University of Reading

Dynamic load balancing in parallel KD-tree k-means

Book or Report Section

Accepted Version

Di Fatta, G. and Pettinger, D. (2010) Dynamic load balancing in parallel KD-tree k-means. In: CIT '10: Proceedings of the 10th IEEE International Conference on Computer and Information Technology. IEEE, Washington DC, pp. 2478-2485. ISBN 978-0-7695-4108-2 doi: https://doi.org/10.1109/CIT.2010.424 Available at https://centaur.reading.ac.uk/6127/

It is advisable to refer to the publisher's version if you intend to cite from the work. See Guidance on citing.
Published version at: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5578280
To link to this article DOI: http://dx.doi.org/10.1109/CIT.2010.424

Publisher: IEEE

Publisher statement: (c) 2010 IEEE

All outputs in CentAUR are protected by Intellectual Property Rights law, including copyright law. Copyright and IPR is retained by the creators or other copyright holders. Terms and conditions for use of this material are defined in the End User Agreement.

www.reading.ac.uk/centaur

CentAUR

University of Reading

Central Archive at the University of Reading

Reading's research outputs online

Dynamic Load Balancing in Parallel KD-Tree k-Means

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

***Abstract*—One among the most influential and popular data mining methods is the k-Means algorithm for cluster analysis. Techniques 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 geometrical constraints and an efficient data structure, notably a multi-dimensional binary search tree (KD-Tree). These techniques allow to reduce the number of distance computations the algorithm performs at each iteration. 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 variants in parallel computing environments. In this work, we provide a parallel formulation of the KD-Tree based k-Means algorithm for distributed memory systems and address its load balancing issue. Three solutions have been developed and tested. Two approaches are based on a static partitioning of the data set and a third solution incorporates a dynamic load balancing policy.**

***Keywords*-Dynamic Load Balancing; Parallel k-Means; KD-Trees; Clustering;**

I. Introduction

One of the most popular methods of partitional clustering is k-Means [1], [2]. Given a set of N input patterns and a user-defined parameter K, it determines 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. 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 clustering method which adopts variance-based error criteria. It uses a distortion measure to determine a convergence to a final result.

The simple implementation of the k-Means algorithm is often referred to as a 'brute force' approach, because it performs a 'nearest neighbour' search among the K centres for each of the N data points of the dataset. Thus, it performs exactly $N \cdot K$ distance computations at each iteration.

A way to improve k-Means is to use multi-dimensional binary search trees (KD-Trees) [3], which allow very efficient nearest neighbour searches. 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 cache sufficient statistics about the data it represents. KD-trees for accelerating k-Means have been proposed to store the input data vectors to speed up the nearest neighbour search, which is the dominating computation at each iteration. KD-Trees have been adopted to improve the efficiency of the sequential k-Means algorithm in several works ([4], [5], [6]). They have shown a relevant reduction in the total number of distance computations for data sets with well separated clusters.

While a parallel formulation [7] of the brute force k-Means algorithm has been largely studied and widely adopted, little work has been done to provide an efficient and scalable parallel formulation of the sequential k-Means variants based on KD-Trees for distributed memory systems.

Parallel algorithms based on KD-Trees are expected to have an irregular distribution of the computation load and to suffer from load imbalance. This issue has so far prevented the adoption of these approaches in parallel computing environments.

A parallel k-Means algorithm based on quadtrees and limited to a two dimensional space was proposed in [8]. The work studied alternative static data decomposition techniques and compares their performance in a small scale computing environment (up to 24 processors). Although the work recognised load balance issues, it did not address them.

In this work, we provide a general, efficient and scalable parallel formulation for the k-Means algorithm based on KD-Trees for distributed memory systems. We show that, in general, it is not convenient to adopt the efficient techniques based on KD-Trees to speed up k-Means applications in a parallel computing environment without a dynamic load balancing policy. Our experimental analysis (up to 128 processors) shows that approaches based on static partitioning and static load balancing fail to match the scalability and the efficiency of the parallel brute force k-Means algorithm. On the contrary the proposed approach based on dynamic load balancing is able to address the load imbalance issue and to provide an overall better parallel efficiency.

