consequently changing the definition of the clusters. K-Means is a type of mean-squared error clustering, which uses a distortion measure to determine a convergence to a final result, and is classified as a variance-based clustering algorithm. The function of K-Means is to determine a set of K points, called centres or centroids, so as to minimise the mean squared distance from each data point to its nearest centre [10]. Naive K-means, often referred to as a 'brute force' approach, performs a 'nearest neighbour' query for each of the data points that are contained within the entire dataset.

A way to improve K-Means is to use multi-dimensional binary search trees (KD-Trees) [11], which allow very efficient nearest neighbour searches. KD-Trees have shown an average running time of $O(\log n)$ for nearest neighbour queries [4]. A KD-tree is a binary tree, where each node is associated with a disjoint subset of the data. A KD-Tree node may also contain summary information about the data it represents. The KD-tree can be used to store either the centroid centres or the input data vectors to improve the nearest neighbour search, which is at the core of the K-Means algorithm. The definitions of the centroids change at each iteration. If the KD-Tree is used to store the centroids, a new KD-Tree has to be computed at each iteration. For this reason it is more efficient to use the KD-Tree to store the input data vectors. In this case the tree is only computed once, during a pre-processing phase. KD-Trees have been adopted to improve the efficiency of the sequential K-Means algorithm in several works [9, 12, 14].

While a parallel formulation of the original K-Means algorithm [7] has been largely studied and adopted, no work has been done to provide an efficient parallel formulation of the sequential techniques based on KD-Trees. The main reason is that such approaches tend to have an irregular distribution of the computation load and would suffer from load imbalance. This issue has so far limited the adoption of the efficient sequential techniques in parallel computational environments. In this work, we provide a parallel formulation for the efficient K-Means algorithm based on KD-Trees and address its load balancing issues. Our experimental analysis shows that it is convenient to adopt the most efficient techniques for small and medium parallel environments (up to 64 computing elements). Load imbalance makes the adoption of these techniques in large scale systems particularly challenging.

The rest of the paper is organised as follows. Section 2 recalls the basic K-Means algorithm and introduces the notation adopted in the paper. Section 3 presents KD-Trees and their use for speeding up nearest neighbour queries in K-Means. In section 4 we introduce a parallel formulation of the efficient K-Means algorithm based on KD-Trees. Section 5 provides an experimental analysis of the proposed approach. Finally section 6 provides conclusive remarks and future research directions.

2 K-Means

Given a set of n data vectors in a d dimensional space and a parameter K which defines the number of desired clusters, K-Means determines a set of K vectors $\{m^k\}$, called centers or centroids to minimise the mean squared distance from each data point to its nearest centre. This measure is often called the square-error distortion.

A centroid is defined as a point at the geometric centre of a polygon. This polygon can be regular, or irregularly shaped. In the case that the polygon is irregularly shaped, the centroid is derived and weighted to approximate a 'centre of mass'. The centroid of the cluster k can then be defined as:

$$m^k = \left(\frac{1}{n_k}\right) \sum_{i=1}^{n_k} x_i^{(k)}, \qquad (1)$$

where n_k is the number of data points in the cluster k, and $x_i^{(k)}$ is a data point in the cluster k. The error for each cluster is the sum of a norm, e.g. the squared Euclidean distance, between each input pattern and its associated centroid.

The overall error is given by the sum of the squared errors for each cluster:

$$E = \sum_{k=1}^{K} \sum_{i=1}^{n_k} \left| x_i^{(k)} - m^k \right|^2 . \qquad (2)$$

Difficulties in square-error and other partition-based clustering methods are that there are no computationally feasible methods for guaranteeing that a given clustering minimises the total square-error. Based on this, the number of possible assignments, even for small numbers of patterns and clusters, quickly becomes enormously large. Therefore many clustering algorithms use iterative 'hill climbing' methods that can terminate when a specific condition is met [6]. These conditions include, and usually are:

- when no improvement can be made to the assignments,
- when the change in total squared error drops below a certain threshold, and
- when a predetermined number of iterations have been completed.

