Parallelizing Clustering of Geoscientific Data Sets using Data Streams

Silvia Nittel
Spatial Information Science & Engineering
University of Maine
Orono, ME, USA
nittel@spatial.maine.edu

Kelvin T. Leung
Computer Science Department
University of California
Los Angeles, CA, USA
kelvin@cs.ucla.edu

Abstract

Computing data mining algorithms such as clustering on massive geospatial data sets is still not feasible nor efficient today. In this paper, we introduce a k-means algorithm that is based on the data stream paradigm. The so-called partial/merge k-means algorithm is implemented as a set of data stream operators which are adaptable to available computing resources such as volatile memory and processing power. The partial data stream operator consumes as much data as can be fit into RAM, and performs a weighted k-means on the data subset. Subsequently, the weighted partial results are merged by a second data stream operator. All operators can be cloned, and parallelized. In our analytical and experimental performance evaluation, we demonstrate that the partial/merge k-means can outperform a one-step algorithm by a large margin with regard to overall computation time and clustering quality with increasing data density per grid cell.

1 Introduction

Computing data mining algorithms such as clustering on *massive data sets* is still not feasible nor efficient today. Massive data sets such as raster satellite imagery are continuously produced by remotely sensing instruments with a data rate of over several TB/day. For instance, the NASA Earth Observation System's Information System (EOSDIS) collects data streams from over 20 different satellite instruments daily in large data archives, and the data is made available to diverse scientific communities for data analysis. Data analysis includes feature extraction or trend analysis using diverse data mining techniques.

Since data distribution to scientists represents a significant problem in itself, one approach to improve this situation is to apply statistical data compression algorithms to original data sets, and substitute them with compressed counterparts. Data compression has to be able to capture the high order interaction between the attributes of each multidimensional data point as well as their non-parametric distribution to provide high quality compressed data for further data analysis. Also, they should be easy to interpret by scientists. For example, a data sets can be partitioned into 1 degree x 1 degree (lat, long) grid cells, and each grid cell is compressed individually using a multivariate histogram representation per cell [6]. Multivariate histograms can be computed using the k-means clustering algorithm. Computing a multivariate histogram presentation for instance a *single global coverage* produced by the MISR instrument [23] (see Figure 1), 64,800 individual grid cells have to be clustered each of which can contain up to 100,000 data points with around 100 attributes. This example elucidates the fact that implementing data clustering efficiently and in a *highly scalable* way are necessary. Algorithms need to be scalable with regard to the overall data set size to reduce overall computation time, the number of data points within a single grid cell, the dimensionality of data points, and the utilization of available computing resources in order to achieve an overall high performance computation for massive data set.

1.1 Requirements from a Scalability and Performance Perspective

From a scalability and performance perspective, the following criteria are relevant designing a k-means algorithm for massive data sets:

Handling large numbers of data points: For massive data sets, an algorithm should be highly scalable with regard to the *number* and *dimensionality* of data points that need to be clustered. This includes that it should scale automatically from small to massive data sets.

Overall efficient response time: The algorithm should adapt to available resources such as computational resources and memory automatically, and maximize the utilization of resources in a greedy way.

High quality clustering results: The results of clustering should provide a highly faithful representation of

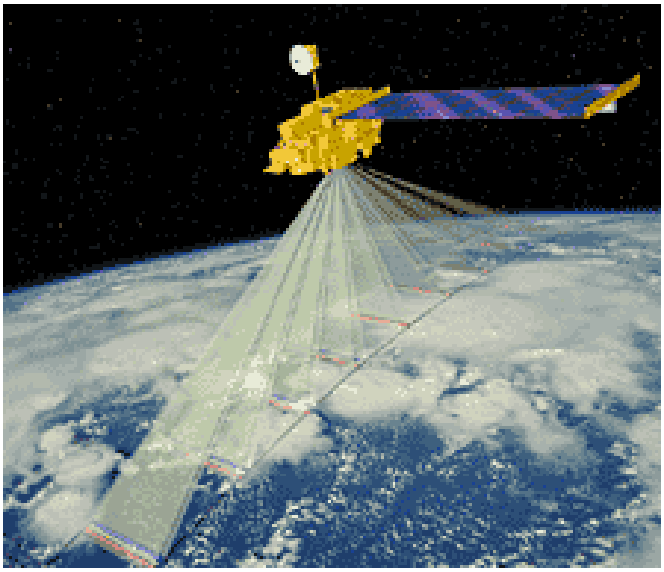

Figure 1. Swath of the MISR satellite instrument

the original data, and capture all correlations between data points.

1.2 Contributions of this Paper

In this paper, we propose a scalable implementation of the k-means clustering algorithm using data streams. Typically, a k-means implementation requires that all data points to be clustered are stored in main memory for computation. However, for massive data sets this is often not possible. Using a data stream paradigm and re-implementing k-means using data streams allows to scale clustering to varying and possible a very large number of points.

Lately, data streams and data streams operators have been successfully used for querying e.g. continuous credit card transactions, network traffic or sensor data streams. Data stream operators are based on the data flow paradigm. A stream operator consumes a fixed or varying number of data items from an incoming data stream, processes the data, and produces result data items which can be processed by other data stream operators. Several characteristics of data streams make this paradigm particularly useful and efficient for processing massive data sets which are stored as files:

1. Each data item from the data set stored on disk is read into main memory *only once* to avoid expensive, multiple data I/O.

2. An stream operator can store only a *limited amount* of state information in volatile memory.

3. There might be limited control over the *order of arriving data items*.

We introduce the *partial/merge k-means* algorithm which is implemented as a set of data stream operators that are adaptable to available computing resources such as volatile memory and processing power. The partial/merge k-means algorithm processes the overall set of points in 'chunks' that can be fit into main memory, and 'merges' the results of the partial k-means steps into an overall cluster representation. The partial k-means and the merge k-means are implemented as data stream operators that are adaptable to available computing resources such as volatile memory and processors by parallelizing and cloning operators. Our analytical and extensive experimental tests, we compare the scalability, performance and clustering quality of a traditional and a data-stream based k-means implementation. A traditional implementation assumes that the entirety of data points of a data cell are present in main memory for computation. The experimental results shows that the partial/merge k-means implementation scales in an excellent way with regard to the memory bottleneck, and produces clustering results that are of significantly higher quality than generated a traditional k-means algorithm, especially for large datasets.