The rest of the paper is organised as follows. Section II recalls the basic k-Means algorithm and introduces the notation adopted in the paper. Section III presents KD-Trees and their use for speeding up the convergence of the

k-Means algorithm. In section IV we introduce a parallel formulation of the efficient k-Means algorithm based on KD-Trees. Section V provides a comparative experimental analysis of the proposed approaches. Finally section VI provides conclusive remarks and future research directions.

II. The k-Means algorithm

Given a set $X = \{\vec{x}_1, \ldots, \vec{x}_N\}$ of N data vectors in a d dimensional space R^d and a parameter K ($1 < K < N$) which defines the number of desired clusters, k-Means determines a set of K vectors $C = \{\vec{c}_1, \ldots, \vec{c}_K\}$, called centers or centroids, to minimise the within-cluster variance. Without loss of generality, we assume the input vectors to be defined in the range $[0, 1]$ in each dimension.

The centroid of the cluster k is derived to approximate the 'centre of mass' of the cluster and is defined as:

$$\vec{c}_k = \left(\frac{1}{n_k}\right) \sum_{i=1}^{n_k} \vec{x}_i^{(k)}, \tag{1}$$

where n_k is the number of data points in the cluster k, and $\vec{x}_i^{(k)}$ is a data point in the cluster k. The error for each cluster is the squared sum of a norm $||\cdot||$, e.g. the Euclidean norm, between each input pattern and its nearest centroid.

The objective function that k-Means optimises is the the overall error (square-error distortion) $E(C)$ which is given by the sum of the squared errors for each cluster:

$$E(C) = \sum_{i=1}^{N} \min_{k=1..K} ||\vec{x}_i - \vec{c}_k||^2 = \sum_{k=1}^{K} \sum_{i=1}^{n_k} \left|\left|\vec{x}_i^{(k)} - \vec{c}_k\right|\right|^2. \tag{2}$$

A general problem in clustering methods is that there are no computationally feasible methods to find the optimal solution. In practical applications it is not possible to guarantee that a given clustering configuration minimises the square-error distortion, as it would require an exhaustive search. The number of possible clustering configurations, even for small numbers of patterns N and clusters K, quickly becomes enormously large. In [9] it has been shown that all distinct Voronoi partitions, which are equivalent to all possible clustering configurations, can be enumerated in $O(N^{Kd+1})$.

Therefore many clustering algorithms use iterative 'hill climbing' methods that can be terminated when specific conditions are met [10]. These conditions typically are when no further change can be made (local convergence), when the last improvement in the total squared error drops below a certain threshold, or when a user-defined number of iterations have been completed. However, only the first condition reaches a local minimum in the search space.

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, finally, centers are updated according to (1). Although the centres can be chosen arbitrarily, the algorithm itself is fully deterministic, once the number of clusters and the initial centres have been fixed.

At the core of each iteration k-Means performs a 'nearest neighbour' query for each of the data points of the input data set. This requires exactly $N \cdot K$ distance computations at each iteration. While the algorithm is guaranteed to converge in a finite number of iterations, this number can significantly vary for different problems and for different choice of the initial centres for the same problem. The overall time complexity of the worst case is often indicated as $O(INKd)$, where I is the number of iterations. However, unless explicitly bounded by the user, the number of iterations depends on the problem.

Only recently the theoretical bounds for the number of iterations k-Means can take to converge have been studied. In [11] it has been shown that k-Means requires more than a polylogarithmic number of iterations and derived polynomial upper and lower bounds on the number of iterations for simple configurations. In particular, for two clusters in one dimension the lower bound is $\Omega(N)$. The upper bound in one dimension and any number of clusters is $O(N\Delta^2)$, where Δ is the spread of the set X, i.e. the ratio between the longest and shortest pairwise distances. Authors in [12] provide a proof that the worst-case running time is superpolynomial and improved the best known lower bound from $\Omega(N)$ iterations to $2^{\Omega(\sqrt{N})}$.