The K-Means algorithm works by first sampling K centres at random from the data points. At each iteration, data points are assigned to the closest centre and centers are finally updated according to (1). Although the centres can be chosen arbitrarily, the algorithm itself is fully deterministic, based on the starting centres.

K-Means performs a 'nearest neighbour' query for each of the data points of the input data set. This requires $(n \cdot K)$ distance computations at each iteration.

Several issues affect the effectiveness of the algorithm. The algorithm converges to a local minimum, not necessarily the global one. The quality of the clustering and the number of iterations depend on the initial choice of centroids. A poor initial choice will have an impact on both performance and distortion measure. Instead of several runs with random choices, Bradley and Fayyad [4] postulate a variant algorithm that uses several preliminary K-Means passes to provide the initial points for the next run of the algorithm. This adds computational overhead to the overall running, however it provides a better reduced error.

Moreover, the assumption that the number of clusters K is known a priori is not true in most explorative data mining applications. Solutions to an appropriate choice of K go from 'trial and error' strategies to more complex techniques based on Information Theory [13] and heuristic approaches with repeated K-Means runs with varying number of centroids [6].

For these reasons and for the interactive nature of data mining applications, several runs are often carried out with a different number of centroids and/or different initial choice. Thus, it is important to identify appropriate techniques in order to improve the efficiency of the core iteration step.

3 Improving K-Means with KD-Trees

A KD-tree [3] is a multi-dimensional binary search tree commonly adopted for organising spatial data. It is useful in several problems like graph partitioning, n-body simulations and database applications.

KD-Trees can be used to perform an efficient search of the nearest neighbour for classification. In the case of K-Means algorithm, the KD-Tree is used to optimise the identification of the closest centroid for each pattern. The basic idea is to group patterns with similar coordinates to perform group assignment, whenever possible, without the explicit distance computations for each of the patterns.

During a pre-processing step the input patterns are organised in a KD-tree. At each K-Means iteration the tree is traversed and the patterns are assigned to their closest centroid. Construction and traversal of the tree are described in the next two sections.

3.1 Tree Construction

Each node of the tree is associated with a set of patterns. The root node of the tree is associated with all input patterns. The data set is partitioned in approximately two equal sized sets, which are assigned to the left and right child nodes. The partitioning process is repeated until the full tree is built and each leaf node is associated to a single data pattern. A minimum leaf size can also be defined. This leads to an incomplete tree, where leaf nodes contain a minimum number of patterns greater than 1.

The partitioning operation is performed by selecting a dimension and a pivot value. Data points are assigned to the left child if the coordinate in that dimension is smaller than the pivot value, otherwise to the right child. The dimension for partitioning can be selected with a round robin policy during a depth first construction of the tree. This way the same dimension is used to split the data sets of internal nodes which are at the same tree level. An alternative method is to select the dimension with the widest range of values at each node.

The pivot value can be the median or the mid point. The median guarantees equal sized partitions with some computational cost. The computation of the mid point is faster but may lead to imbalanced trees.

In all our tests we have adopted a minimum leaf size of 50, the dimension with widest range and the median value.

KD-Tree nodes contain summary information which is exploited for bulk assignment operations during the traversal of the tree at each iteration. The information associated to each KD-Tree node is computed during the preprocessing step and includes:

- The set of patterns,
- Two boundary vectors (the range of values on each dimension),
- The dimension used for partitioning the data,
- The pivot value used for partitioning the data,
- The vector sum of the patterns $\sum_i x_i$,
- The scalar sum of squared normalised values $\sum_i \sum_j^d \parallel x_{i,j} \parallel^2$,
- The vector sum of the normalised patterns $\sum_i \parallel x_i \parallel$.

3.2 Traversal of the Tree

At each K-Means iteration the KD-Tree is visited to perform the closest centroid search for the input patterns. After all patterns have been assigned to their closest centroid, the centroids can be updated.