The remaining paper is structured as follows: Section 2 states the problem in more detail and discusses related work. Section 3 contains the detailed description of the partial/merge k-means, and Section 4 includes implementation details of our prototype. Section 5 contains our experimental results. We conclude this paper in Section 6 with few remarks.

2 Clustering Massive Data Sets

Several methods of data compression in sparse high dimensional temporal-spatial spaces are based on *clustering*. Here, a data set is subdivided into temporal-spatial grid cells, and each cell is compressed separately. A grid cell can contain up to 100,000 n-dimensional data points. During clustering, grid cell's data points are partitioned into several disjunct clusters such that the elements of a cluster are similar, and the elements of disjunct clusters are dissimilar. Each cluster is represented by a specific element, called *cluster centroid*. For non-compression applications, the goal is to partition a data set into cluster; for compression, we are interested in representing the overall data set via the cluster centroids.

Two strategies can be used to identify cluster: a) finding densely populated areas in a data set, and dissecting the data space according to the dense areas (clusters) (density estimation-based methods), or b) using an algorithm that

attempts to find the clusters iteratively (k-means). In our work, we will focus on the k-means approach. In detail, the k-means algorithm can be formalized as described as follows. K-means is an iterative algorithm to categorize a set of data vectors with metric attributes in the following way: Given a set S of N D-dimensional vectors, form k disjoint non-empty subsets $\{C_1, C_2, \ldots C_k\}$ such that each vector $v_{ij} \in C_i$ is closer to mean(C_i) than any other mean. Intuitively, k-means algorithms can be explained with the following steps:

1. *Initialization*: Select a set of k initial cluster centroids randomly, i.e. m_j, $1 \leq j \leq k$.

2. *Distance Calculation*: For each data point X_i, $1 \leq i \leq n$, compute its Euclidean distance to each centroid m_j, $1 \leq j \leq k$, and find the closest cluster centroid. The Euclidean distance from vector v_j to a cluster centroid c_k is defined as $dis(c_k, v_j) = (\sum_{d=1,D}(c_{kd} - v_{jd})^2)^{1/2}$.

3. *Centroid Recalculation*: For each $1 \leq j \leq k$, computed the actual mean of the cluster C_j which is defined as $\mu_j = 1/|C_j| * \sum_{v \in C_j} v$; the cluster centroid m_j 'jumps' to the recalculated actual mean of the cluster, and defines the new centroid.

4. *Convergence Condition*: Repeat (2) to (3) until convergence criteria is met. The convergence criteria is defined as the difference between the mean square error (MSE) in the previous clustering iteration $I = (n-1)$ and the mean square error in the current clustering iteration $I = (n)$. In particular, we choose $\parallel (MSE(n-1) - MSE(n)) \parallel \leq 1 \times 10^{-9}$.

The quality of the clustering process is indicated by the *error function E* which is defined as

$$E = \sum_{k=1,K} \sum_{v \in C_k} \parallel \mu_k - v \parallel^2$$

where K is the total number of clusters.

The k-means algorithm iteratively minimizes this function. The quality of the clustering depends on the selection of k; also, the k-means algorithm is sensitive to the initial random choice of k seeds from the existing data points. In the scope of this paper, we assume that we are able to make an appropriate choice of k; however, we will vary on the selection of initial random seeds.

There are several improvements for step 2 that allow us to limit the number of points that have to be re-sorted; however, since this is not relevant with regard to the scope of the paper, we will not consider it further.

2.1 Problem Statement

Clustering temporal-spatial data in high dimensional spaces using k-means is expensive with regard to both **computational costs** and **memory requirements**. Table 1 depicts the symbols used in the complexity analysis of the algorithms.

N	Number of data points
I	Number of Iterations to converge
K	Total number of Centroids
G	Total number of grid cells
R	Total number of experiment run (with different initial k seeds)
p	Total number of chunks/partitions used in partial/merge K-Means

Table 1. Symbols used in complexity analysis of the algorithms.

Computing k-means via a **traditional, serial algorithm**, i.e. scanning a grid cell C_j at a time, and compressing it, and then scanning the next grid cell, all N data points belonging to one grid cell have to be kept in memory. The algorithm uses I iterations to converge, and it is run R times with R different sets of randomly chosen initial seeds. In this case, the memory complexity is $O(N)$, and the time complexity is $O(GRIKN)$ whereby G is the number of overall grid cells. Here, both the memory and the computational resources are bottlenecks for the serial k-means.

Two aspects to the memory bottleneck need to be considered: the volatile memory that is available via the virtual memory management, and the actual memory available via RAM. From a database perspective, control over RAM is essential to control any undesired paging effects. If the relevant data points do not fit into RAM and to avoid uncontrolled paging effects by the underlying operating system, the data set has to be broken up, and clustered incrementally. In the database literature, several approaches deal with the problem of large data sizes ([2, 5, 15, 19, 26]), a problem that is not considered in k-means algorithms used in statistics, machine learning, and pattern recognition. We will refer to this work in more detail in the related work section 2.2.

Parallel implementations of k-means deal with the bottleneck of *computational resources* when clustering large amounts of high dimensional data. A group of processors is utilized to solve a single computational problem. Several ways of parallelizing k-means using either massively parallel processor machines or networks of PCs can be considered[1].

As shown in Figure 2, **Method A** is a naive way of parallelizing k-means is to assign the clustering of *one grid*

[1] For price performance reasons, we consider shared-nothing, or shared-disk environments as available in networks of PC.