The clustering configuration produced by k-Means is not necessarily a global minimum and it can be arbitrarily bad compared to the optimal configuration. Both the quality of the clustering and the number of iterations for convergence depend on the initial choice of centroids. It is common to run k-Means several times with different initial centroids (e.g. [13]). 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 [14] and heuristic approaches with repeated k-Means runs with varying number of centroids [10].

For these reasons and for the interactive nature of data mining applications, several k-Means runs often need to be carried out. Thus, it is important to identify appropriate techniques in order to improve the efficiency of the core iteration step, which is the aim of the techniques based on KD-Trees and described in the following section.

III. KD-Tree k-Means

A KD-Tree [15] 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 the k-Means algorithm, a 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 assignments, 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.

A. 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 and a binary partition is recursively carried out. At each node 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 (> 1) can also be defined which leads to an incomplete tree.

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.

An example of a KD-Tree for a data set in a two dimensional space is shown in Figure 1.

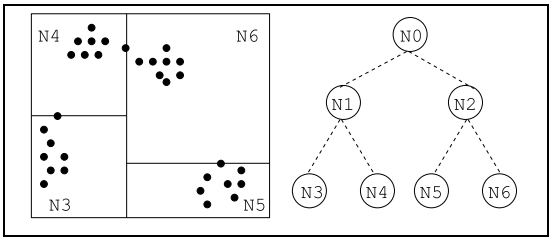


Figure 1. KD-Tree example in 2D

Each KD-Tree node stores parameters and statistics of the associated set of patterns. The information cached at each KD-Tree node is computed during the pre-processing step and includes:

- The set of patterns $\{\vec{x}_i\}$ associated to the KD-Tree node,
- The KD-Tree node parameters
  - The dimension used for partitioning the data,
  - The pivot value used for partitioning the data,
- The KD-Tree node statistics
  - The cardinality of the pattern set $|\{\vec{x}_i\}|$,
  - Two boundary vectors (the range of values at each dimension),
  - The vector sum of the patterns $\sum_i \vec{x}_i$,
  - The scalar sum of normalised patterns $\sum_i ||\vec{x}_i||^2$,

The dimension and the value for partitioning the data are used to generate the child nodes during the construction of the tree. The KD-Tree node statistics (aka 'Clustering Feature' [16]) are generated during the construction and exploited during the traversal for speeding up the computation of each iteration, as described in the following section.

B. 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 list of candidates at the root of the tree is initialised with all centroids. At the visit of a node the list of candidates is filtered from any centroid that cannot be the closest one for any of the patterns associated to the node. The excluded centroids are not propagated to the child nodes as, by definition, a child node's data set is a partition of the parent node's data set.

If during the traversal the list of candidates reduces to a single centroid, the traversal of the subtree rooted at the node is skipped (backtrack). In this case, all the patterns of the node can be assigned to the centroid without any further operation (bulk assignment). The explicit computation of the distances between these patterns and the centroids is entirely avoided. If the visit reaches a leaf node, the explicit distance computations must be performed. However, the distances are computed only between the patterns in the node and the centroids which have reached the leaf node. In this case, the performance gain depends on how many centroids have been filtered out from the complete list along the path from the root node to the leaf.

In both cases, early backtrack and leaf node visit, there is a potential reduction of the number of distance computations performed by the brute force k-Means algorithm. Moreover, in the case of early backtracking there is a further computation improvement. The statistics cached at the node are used to update the partial sum (bulk assignments) required to update the centroids at the end of the iteration.

The total number of distance operations can be significantly reduced at the cost of the preprocessing for constructing the KD-Tree and of some computation overhead

at each iteration. The computation overhead is due to extra distance computations for the filtering of centroids during the traversal. However, these extra distance computations are only proportional to the number of candidates (at most K) and the number of internal nodes which are visited.

In general, we expect that the performance gain is more effective for data sets with low inter-cluster similarity. When groups of patterns are well separated and distant from each other, more centroids are likely to be discarded during the traversal of the tree.