The traversal follows a depth first search (DFS) strategy. During the traversal a list of centroids (candidates) is propagated from the parent node to the child nodes. The list contains the centroids that might be closest to any of the patterns in the node. The root of the tree is associated to all patterns and the list of candidates is initialised with all centroids. At each node the list of candidates is filtered from any centroid that cannot be the closest centroid for any of the patterns in the data set of the node. Alternative geometric constraints can be used to provide such guarantees.

If during the traversal the list of candidates reduces to a single centroid, then the patterns can be assigned to it without an explicit computation of the distances between the patterns and the centroids. In this case the traversal of the subtree rooted at the node can be avoided (backtrack).

If the visit reaches a leaf node, the explicit distance computations must be performed. However, the distances are computed only between any pattern in the node and any centroid in the list of candidates. In this case, the performance gain depends on how many centroids have been filtered out from the list.

In both cases, early backtrack and leaf node visit, there is a potential reduction of the number of distance computations performed by the original K-Means algorithm.

The total number of distance operations can be significantly reduced at the cost of the preprocessing overhead for constructing the KD-Tree and of few extra distance computations during each iteration for the filtering of the centroids during the visit of the KD-Tree. The extra distance computations at each node depend on the number of centroids and not on the number of data points.

In general, the performance gain is more effective for data set with low inter-cluster similarity. Many centroids can be discarded from the search for the nearest neighbour when the groups of patterns are well separated and distant from each other.

Different variants have been proposed to filter the centroids at each KD-Tree node. They share the same general approach and differ in the criteria which guarantee that a centroid cannot be chosen as the closest one to any of the patterns in a KD-Tree node and, thus, can be removed from the list to be propagated in the subtree.

In all these works it has been shown that K-Means based on KD-trees can be more efficient in orders of magnitude than the original K-Means algorithm. The most general approach is described in [14]. In a preliminary analysis we have carried out, this approach has shown comparable or better performance than others [9, 12]. In all our tests we have adopted this approach.

4 Parallel K-Means

Parallel computing has been extensively proposed for increasing the efficiency of clustering algorithms [5, 6, 7]. In particular [7] proposes a straightforward implementation of the original K-Means algorithm for distributed memory systems, which is based on a master-slave approach and static data partitioning. The input patterns are partitioned in equal sized sets and distributed to the processes. Initial centroids are generated at the master and broadcasted to the other processes. Each node performs a K-Means iteration on the local data partition. At the end of each iteration a global reduction operation generates the updated centroid vectors and the next iteration can start. The computation terminates similarly to the sequential method as discussed in section 2. In an homogeneous environment this approach guarantees a perfectly balanced load among the processes. In the following we refer to this approach as $pKMeans$.

Little or no work has been done to provide an efficient parallel formulation of the sequential algorithms based on KD-Trees. These parallel algorithms are expected to suffer from load imbalance in contrast to the perfectly balanced approach in [7].

The parallel algorithm for K-Means based on KD-Tree follows an approach similar to [7]. The main difference is that a KD-Tree is constructed in a preprocessing step and is adopted in the core computation of each iteration to reduce the number of distance computations.

The efficiency of filtering the centroids at each KD-Tree node depends on the similarity (spatial aggregation) of the patterns within each local partition. If random data partitions are generated, as in [7], the filtering step would become less effective. And this effect would become more evident for larger number of processes. Appropriate data partitions can be generated by exploiting the KD-Tree. The master node builds an initial KD-Tree up to the level $log(p)$, where p is the number of processes. This generates p KD-Tree leaves with data partitions with a good spatial aggregation to be distributed to the processes. Each process independently builds a KD-Tree for the local data partition. The local tree is used to accelerate the computations in each local K-Means iteration. At the end of each iteration a global reduction phase is performed to update the centroids, similarly to [7].

The algorithm describe above guarantees equal sized data sets in each processing node. However, the computation is not expected to be balanced. The effectiveness of filtering centroids in KD-nodes depends on the inter-cluster similarity of the data in each local partition. In general we can expect different number of distance computations will be performed by the processes at each iteration. This approach adopts a static partition of the data and is indicated as $pKDKM1$.