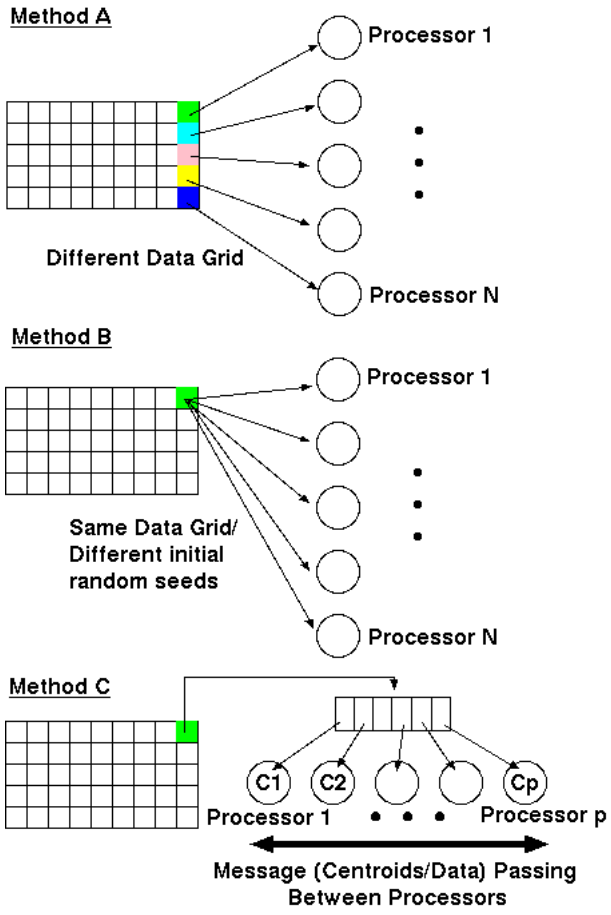

Figure 2. Several Ways of Parallelizing K-Means

cell each to a processor. **Method B** is the second approach to assign *each run* R_i of k-means on one grid cell C_j using one set of initial, randomly chosen k seeds to a processor. The third approach called **Method C** that reduces both the computational as well as the memory bottleneck is to divide the grid cell into disjunct subsets (*clusters*) by choosing a set of initial centroids, and sort all data points according to the centroids. The subsets are assigned to different slaves. Each slave computes the new mean for its subset, and broadcasts it to all other slaves. Each slave computes the Euclidean distance between the data points of its disjoint subset and all broadcasted means. The data points with $Min(d)(v, c_i)$ are sent to the slave that handles the centroid c_i. This approach utilized memory and computing resources efficiently for large data sets, however, it also introduced an overhead of message passing between the slaves. Overall, related work in the area does reports of near-linear scale-up and speed-up for parallelizing k-means ([14, 3]).

Parallelization adds the necessary options to deal with the computational bottlenecks of clustering, however, options mentioned above do not deal with the memory limitation appropriately. In each approach mentioned above, memory limitations still apply, since a well-defined set of points has to fit into memory (i.e. complete grid cells, or clusters). An approach is necessary that does not have *any* memory limitation, i.e. it is able to adapt to available memory in a flexible way independently if 1000, 20,000, 100,000 or 1 mio points have to be clustered. Also, the algorithm has to be able to exploit available memory and computational power optimally and in a greedy way.

2.2 Related Work

The related work that deals with the memory bottleneck or memory limitations of clustering large data sets can be found especially in the database literature. Here, the clustering techniques can be classified into two categories: *partitional* and *hierarchical*. In this paper, we focus on partitional clustering techniques since we partition a large temporal-spatial data set into predefined grid cells, and cluster each grid cell individually. In this case, related work that deals with the partitional clustering of large spaces such as CLARANS [26], and BIRCH [31] does apply only in a limited sense. These approaches consider the overall data set as the data that needs to be clustered, while in our case a very large number of grid cells with possibly a large number of high-dimensional data points has to be clustered very efficiently. Other work such as CLIQUE [2] is more similar to our work; this approach finds subspaces in high dimensional data sets that are clustered later on, and it focuses on identifying this subspaces. However, in our case, the subspaces are well-defined.

The work that is most closely related to our work is [8]. The approach uses a data stream approach to compute k-means on continuous data streams such as network data, or web site user click streams. The approach deals with similar restrictions as our work such as limited memory to store state; however, their data stream-based k-means implementation called LOCALSEARCH is designed to cluster one particular high dimensional space *continuously*. Although, LOCALSEARCH also computes data streams in as much data as can be fit in memory, there is no merge step with earlier results, an essential step in the partial/merge k-means implementation that allows to get an excellent cluster representation of an overall grid cell.

3 Implementing K-Means Using Data Streams

The paradigm of *data streaming* is used for different problems in database research. Traditionally, it has been used to implement operators in query execution plans and for parallel DBMS. Lately, it has been applied to processing continuous data streams, and answering queries on such data streams. The underlying *computational model* is the similar: a data stream based operator consumes one or more data items from an incoming data stream, processes the data, and produces an output data item which is stored in its output queue. The data item is immediately consumed by the next data stream operator; thus, all data stream operators process data in a pipelined fashion. The model allows to automatically clone operators to parallelize operators (see Figure 2).

Several restrictions apply for both processing continuous data streams as well as for massive data sets: to minimize I/0, data is scanned only once, and a data stream operator gets only *'one look'* at the data. Processing massive data sets, there is *little control over the order of incoming data items* since the data items necessary to cluster one grid cell are likely distributed over a large number of files. The most significant restriction is that each data stream operator has a *limited volatile memory* to store operator state. Processing massive data sets, the limitation of memory is due to the fact that aggregation operators such as $Average$ or a clustering algorithm needs to store state that is too large to fit into volatile memory. Massive data sets pose the challenge to implement algorithms via a set of data stream-based operators in a way that they automatically scale to available resources such as memory and CPUs, and exploit parallelism for overall performance, a problem that is not solved sufficiently today.

As shown in Figure 3, our algorithm consists of the following steps:

1. Scan the temporal-spatial grid cells.
2. Partial k-means on a subset of data points.
3. Merge k-means the results of step 2.

We discuss algorithms for each of these steps in this section.

3.1 Scanning massive temporal-spatial data sets

When massive temporal-spatial data sets are collected by satellite instruments such as MISR ([23]), the instrument covers 'stripes' of the earth while it flies over the earth, and the earth additionally rotates underneath the instrument (see Figure 1). Over a certain time period, the instrument achieves complete coverage of the earth; the data, however, is stored in a 'stripe'-wise manner in complex, semi-structured files. Pursuing the goal to compress and cluster such data via 1 deg x 1 deg grid cells, all data points belonging to the grid cell have to be available. Typically, these data points are scattered over several large files. For the scope of this paper, we do not consider the optimal I/O problem, since an optimal I/O strategy depends on the A) data layout with regard to spatial and temporal ordering of the data points in a file and B) the partitioning of the data into files. In this paper, we assumed that the data had been scanned once, and sorted into one degree latitude and one degree longitude grid buckets that were saved to disk as binary files. Furthermore, we assume that grid buckets are directly used as data input, and that all data points that belong to a grid cell arrive sequentially, and in random order.