Different variants [4][5][6] have been proposed to identify the centroids to be filtered at each KD-Tree node. These techniques 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 assigned to a KD-Tree node. In all these approaches it has been shown that k-Means based on KD-trees can be more efficient in orders of magnitude than the brute force k-Means algorithm. In a preliminary analysis we carried out, [6] has shown comparable or better performance than [4] and [5], and has been adopted for our parallel formulation.

IV. Parallel k-Means

Parallel approaches have been extensively studied and adopted for increasing the efficiency of clustering algorithms (e.g. [17], [10], [7]). In particular, [7] proposes a straightforward implementation of the brute force 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 process performs a k-Means iteration on its 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 II. In the following we refer to this approach as *pkmeans*.
In a homogeneous environment *pkmeans* guarantees a perfectly balanced load among the processes. The number of patterns in each partitions is N/p, where p is the number of parallel processes. The number of distance computations at each process is exactly NK/p. In general, random data partitions are generated and it can be assumed that the patterns of different clusters are uniformly distributed in the partitions.

In this work we investigate parallel formulations of the k-Means algorithms based on KD-Trees. These parallel algorithms are expected to suffer from load imbalance in contrast to the perfectly balanced approach of *pkmeans*, as discussed in the following sections.

A. Parallel KD-Tree k-Means

The parallel algorithm for k-Means based on KD-Trees follows a computation-communication pattern similar to *pkmeans*. The main difference is that a distributed KD-Tree is constructed in a preprocessing step and is adopted in the core computation at each iteration to reduce the overall number of distance computations.

In general in the sequential k-Means based on KD-Trees, the effectiveness of the filtering operation depends on the similarity (spatial aggregation) of the patterns within the data set. The relative effectiveness of this operation in the parallel algorithm depends on how the data set is partitioned. If random data partitions are generated, as in [7], the filtering step would be less effective than in the sequential algorithm. Spatial aggregation of patterns must be preserved in the partitions. Appropriate data partitions can be generated by exploiting the KD-Tree itself.

An initial KD-Tree is built up to the level $log(p)$, where p is the number of processes. This generates p leaves with data partitions which have a spatial aggregation as good as in the sequential algorithm. The leaves of this initial KD-Tree define the data partitions to be distributed to the parallel processes (Figure 2).

In order to accelerate the pre-processing step, the construction of the initial KD-Tree can be distributed among the parallel processes. Nevertheless, a sequential construction of a KD-Tree is typically performed once for a given data set and can be used many times for sequential and for parallel algorithm executions. For this reason we did not include the pre-processing step in our experimental analysis. For very large data sets and for intrinsically distributed data sets the most appropriate parallel approach should be adopted [18].

Regardless of how the initial (global) tree is built, each process receives a node of the level $log(p)$, which defines the local data partition on which a complete local KD-Tree is built. The local trees are used to accelerate the computations in each local k-Means iteration similarly to the sequential algorithm. At the end of each iteration a global reduction phase is performed to update the centroids. This approach adopts the static partitioning (*SP*) defined by the KD-Tree. This approach is similar, in principle, to the best performing technique ('tree decomposition') proposed in [8].

The algorithm described above guarantees approximately equal sized data sets in each process when the median value is used during the construction of the tree. Nevertheless, the computation is not expected to be balanced. As mentioned above, the effectiveness of filtering centroids in KD-Tree nodes depends on the inter-cluster similarity of the data in the local partition. In general we can expect that partitions have different inter-cluster similarity. As a consequence, the number of distance computations performed by each process may vary in spite of the equal sized partitions. The level of the consequent load imbalance depends on the skewness

of the cluster distribution in the entire data set. Sets of points with non-uniform cluster distribution and low inter-cluster similarity are expected to produce higher levels of load imbalance.

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 initial tree and the partitions are generated statically during the preprocessing step at the master node and we will refer to this approach as Static Load Balancing (*SLB*).