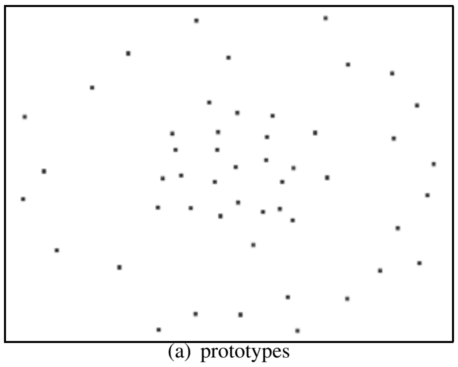
A second approach attempts to mitigate the potential load imbalance by adopting a parameter L_0 ($L_0 > log(p)$) for the level at which the initial KD-Tree is constructed. This generates more KD-Tree leaves than the number of processing elements. In this case each process receives $m = 2^{L_0}/p$ ($m > 1$) initial partitions and, thus, builds multiple local KD-Trees. The aim of this approach is to average the load at each process over many local KD-Trees. The partitions are still generated statically during the preprocessing step at the master node and we will refer to this approach as $pKDKM2$.

5. Experimental Analysis

In order to test the effectiveness of the two parallel approaches ($pKDKM1$ and $pKDKM2$) for KD-Tree K-Means we compare them with the original parallel K-Means algorithm ($pKMeans$) [7].

We have generated an artificial data set with 500000 patterns in a 20 dimensional space with a mixed Gaussian distribution as described in the following. First we have generated 50 pattern prototypes in the multi-dimensional space. This corresponds to the number of clusters K. For each cluster we have generated 10000 patterns with a Gaussian distribution around the prototype and with a random standard deviation in the range $[0.0, 0.1]$. In order to create a more realistically skewed data distribution, we did not generate the prototypes uniformly in the multi-dimensional space. We have distributed 25 prototypes uniformly in the whole domain and 25 prototypes were restricted to a subdomain. This generated a higher density of prototypes in the subdomain. The skewed distribution of data patterns in the domain emphasizes the load imbalance problem. The parameters were chosen in order to generate a dataset which contains some well separated clusters and some not well separated clusters.

We have applied Multi-Dimensional Scaling [8] to the set of prototypes and to a sample of the patterns to visualize them in 2-dimensional maps. Figure 1(a) shows the 50 prototypes and the higher density area is clearly visible in the center of the map. Figure 1(b) shows a 2D map of a sample of the generated patterns.


(a) prototypes

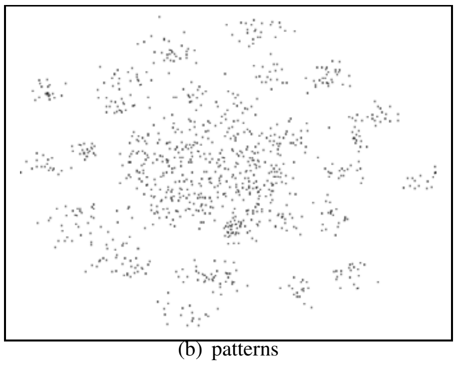

(b) patterns

Figure 1. 2D representation of the multi-dimensional prototypes (a) and patterns (b)

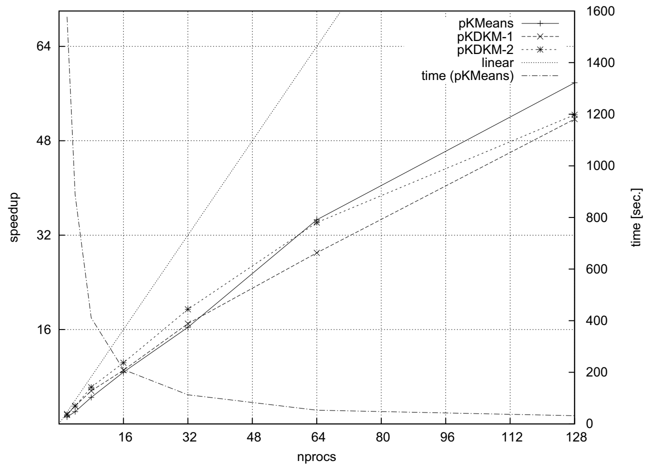


Figure 2. Speedup and running time

The software has been developed in Java and adopts MPJ Express [2], a Java binding for the MPI standard. The experimental tests were carried out in a IBM Bladecenter JS21 cluster (2.5GHz dual-core PowerPC 970MP) connected via a Myrinet network running Linux (2.6.16.60-0.42.5-ppc64) and J2RE 1.5.0 (IBM J9 2.3).