3.2 Partial K-Means

The k-means algorithm is implemented via several stream-based operators. At the core are two operators that deal with the scalability of k-means. The first operator copes with the memory limitation for very large data sets. Instead of storing all data points $v_1, \ldots v_n$ of a grid cell C_s in memory, we divide the data of C_s into p *partitions* $P_1, \ldots P_p$ with the condition that all data points $v_1, \ldots v_m$ of partition P_j can be stored into available volatile memory (physical memory, not virtual memory). The stream operator *partial k-means* selects a set of random k seeds for a partition P_j, and performs a k-means on the subset of data points of the overall grid cell until the convergence criteria is met. This step is repeated for several sets of random k-seeds, and the representation with the minimal mean square error is selected to represent the clustering of partition P_j. The partial k-means operator produces a set of *weighted centroids* $c_{ij} \in P_j$ $\{(c_{1j}, w_{1j}), (c_{2j}, w_{ij}), \ldots, (c_{kj}, w_{kj})\}$. The weight w_{ij} is defined as the number of points that are assigned to the centroid c_{ij} in the converge step for partition P_j. The sum $\sum_{i=1\ldots k} w_{ij}$ is the number of points N_j in the partition P_j.

Each partition is clustered independently. We assume a fixed k is selected for the k-means in all partitions of the overall grid cell that is appropriate for the data distribution in the overall grid bucket. Since we assume that the *size* of partitions is defined by the available memory, each grid cell is partitioned into about equal-sized chunks of data that are clustered.

Executing k-means for a data partition the cluster result is still sensitive to the selection of the initial, randomly chosen seeds. Therefore, to improve the quality k-means can be run several times with different sets of initial seeds, and the representation producing the smallest mean square error is chosen for Step 3.

Space complexity: For a serial k-means, the memory space complexity is $O(N)$, where N is the number of data points of the data set to be clusters. For the partial step

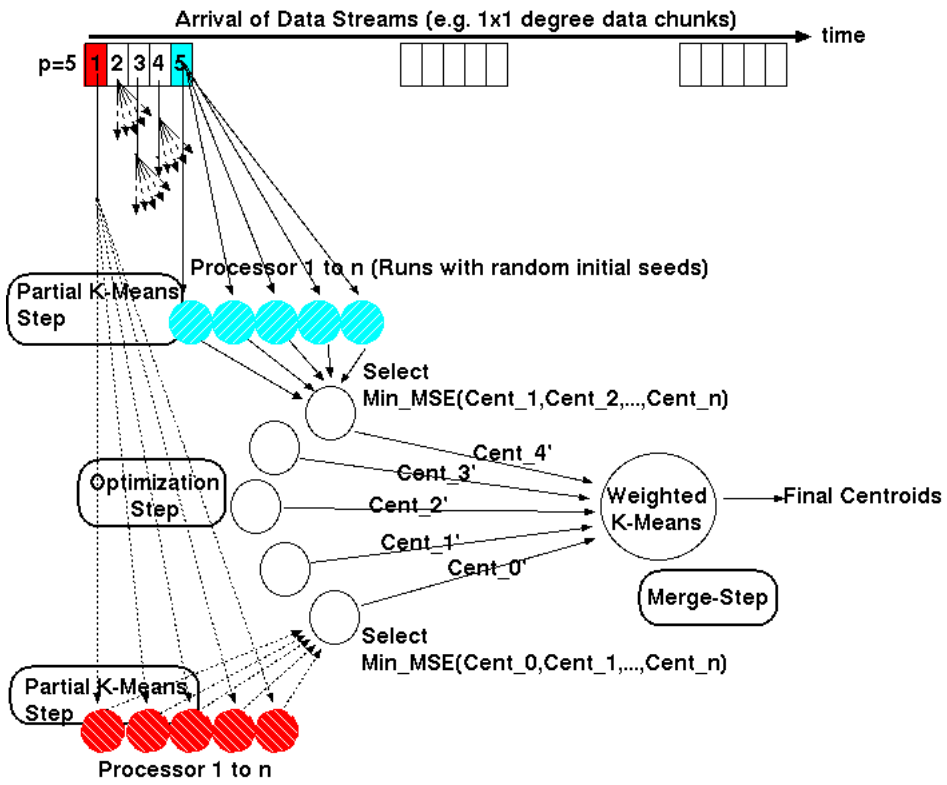

Figure 3. Diagram of Partial/Merge K-Means Algorithm

of partial/merge k-means, the space complexity is $O(N')$, where N' is the number of data points in a chunk, and $N' << N$. The space complexity for all partial steps is $O(N'p) = O(N)$, but partial steps can be run on different machines.

Time complexity: The time complexity of serial k-means is $O(NKI)$, where N is the number of data points, K the number of centroids, and I the number of iterations to converge. If N is large, then I increases exponentially. Using R sets of different seeds, the time complexity of serial k-means is $O(RIKN)$.

For the partial step of partial/merge k-means, the time complexity for a data partition P_j is $O(I'KN')$. K remains the same. N' is the number of data points in P_j, $N' = N/p$, thus, $N' << N$. I' is the number of iteration to converge k-means in P_j. Since $N' << N$, consequently $I' << I$ for each data partition.

The complexity of all partial k-means steps is $O(N'KI'p)$ whereby p is the number of partitions. Here, $O(N'KI'p) << O(NKI)$ for large N.