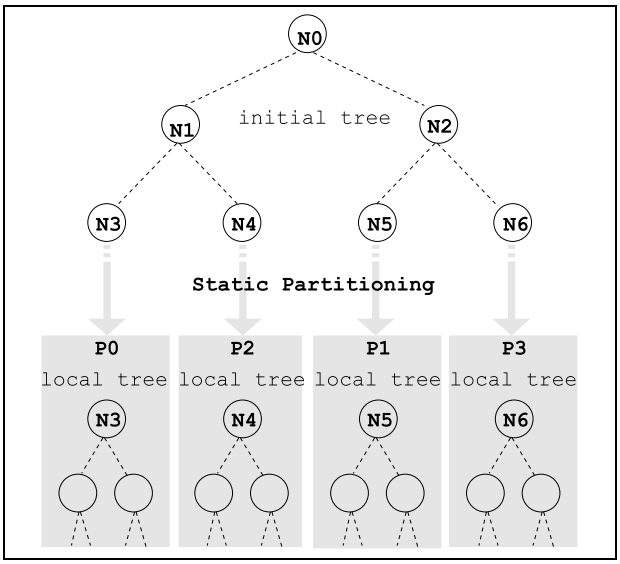


Figure 2. Static Partitioning of data based on KD-Tree

B. Dynamic Load Balancing

In general a static partition of the data set is very effective when the computational requirements are well defined in terms of the problem size. For example, the number of distance computations in *pkmeans* is exactly NK/p for each parallel process. In KD-Tree k-Means the proportionality of number of distance computations and number of patterns does not hold in general. Specific characteristics of the data set, such as inter-cluster similarity distribution, will determine the effectiveness of the optimisation techniques based on KD-Trees.

Since we cannot estimate the complexity of subtasks based on the size of the data partition, we adopt a Dynamic Load Balancing (DLB) policy, which monitors and eventually adjust the load distribution at the end of an iteration. Approaches based on global statistics are known to provide optimal load balancing performance, while they may limit scalability and introduce excessive overhead. We excluded an approach based on a centralised job pool, which represents a bottleneck and introduces excessive overhead. For the load donation among the parallel tasks we adopted a decentralized approach. While in order to maintain statistics of loads and to issue DLB orders at each iteration, we adopted a centralized approach. The scalability limitations due to this kind of centralised server are not considered an issue in this case. The algorithm already requires a global synchronization at the end of each iteration and is not suitable for very large-scale systems in wide area networks. A centralised collection of statistics adds a very limited overhead in the *AllReduce* operation.

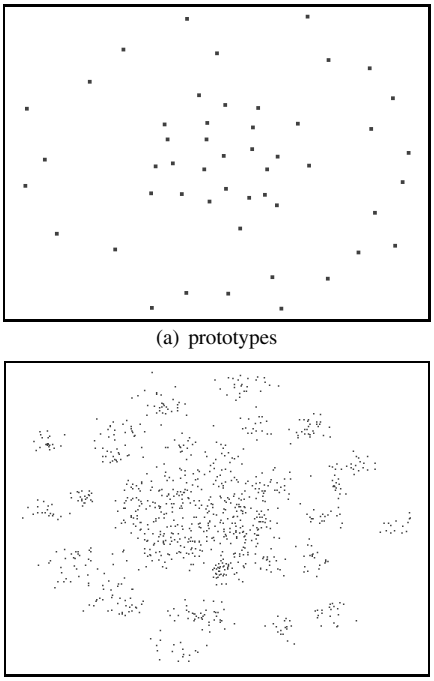
At the end of each iteration the master process collects the iteration time and the number of distance computations performed by each process. The master uses this information to generate and issue load balancing orders to processes. An order matches a donor with a receiver and communicates them the amount of work to be migrated. The actual exchange of work happens in a decentralised way, independently from other job donations.

In general, a DLB policy has to provide a mechanism to fairly distribute the load among the processors without introducing excessive communication and computation overhead. The generation of load donation orders must be minimised.

In KD-Tree k-Means the movement of the centroids at the end of each iteration has an impact on the load imbalance. However, it has been experimentally observed that these movements are monotonically decreasing in the number of iterations. Typically the centroids may change quite significantly only in the very first iterations. This can be exploited to minimise the generation of load donation orders and the communication and computation overhead of the DLB policy.