Figure 2 shows the relative speedup of the three parallel algorithms and the running time of $pKMeans$. For a small number of processes (up to 32) the algorithms $pKMeans$ and $pKDKM1$ have similar performance and $pKDKM2$ is slightly outperforming them. For 64 processes the algorithm $pKDKM1$ is showing worse performance than the other two, which have comparable speedup. For 128 processes both the approaches based on KD-Trees are now showing worse performance than the original K-Means algorithm. The worse performance of the KD-Tree based algorithms for larger computational environment can be explained by the load imbalance. It is evident that the $pKDKM2$ has reduced the problem, but has not completely solved it.

To analyse the performance of the algorithms in more detail, we show the contributions to the overall running time of the different parallel costs. We have accumulated the time spent by each process, respectively, in computation, communication and idle periods. Figure 3 shows the components of the overall parallel cost respectively for $pKMeans$, $pKDKM1$ and $pKDKM2$. The charts show that both KD-Tree based approaches have always higher idle time than the simple $pKMeans$. The advantage of using a more complex data structure is neutralised by the inefficient load balance. As expected, the communication time of the three methods are very similar. The computation time in both approaches based on KD-Trees tends to slightly increase with the number of processes. This effect is due to the extra computation introduced by the filtering criteria. In a larger environment the master node generates a deeper initial tree. The resulting local subtrees are smaller and rooted deeper in the initial tree. This makes the filtering criteria less effective by introducing additional distance calculations for redundant filtering. The effect is more evident in $pKDKM2$ where the initial tree is always constructed at a deeper level than in $pKDKM1$ ($L_0 > log(p)$).

6. Conclusions

We have presented a parallel formulation of the K-Means algorithm based on an efficient data structure, namely multi-dimensional binary search trees. While the sequential algorithms benefits from the additional complexity, the same is not true in general for the parallel approaches. Our experimental analysis shows that it is convenient to adopt the most efficient techniques for small and medium parallel environments (up to 64 computing elements). The cost of load imbalance still makes the adoption of these techniques unsuitable for large scale systems, where the simple parallel implementation of the K-Means algorithm will always provide a perfect load balance. However, this is valid only for dedicated homogeneous environments. We intend to test dynamic load balancing policies which could make the efficient techniques suitable also for large scale and heterogeneous environments. A further interesting direction of research is the optimisation of the communications requirements. At the moment, the global reduction operation hinders the adoption of any of these parallel K-Means algorithms in distributed environments, where high network latency would make the communication cost dominate the computation.

References

[1] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[2] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards thread safe Java HPC. *Proceedings of the IEEE International Conference on Cluster Computing*, 2006.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] P. S. Bradley and U. M. Fayyad. Refining initial points for k-means clustering. *Proceedings of Fifteenth Intl. Conference on Machine Learning*, pages 91–99, 1998.

[5] D. Foti, D. Lipari, C. Pizzuti, and D. Talia. Scalable parallel clustering for data mining on multicomputers. *Proceedings of the 15th IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 390–398, 2000.

[6] D. Judd, P. K. McKinley, and A. K. Jain. Large-scale parallel data clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):871–876, Aug. 1998.

[7] I. S. Dhillon and D. S. Modha. A data-clustering algorithm on distributed memory multiprocessors. *In Large-Scale Parallel Data Mining, Lecture Notes in Computer Science*, 1759:245–260, Mar. 2000.