3.3 Merging K-Means

The *merge k-means* operator is the consumer operator for *partial k-means*. The operator basically performs another k-means using the set of all centroids that were computed in the partial k-means for all partitions $P_1, \ldots P_p$. There are several options to perform this second merge k-means: a) *incrementally*, or b) *collectively*. From a information theoretic perspective, the second approach is able to generate a more faithful representation of the original data set since the centroids of the chunk that are consumed first by the merge operator are not treated preferentially, and therefore would be involved in *all* consecutive incremental merge k-means runs for each new set of centroids. In the collective merge k-means, all sets of centroids $Centroids = \{Cent_0, \ldots Cent_p\}$ and $Cent_j = \{(c_{1j}, w_{1j}), (c_{2j}, w_{2j}), \ldots c_{kj}, w_{kj})\}$ have the same statistical chance to contribute to the overall representation. Each centroid is weighted with the count of data points that were assigned to it in the cluster representation for its partition. This way, the relative size of a partition also contributes to the merged cluster representation; a centroid that emerged in a large data partition, or a data partition with more data points automatically has a larger weight since a larger count

was assigned to it than to a centroid in a small partition.

In more detail, merge k-means computes the following algorithm: Given a set S of M D-dimensional centroids $\{(c_1, w_1), (c_2, w_2), \ldots (c_m, w_m)\}$ where $M = \sum_{p=1,\ldots P} k_p$ (k of each partition). Form k disjoint non-empty subsets $\{C_1, C_2, \ldots, C_k\}$ such that each centroid $(c_{ij}, w_{ij}) \in C_i$ is closer to mean(C_i) than any other mean.

- *1. Initialization*: Select the set of k initial cluster centroids z_i such that the weight w_i of z_i is one of the k largest weights in S.
- *2. Distance Calculation*: For each data point c_i, $1 \leq i \leq m$, compute its Euclidean distance to each centroid z_j, $1 \leq j \leq k$, and then find the closest cluster centroid. The Euclidean distance from a data point c_i to a cluster centroid z_k is defined as $dis(z_k, c_i) = (\sum_{d=1,D}(z_{kd} - c_{id})^2)^{1/2}$.
- *3. Centroid Recalculation*: For each $1 \leq j \leq k$, computed the actual, *weighted* mean of the cluster C_j which is defined as $\mu_j = 1/|C_j| * \sum_{c_i \in C_j} c_i * w_i$; the cluster centroid m_j 'jumps' to the recalculated actual weighted mean of the cluster, and defines its new centroid.
- *4. Convergence Condition*: Repeat (2) to (3) until convergence criteria is met; e.g. $\parallel (MSE(n-1) - MSE(n)) \parallel \leq 1 \times 10^{-9}$.

The quality of the overall partial/merge k-means algorithm is indicated by the *error function* E_{pm} which is defined as

$$E_{pm} = \sum_{k=1,K} \sum_{c_i \in C_k} ((\parallel \mu_k - c_i \parallel^2) * w_i)$$

where K is the total number of clusters.

The k-means algorithm iteratively minimizes this function. The selection of the data points with the largest weights as seeds in the initialization forces the algorithm to take into account which data points are likely to represent significant cluster centroids already. This would not be enforces if the set of seeds would be chosen randomly in this step.

Space complexity: The space complexity of the merge step is $O(Kp)$, whereby K is the number of weighted centroids from each partition, and p is the number of partitions.

Time complexity: The time complexity of the merge step is $O(I2Kp)$, whereby K is the number of points per partition, and p partitions are used as input to the merge step, thus, $K*p$ is the number of data points to be clustered. K is also the number of centroids in the merge step, and I is the number of iteration to converge.

Remarks: The partial/merge algorithm is designed to be less sensitive to the size of data partitions by taking weights into account. However, the question is still open which is the best choice of k depending on the partition size. For example, a small cluster can be represented with a smaller k than a large partition. Still, weighted centroids can be used in the merge step. Algorithms such as Entropy Constraint Vector Quantization (ECVQ) ([12, 6]) seem appropriate to deal with this problem. ECVQ-based algorithms do not fix the parameter k at the beginning of the k-means computation, but define a maximum k, and use a penalizing function that enforces that data points are more likely assigned to already large clusters. Therefore, some seeds might be starved, and can be discarded. This allows to find an optimal k for a partition on the fly.

3.4 Scaling and Parallelizing Partial/Merge K-Means

The partial/merge k-means algorithm can be considered as a data flow query specified in the form of a dataflow diagram (see Figure 3). Then, each leaf node in the dataflow query represents a collection of logical data objects, and non-leaf node represent logical operations that are applied to streams of data items. The data stream flows from leaf nodes through the non-leaf operator nodes to the root node which represent the result of the computation. Producer operator(s) and consumer operator(s) are connected via smart queues to avoid buffer overflow or underflow. Data flow in each stage of processing can be maintained with the objectives to minimize query response time as well as maximizing data throughput by fully utilizing the available computing resources.

Analyzing the computational expensive parts of the partial/merge k-means algorithm, the partial k-means operator is by far the most expensive computation since it sorts a very large number of high-dimensional points to a nearest centroid (ca. 5000 points, n=6). The merge operator as the consumer operator is likely to be idle most of the time. The partial k-means is the most likely operator candidate to be cloned by the query optimizer, and be sent for execution to different machines. Several options for parallelization can be considered.

Option 1 is to clone the partial k-means to as many machines a possible, and compute all k-means algorithms on the data partitions in parallel, and merge the results on one of the machines. This step improves the overall data throughput significantly. This data partitioning scheme has the advantage that the data for one data partition has to be sent to one machine only. *A second option* is to send a data partition to several machines at the same time, and perform partial k-means with a different set of initial seeds on each machine in parallel. We assume, however, that overall resources are exploited in a better way, if the first option is used, and several grid cells are computed at the

same time if more computing resources are available. A *third option* is to break up the partial k-means into several finer grained operators such as $ChooseRandomSeeds$, and $SortDataPoint$, $ComputeClusterMean$, etc. Within the partial k-means, the $SortDataPointSorting$ is the most expensive operation, and could be parallelized.

Implementing the partial/merge k-means via a data stream engine such as Conquest ([24]) the parallelization of the operators is performed automatically during query optimization when the logical data streaming query is compiled into a query execution plan, and a concrete physical implementation is chosen. The k-means operators themselves are unchanged, and the implementation automatically adapts to available resources and data set characteristics.