We have introduced a mechanism to activate/deactivate the DLB policy in the processes. In particular, we adopted a configurable timeout (number of iterations) to enable the DLB policy. The algorithm starts with the DLB policy disabled and a default value for the timeout (5 iterations). At each iteration the timeout is decremented and when it reaches zero the process synchronously activate the DLB policy. When the DLB is active, at the end of the iteration the master gathers the load statistics from the processes and, if necessary, computes and distributes DLB orders. While the DLB policy is active, the master node monitors and adjusts the distribution of the load in the system. When the overall load imbalance goes below a defined threshold ($maxLI$, default value is 1%), the master disables the DLB and sets a new timeout (default value is 10).

This relatively simple approach revealed to be quite effective in reducing the load imbalance without introducing excessive overhead. More complex algorithms may provide better performance in terms of scalability. Further research in this direction is, however, secondary to a complete decen-



(a) prototypes

(b) patterns

Figure 3. 2D representation of the multi-dimensional prototypes (a) and patterns (b)

tralized and asynchronous k-Means algorithm suitable for large scale systems in wide area networks.

V. Experimental Analysis

In order to evaluate the three proposed approaches for parallel KD-Tree k-Means, Static Partitioning (*SP*), Static Load Balancing (*SLB*) and Dynamic Load Balancing (*DLB*), we compare them with the parallel formulation [7] of the brute force k-Means algorithm (*pkmeans*).

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. We have generated 10000 patterns with a Gaussian distribution around each 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 is meant to emphasize 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 [19] to the set of prototypes and to a sample of the patterns to visualize

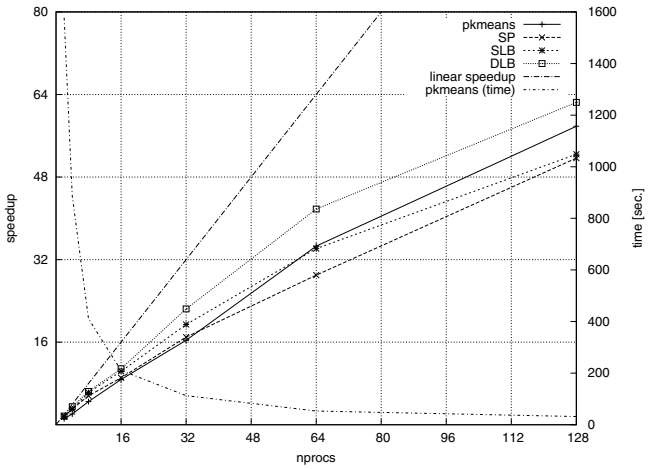


Figure 4. Speedup and running time

them in 2-dimensional maps. Figure 3(a) shows the 50 prototypes and the higher density area is clearly visible at the center of the map. Figure 3(b) shows a 2D map of a sample of the generated patterns.

The software has been developed in Java and adopts MPJ Express [20], a Java binding for the MPI standard. The experimental tests were carried out in a IBM Bladecenter 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).

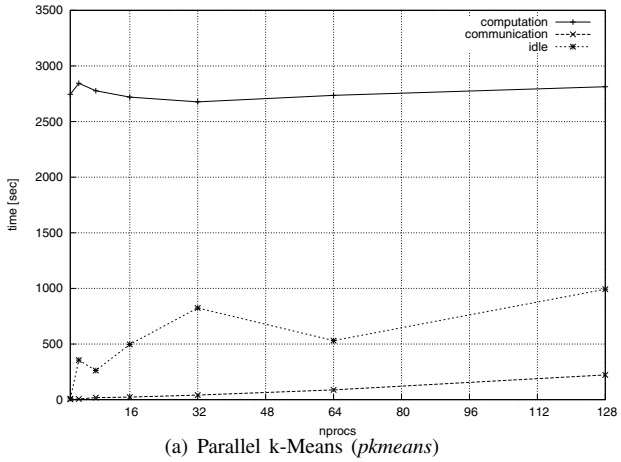
In all our tests we have built KD-Trees with a minimum leaf size of 50, attribute and value selection based, respectively, on the widest range and the median.