[8] J.B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage University Paper series on Quantitative Applications in the Social Sciences, Beverly Hills and London: Sage Publications, 07-011, 1978.

[9] K. Alsabti, S. Ranka, and V. Singh. An efficient k-means clustering algorithm. *Proceedings of First Workshop on High Performance Data Mining*, 1998.

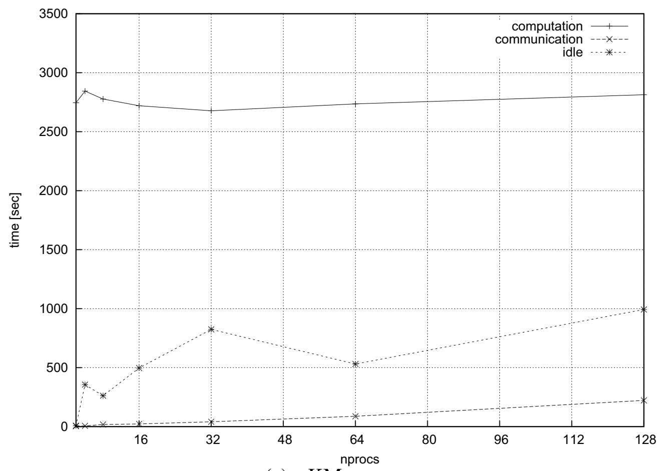
[10] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*.

[11] A. W. Moore. Efficient memory based learning for robot control. *PhD thesis*, 1991.

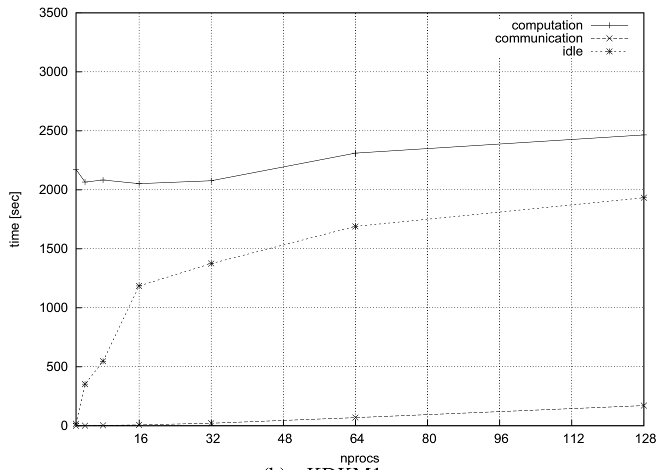
[12] D. Pelleg and A. Moore. Accelerating exact k-means algorithms with geometric reasoning. *Proceedings of Fifth ACM SIGKDD Intl. Conference on Knowledge Discovery and Data Mining*, pages 277–281, July 1999.

[13] D. Pelleg and A. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. *Proceedings of the Seventeenth Intl. Conference on Machine Learning*, pages 727–734, 2000.

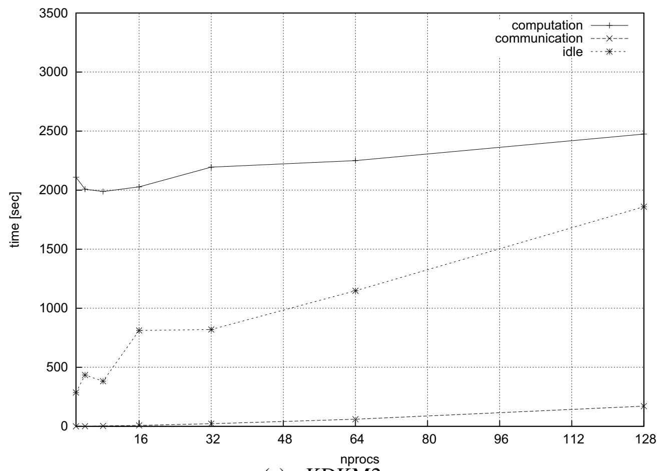
[14] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, July 2002.



(a) pKMeans



(b) pKDKM1



(c) pKDKM2

Figure 3. Parallel costs: cumulative times over all processes