4 Implementation

A prototype of the partial/merge k-means was implemented using the CONQUEST system ([24, 27, 25]). Conquest is a stream-based query engine, and provides an extensible data model that allows introducing user-defined abstract data types, and operators ([25, 27]). Conquest is designed to handle complex queries involving computationally expensive calculations over large data sets in a massively parallel execution environment. The Conquest query execution environment is implemented in Java, and allows queries to be processed and migrated between heterogeneous machines running Conquest's query executors. Conquest includes a query re-optimizer [24] for dynamic adaptation of long running queries, but we did not exploit this component in the tests described in this paper. For more details on Conquest, see ([24, 27]).

The partial/merge k-means was implemented as a collection of user-defined data types and operators in Conquest. Several scan operators were used to read the data from HDF and binary files to populate $Gridbuckets$. As mentioned, the goal of the implementation was to provide a prototype for the overall concept; the implementation of both the partial/merge k-means and the serial k-means for the tests straight forward. In this version, we do not exploit many optimizations such as improved search mechanism for finding the nearest centroid of a data point.

5 Experimental Evaluation

The goal of the experimental evaluation is to compare the *scalability* of the partial/merge k-means with a serial implementation as well as the achieved *quality* of the clustering with a serial k-means that clusters all data points in the same iteration. Furthermore, we analyze the sensitivity of the quality of the merge k-means operator with regard to the *size*, and *number* of data partitions that are processed in the partial k-means.

pts/c	P	$t\ C_0 - C_i$	t merge	Min MSE	overall t
75K	10	1,954,614	74,364	15,680	2,028,978
	5	5,833,501	7,820	65,740	5,841,321
	1	–	–	105,020	5,908,854
50K	10				
	5	3,185,284	8,213	44,760	3,193,497
	1	–	–	69,441	4,032,755
25K	10	388,988	17,696	25,625	406,673
	5	788,098	8,440	22,600	796,538
	1	–	–	34,879	1,344,528
12,5K	10	148,142	13,216	16,109	161,358
	5	183,384	8,513	11,432	191,897
	1	–	–	17,086	471,651
2,5K	10	33,878	3,347	1,852	37,225
	5	30,019	9,679	4,906	39,698
	1	–	–	3,208	34,412
250	5	17,004	9,158	372	26,162
	1	–	–	359	3,251

Table 2: Comparing different partitioning sizes

5.1 Experiment Setting

To run experimental tests, we used a Conquest version that was implemented using JDK 1.3.1. We used four Dell Optiplex GX260 PCs each of which is equipped with a 2.8 GHz Intel Pentium IV processor, 1 GB of RAM, and a 80 GB hard disk. The PCs were connected using a Netgear GS508T GigaSwitch. The Conquest version was NSF mounted to all PCs; the data files for the test were stored on the local disk of each machine.

Data Sets: For our test, we created data sets that are based on a the EOS MISR (Multi-angle Imaging SpectroRadiometer) data set [23]. We recreated diverse 1 X 1 (lat, long) grid cells with the following characteristics:

- We varied the *number of data points per grid cell* between 250, 2,500, 5,000, 20,000, 50,000, 75,000 points,
- We used a fixed dimensionality for each data point which is six attributes, and
- We used a fixed k for all configurations $(k = 40)$.

A typical 1' x 1' grid cell of the MISR data set contains about 20,000 data points per grid cell for a monthly summary. We used the R statistical package to recreate the files with the same distribution, and created 5 different versions for each configuration.

Comparisons: In the test, we compared a serial implementation of k-means with two version of the partial/merge K-means. In the partial/merge test case, we distinguish a 5 split and a 10 split of an overall grid cell; here, the data points of a complete cell were randomly distributed over 5 or 10 'chunks' that served as input for the 5 and

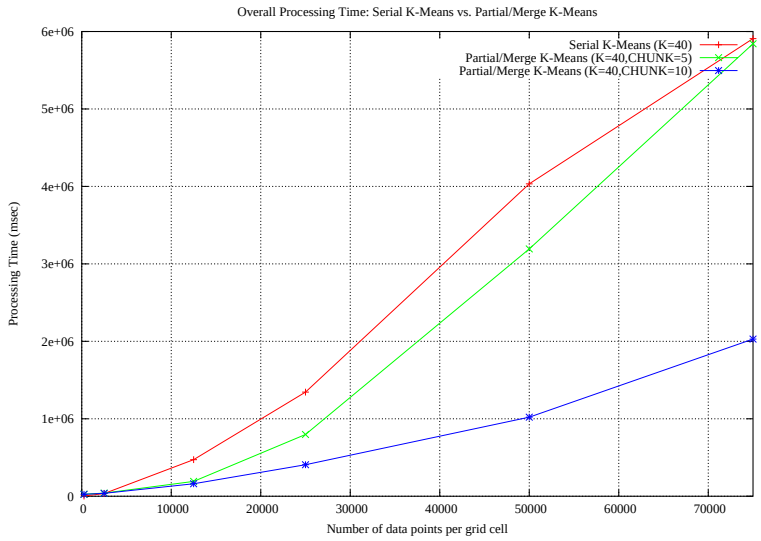

Figure 4. Execution time serial vs. partial/merge k-means

10 split case. From a temporal-spatial perspective, the areas of all 5 or 10 chunks were largely overlapping (¿90%). The code for the serial and the partial k-means implementation are identical besides that the partial k-means generates weighted centroids. For the serial implementation, we loaded the complete grid cell into (virtual) memory, and ran k-means until it converged. We documented the MSE, number of iterations until convergence, and overall computation time for each configuration.

For the partial/merge k-means we assumed an equi-size partitions of the data points of the overall grid cell. We ran the partial k-means on the data partitions, and the merge operator consumed the results and created the overall clustering for the grid cell. We tested two configurations:

- sensitivity of the clustering quality with regard to the number of partitions of the overall data of a grid cell (5split/10 split case), and
- speed-up of the processing if the partial k-means operators are parallelized, and run on different machines.