Figure 4 shows the relative speedup of the four parallel algorithms and the running time of *pkmeans*. For a small number of processes (up to 32) the algorithms *pkmeans* and *SP* have similar performance. For larger number of processes *pkmeans* outperforms *SP*. In spite of the more efficient data structure, *SP* shows clear scalability problems with respect to the parallel brute force approach.

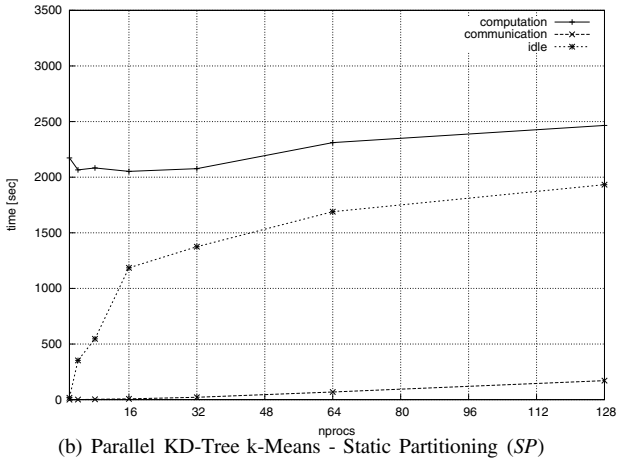
The second approach, *SLB*, is slightly outperforming *pkmeans* up to 32 processes. For 64 processes they have comparable speedup. For 128 processes *SLB* is also showing worse performance than the parallel brute force k-Means algorithm.

In the entire range of tests the algorithm *DLB* is outperforming all the others.

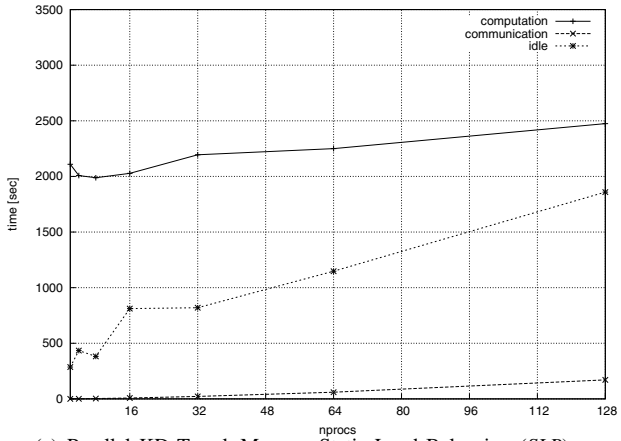
For computing environments with more than 32 processors the worse performance of the first two KD-Tree based algorithms (*SP* and *SLB*) is due to load imbalance. It is evident that the *SLB* has reduced the problem, but has not completely solved it. The third approach based on KD-Tree which adopts a dynamic load balancing policy is indeed able to overcome the problem and to make the use of more efficient data structure convenient for a parallel k-Means implementation.



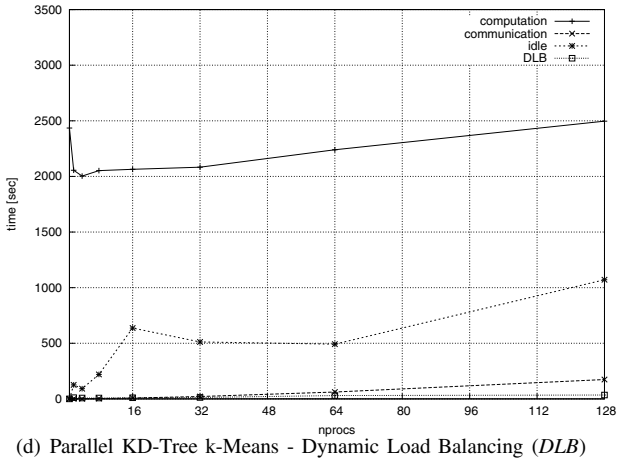
(a) Parallel k-Means (*pkmeans*)



(b) Parallel KD-Tree k-Means - Static Partitioning (*SP*)



(c) Parallel KD-Tree k-Means - Static Load Balancing (*SLB*)



(d) Parallel KD-Tree k-Means - Dynamic Load Balancing (*DLB*)

Figure 5. Parallel costs: cumulative times over all processes

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, idle periods and in performing dynamic load balancing for the algorithm *DLB*. Figure 5 shows the cumulative computation time and the components of the overall parallel cost. The charts confirm that *SP* and *SLB* spend less time in computation as effect of the better data structure and algorithm they adopt. Although, they always have higher idle time than the simple *pkmeans*. The advantage of using a more complex data structure is neutralised by the inefficient load distribution. However, the algorithm *DLB* is able to reduce the load imbalance and to provide an overall better parallel efficiency.

As expected, the communication times of all methods are very similar. The computation time in the 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 computing 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 filtering which are not present in the corresponding sequential algorithm. Local root nodes start with a full list of candidates, while in the sequential algorithm they get a sublist filtered along the path from the global root. We could start the filtering process at each parallel process at the global root, perform the filtering along the path to the local root as in the sequential case. Nevertheless, we would not solve the issue because this initial filtering would be redundantly repeated at each parallel process and would still introduce extra computation.

VI. 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 algo-

rithm benefits from the additional complexity, the same is not true in general for parallel approaches. Our experimental analysis shows that it is convenient to adopt the straightforward parallel approaches based on static partitioning and on static load balancing only for small parallel computing environments (up to 32 computing elements). The cost of the load imbalance in these two approaches makes their adoption unsuitable for larger scale systems, where the parallel implementation of the brute force k-Means algorithm still provides an almost perfect load balance. Although, this is valid only for dedicated homogeneous environments.

The dynamic load balancing policy adopted in our third approach has shown better scalability behaviour than the parallel brute force k-Means and is the best candidate k-Means algorithm for large-scale and heterogeneous computing environments.

An interesting direction of research is the optimisation of the communications costs. At the moment, the global reduction operation hinders the adoption of any of the parallel k-Means algorithms in large-scale distributed environments. Wide area network latency would make the communication cost dominate the computation and the global synchronization requirement would render the distributed application unreliable.

REFERENCES

[1] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1. University of California Press, 1967, pp. 281–297.

[2] S. P. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. IT-28, no. 2, pp. 129–137, Mar. 1982.

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

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

[5] 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*, pp. 277–281, Jul. 1999.

[6] 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*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[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*, vol. 1759, pp. 245–260, Mar. 2000.

[8] A. Gürsoy, "Data decomposition for K-Means parallel clustering," in *Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, 2003, pp. 241–248.

[9] M. Inaba, N. Katoh, and H. Imai, "Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering," in *SCG '94: Proceedings of the tenth annual symposium on Computational geometry*. New York, NY, USA: ACM, 1994, pp. 332–339.

[10] D. Judd, P. K. McKinley, and A. K. Jain, "Large-scale parallel data clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 871–876, Aug. 1998.

[11] S. Har-Peled and B. Sadri, "How fast is the k-means method?" in *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005, pp. 877–885.

[12] D. Arthur and S. Vassilvitskii, "How slow is the k-means method?" in *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*. New York, NY, USA: ACM, 2006, pp. 144–153.

[13] P. S. Bradley and U. M. Fayyad, "Refining initial points for K-Means clustering," *Proceedings of Fifteenth Intl. Conference on Machine Learning*, pp. 91–99, 1998.

[14] 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*, pp. 727–734, 2000.

[15] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[16] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," *SIGMOD Rec.*, vol. 25, no. 2, pp. 103–114, 1996.

[17] 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*, pp. 390–398, 2000.

[18] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka, "Parallel construction of multidimensional binary search trees," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 2, pp. 136–148, Feb. 2000.

[19] 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.

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