5.2 Benchmark Results

To assess the quality of the produced clusters, we computed the mean square error. In all cases, we calculated the MSE after convergence. In the serial case, we calculated the distance between the centroids and the *data points* in their cluster. In the partial/merge k-means, we computed the weighted distance between the final centroids and the *weighted data points* in their cluster (both 5-split and 10-split case). Since the MSE is sensitive to the data distribution in the grid cell as well as the initial seeds, we ran the serial k-means with 10 different sets of initial seeds, and selected the representation with the smallest MSE for comparison. In the partial/merge k-means, we computed each *partition* with 10 different sets of random seeds, and chose the best representation for the merge step. The computation time of each step of random seeds is included in the overall execution time. Typically, those steps would be executed in parallel, and thus, the execution time would be significantly smaller.

Comparing serial k-means with the partial/merge k-means with regard to overall execution time, it is evident that the computation time for the serial k-means is increasing exponentially with the number of data points per grid cell (see Figure 4). The overall execution time of the partial/merge k-mean, however, in most cases is significantly lower even if all partial k-means steps are run serially on one machine. The cluster quality between the different types of implementation is depicted in Figure 5. If a grid cell contains a very small number of points ($N < 250$), the partial/merge k-means and the serial k-means produce about the same quality of clustering, however, the execution time of the partial/merge k-means is about 10 times higher. At $N = 2,500$, the serial k-means is still performing slightly better than the 5-split partial/merge k-means with regard to

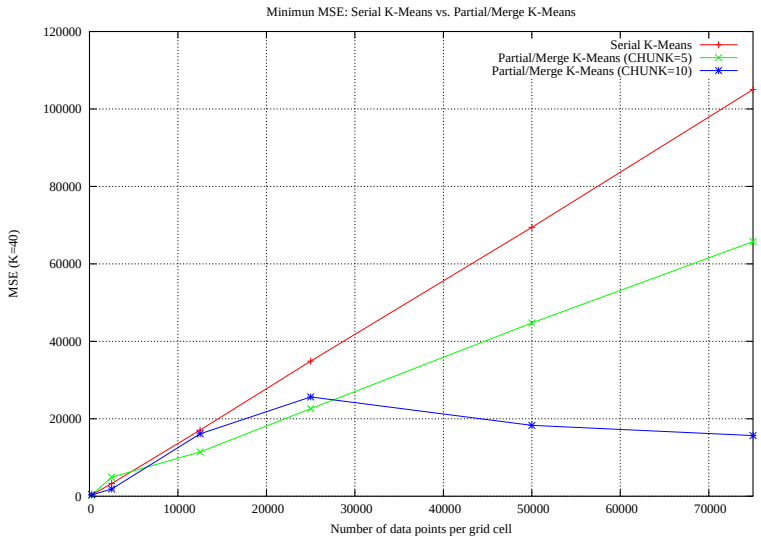

Figure 5. MSE serial vs. partial/merge k-means

MSE and the execution time which is now about the same for both. Splitting a 2,500 data point cell into 10 partitions (10-split), and using partial/merge k-means, however, produces a very low quality MSE, but it has about the same overall execution time as 5-split and serial k-means. We can conclude, that the clustering quality deteriorates if a limited-sized cell is sliced into chunks which are clustered independently. Here, all locality of clusters in the data set is lost.

In the remaining test cases, we can identify the following trend: at $N = 12,500$, partial/merge k-means breaks even, and the MSE and execution time even run serially on one machine as in our tests is significantly better than a serial k-means. The clustering quality for 5-split is about 30% better than for a serial implementation, and the execution time is 60% lower. The larger N becomes in the following cases, the better 10-split performs overall. With $N = 125,000$, the maximum size of points in a grid cell in our tests, 10-split produces a MSE of 15,680 while the serial k-means generates a clustering with a $MSE = 105,020$. The execution time for 10-split is about 30% of the execution time of the serial execution, and in this case the execution time for a 5-split is almost as high as the serial k-means execution (if run serially on one machine).

Overall, it can be concluded that the following strategies apply for using partial/merge k-means: a data sets does need to have a minimum number of data points for a partial/merge k-means being of advantage in (our case with k=40, it was $N = 500$) . Second, the larger the number of data points in the overall data set, the more effective it is to partition the data set into smaller subsets to improve MSE and execution time. Intuitively, the results can be explained that partial/merge k-means (or a traditional k-means implementation) performs best if the proportion of centroids to overall data points is well proportioned. We can exploit this fact for choosing an optimal 'chunk' size of a overall grid cell for the partial/merge k-means. In our experiments for instance for the optimal chunk size for $k = 40$ is ≤ 5000 data points per partition. Increasing the number of data points for a partition, the random pick of seeds is less likely to be representative. More results can be found in [7].

Overall, we can conclude that the partial/merge stream-based k-means implementation provides a scalable implementation with an excellent overall execution time, and produces cluster representation that outperform the results of clustering data in high dimensional space directly. This can be attributed to the fact that it is simpler to find an appropriate cluster representation with a large number of k for a small data set than for a large data set with the same k. Since the k is chosen randomly, it is more likely to find more representative seeds in a smaller data set.

6 Conclusions and Future Work

In this paper we presented the partial/merge k-means as a highly scalable approach to clustering massive data sets

using a data stream paradigm. We showed that the partial/merge k-means is adaptable to available computing resources such as volatile memory and processors by computing k-means on data partitions that fit into memory, and merging the partial results, and by parallelizing and cloning operators. In our analytical and experimental evaluation, we compared the scalability, performance, and clustering quality of the partial/merge k-means with a serial implementation. The results show that the partial/merge k-means provides a highly scalable and parallel approach, and outperforms a serial implementation by a large margin with regard to overall computation time and achieving a significantly higher clustering quality. Future work includes experimentations with different 'slicing' strategies of especially spatial grid cells into sub partitions; for example, data cells can be partitioned into spatially non-overlapping subcells, or a mostly overlapping cells as in our test cases. Since temporal-spatial phenomena do have spatial clustering characteristics, a well-chosen partitioning strategy will improve overall clustering quality and applicability.

Acknowledgements

This work was partially supported by the National Aeronautics and Space Agency under NASA grant number NCC5-30, and the National Science Foundation under NSF grant number EPS-9983432.

References

[1] A. Aboulnaga, and S. Chaudhuri, *Self-tuning Histograms: Building Histograms Without Looking at Data*. in Proc. of the 1999 ACM SIGMOD Intern. Conf. on Management of Data, June 1999.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, *Automatic subspace clustering of high dimensional data for data mining applications*. In Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data, Seattle, Washington, June 1998.

[3] R. Baraglia, D. Laforenza, S. Orlando, P. Palmerini, and R. Perego, *Implementation issues in the design of I/O intensive data mining applications on clusters of workstations.* In Proc. of the 3rd Workshop on High Performance Data Mining, Lecture Notes in Computer Science, 2000.

[4] S. Berchtold, H.V. Jagadish, K.A. Ross, *Independence Diagrams: A Technique for Visual Data Mining*, Proc. 4th Intl. Conf. on Knowledge Discovery and Data Mining, New York City, 1998, pp. 139-143.

[5] P. S. Bradley, U. Fayyad, and C. Reina, *Scaling Clustering Algorithms to Large Databases*, Proc. 4 th International Conf. on Knowledge Discovery and Data Mining (KDD-98). AAAI Press, Aug. 1998.

[6] A. Braverman, *Compressing Massive Geophysical Data Sets Using Vector Quantization*, Journal of Computational and Graphical Statistics, March 2002.

[7] A. Braverman, E. Fetzer, A. Eldering, S. Nittel, K. Leung, *Semi-Streaming Quantization for Remote-Sensing Data*, Journal of Computational and Graphical Statistics, Special Issue on Massive Data Streams, Volume 12 Number 4 Issue Dec 2003.

[8] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani, *Streaming-data algorithms for high-quality clustering*, Proceedings of IEEE International Conference on Data Engineering, March 2002.

[9] S. Chaudhuri, R. Motwani, and V. Narasayy, *Random sampling for histogram construction: how much is enough?*, In Proc. of Int. Conf. Management of Data (SIGMOD), Seattle, Washington, June 1998, pp. 436–447.

[10] J. Chattratichat, J. Darlington, M. Ghanem, Y. Guo, H. Huning, M. Kohler, J. Sutiwaraphun, H. W. To, and D. Yang, *Large scale data mining: The challenges and the solutions*, In Proceedings of Third International Conference on Knowledge Discovery and Data Mining (KDD-97), August 1997.

[11] C. Cheng, A. W. Fu, and Y. Zhang. *Entropy-based subspace clustering for mining numerical data*, In Proceedings of Internationl Conference on Knowledge Discovery and Data Mining (KDD'99), 1999.

[12] Chou, Lookabaugh, and Gray, *Entropy-constrained Vector Quantization*, IEEE Transactions on Speech, Acoustics, and Signal Processing, Vol. 37, 1989.

[13] A. Deshpande, M. Garofalakis, R. Rastogi, *Independence is Good: Dependency-Based Histogram Synopsis for High-Dimensional Data*, In Proceedings of Internatational Conference ACM SIGMOD, Santa Barbara, May 2001.

[14] S. Dhillon and D. S. Modha. *A data clustering algorithm on distributed memory machines*, In Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence, 2000, pp. 245-260.

[15] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu., *Incremental clustering for mining in a data warehousing environment*, In Proc. of 24th International Conference on Very Large Data Bases (VLDB), 1998.

[16] V. Ganti, J. Gehrke, R. Ramakrishnan. *Mining Very Large Databases*. In IEEE Computer, Volume: 32 Issue: 8, pp. 6875, Aug. 1999.

[17] J. A. G. Gendrano, B.C. Huang, J.M. Rodrigue, B. Moon, and R.T. Snodgrass, *Parallel algorithms for computing temporal aggregates*, In Proc. of the International Conference on Data Engineering, Sydney, Australia, March 1999, pp. 418–427.

[18] P. B. Gibbons, Y. Matias, and V. Poosala, *Fast incremental maintenance of approximate histograms*, Proc. of the 23rd International Conference on Very Large Databases, August 1997, pp. 466-475.

[19] S. Guha, R. Rastogi, and K. Shim, *CURE: An efficient clustering algorithm for large databases*, In Proceedings of ACM SIGMOD International Conference on Management of Data, New York, 1998, pp. 73–84.

[20] M. Halkidi M, M. Vazirgiannis, *A data set oriented approach for clustering algorithm selection*, Proceedings of PKDD, Freiburg, Germany, 2001.

[21] Y. Ioannidis and V. Poosala, *Balancing histogram optimality and practicality for query result size estimation*, In Proceedings of ACM SIGMOD Conference, May 1995, pp. 233–244.

[22] H.V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K.C. Sevcik and T. Suel, *Optimal Histograms with Quality Guarantees*, In Proceedings of 24th International Conference on Very Large Data Bases (VLDB), New York City, August, 1998, pp. 275-286.

[23] MISR (Multi-Angle Imaging SpectroRadiometer) Home page, http://www-misr.jpl.nasa.gov/.

[24] K. Ng, Z. Wang, R.R. Muntz, and S. Nittel, *Dynamic Query Re-Optimization*, International Conference on Scientific and Statistical Databases (SSDBM99), Cleveland, Ohio, July, 1999.

[25] K. Ng, and R.R. Muntz, *Parallelizing User-Defined Functions in Distributed Object-Relational DBMS*, International Database Engineering and Applications Symposium, Montreal, Canada, August, 1999.

[26] R. T. Ng and J. Han, *Efficient and Effective Clustering Methods for Spatial Data Mining*, In Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, pp. 144–155.

[27] S. Nittel, K. Ng, and R. R. Muntz, *Conquest: CONcurrent Queries over Space and Time*, International Workshop "Integrated Spatial Databases: Digital Images and GIS" (ISD'99), Portland, Maine, June, 1999.

[28] B. Moon, I. F. V. Lopez, and V. Immanuel, *Scalable algorithms for large-scale temporal aggregation*, Technical Report TR 98-11, Tucson, AZ 85721, November 1998.

[29] N. Ramakrishnan and A.Y. Grama. *Mining Scientific Data*. Advances in Computers, Vol. 55, 2001, pp. 119–169.

[30] W. Wang, J. Yang, and R.R. Muntz, *STING: A statistical information grid approach to spatial data mining*, In Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997, pp. 186–195.

[31] T. Zhang, R. Ramakrishnan, and M. Livny, *BIRCH: An Efficient Data Clustering Method for Very Large Databases*, In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 103–114.