Parallel K-Means

Contents

1 Introduction to clustering and distributed Batch K-Means

Clustering is the task of unsupervised classification that assigns a set of objects into groups such that each group is composed of similar objects. Clustering has been widely used for more than half a century as a resume technique to build a simplified data representation and is one of the primary tools of unsupervised learning. It plays an outstanding role in several pattern analysis, decision making, proximity exploration or machine-learning situations including text mining, pattern or speech recognition, web analysis and recommendation, marketing, computational biology, etc.

Two main clustering questions have been addressed in many contexts, reflecting the clustering's broad utility. The first question deals with the similarity notion between objects. In general, pattern proximity is based on some similarity measure (or distance function) defined for any pair of data elements. Such a similarity measure is of prime interest since it totally defines the proximity notion and therefore need to reflect what the user considers as close or similar. When the data set is contained in a vectorial space, a simple euclidean distance is often used. In many other situations, a more specific similarity measure is used, such as in the context DNA microarrays exploration. In other cases, clustering output is used as the training input of a set of supervised learning functions. This latter case, often referred as clusterwise linear regression (see for example [10]), is of prime interest for Lokad since the timeseries clustering is used to produce groups of timeseries that are processed group by group. From a computation point of view, the complexity of the similarity measure has a significant impact on the computational cost of clustering.

The second clustering aspect that has been widely addressed is the algorithm choice. For a given similarity measure, one can define a performance criterion (often referred in the literature as a loss function) for a clustering result to quantify how close is each point to its assigned cluster. The problem of returning the optimal clustering solution in regard to this loss function is reputed to be in many cases computationally untractable (see for example [18] in the case of the K-Means). Many algorithms are devised to return approximations of the optimal solution, often by converging to a local minimum of the loss function. We refer the reader to [14] for an in-depth clustering algorithms review. Some well-known techniques of non-hierarchical clustering are Batch K-Means, Vector Quantization (VQ), Neural Gas, Kohonen Maps, etc.

This chapter focuses on the Batch K-Means clustering algorithm. The choice of Batch K-Means is motivated by several reasons. Firstly, while better clustering methods are available, Batch K-Means remains a useful tool, especially in the context of data summarization where a very large data set is reduced to a smaller set of prototypes. As such, Batch K-Means is at least a standard pre-processing tool. Secondly, Batch K-Means has a low processing cost, proportional to the data size and the number of clusters. Thus, it is a good candidate for processing very large data sets. Finally, apart for the number of iterations to convergence, the processing time of Batch K-Means depends only on the data dimensions and on K, rather than on the actual data values: timing results obtained on simulated data apply to any data set with the same dimensions.

Distributed computing in Machine Learning or Data Mining arise when the computation time to run some sequential algorithm is too long or when the data volume is too big to fit into the memory of a single computing device. Such algorithms have already been successfully investigated for fifteen years (see for example [1], [5], or [13]) and applications build on top of these distributed algorithms are presently used in a wide range of areas, including scientific computing or simulations, web indexing applications such as Google Search, social network exploration applications such as Facebook, sparse regression in computer vision, etc. Parallelization is today one of the most promising ways to harness greater computing resources, whereas data sets volume keep growing at a much faster rate than sequential processing power.

In this chapter, we investigate parallelization of Batch K-Means over different computing platforms. Batch K-Means is known to be easy to parallelize on shared memory computers and on local clusters of workstations: numerous publications (see e.g., [5]) report linear speedup up to at least 16 processing units (which can be CPU cores or workstations). There are two reasons why Batch K-Means is reputed to be an algorithm suited for parallelization. The first reason is that distributed Batch K-Means produces exactly the same result than sequential Batch K-Means. For algorithms where the sequential and the distributed versions produce different results, it is necessary to confront the two algorithm versions on both speed-up and accuracy criterion. In the case of Batch K-Means, the comparison of the sequential and distributed versions is bounded to the speed-up criterion. In addition, the exact matching of the two algorithm version results provides an easy mechanism to guarantee that the distributed algorithm version is correctly implemented. From an engineering point of view, this property ease a lot the development process.

The second reason why distributed K-Means is reputed to be easy to parallelize is that it has been claimed to be an embarrassingly distributed algorithm. Most

of the computation time is spent in evaluating distance between data points and prototypes. These evaluations can be easily distributed on multiple processing units. Distributed K-Means can then be viewed as an iterated Map-Reduce algorithm: firstly, the data set is separated into M subsets of equal length (where M is the number of processing units), each processor (mapper) being responsible of computing distances between its data points and the prototypes. At the end of the step, all processors compute a local prototypes version, then forward this version to a unique processor responsible of gathering all local prototypes versions, computing the prototypes shared version (reduce step) and sending it back to all the units. Forwarding the local prototypes versions can be done in several ways, depending on the hardware/software framework: broadcasting can be done using MPI (Message Passing Interface) implementation or web services for example.

Batch K-Means has already been successfully parallelized on multiprocessors machines using MPI (see for example [5], [19] or [24]). This algorithm has also already been implemented on shared nothing platforms, for example in Hadoop, but to our knowledge no theoretical study of the behavior of this algorithm on such a platform has been done yet. This chapter therefore investigate parallelization techniques of Batch K-Means over a platform of cloud computing (in the present case Azure). The main difficulty of the cloud algorithm consists in implementing synchronization and communication between the processing units, using the facilities provided by Windows Azure cloud operating system. We detail this technical challenge, provide theoretical analysis of speed-up that can be achieved on such a platform and the corresponding experimental results.

We now briefly outline the chapter. In section 2, we present the sequential K-Means algorithm and its computational cost. In section 3, we describe how the Batch K-Means is often distributed on several processors and examine the new corresponding computational cost. We also build some modeling of the real cost of a distributed Batch K-Means on Distributed Memory Multiprocessors and develop a bandwidth condition inequality. Section 4 is devoted to present some specificities of the cloud that prevent previous developments of Distributed Memory Multiprocessors K-Means cost to be applied on the cloud. We provide a new parallelization implementation to adapt the distributed Batch K-Means to the cloud specificities. Section 5 presents experimental results of a distributed K-Means on the cloud.

2 Sequential K-Means

2.1 Batch K-Means algorithm

Let us consider the following problem: given a data set of N points $\{\mathbf{z}_i\}_{i=1}^N$ of a d dimensional space, we want to construct K points $\{w_k\}_{k=1}^K$, referred in the following as prototypes or centroïds, as a resume of the data set using the euclidean distance as a similarity measure. Those K prototypes are a minimizer of the following loss function (recalled as the empirical distortion function):

$$\{w_k\}_{k=1}^K = \underset{\{c_k\}_{k=1}^K \in \left(\mathbb{R}^d\right)^\kappa}{\operatorname{argmin}} \sum_{i=1}^{N} \min_{k=1..K} (||\mathbf{z}_i - c_k||_2^2) \tag{1}$$

The existence of an optimal solution is always defined because we can reduce the optimization set to a compact set. As already stated, the computation of an exact minimizer is often untractable, and this problem is proved to be NP-Hard, even in the easier case of d=2 (see [18]). Among the many algorithms that compute an approximation of the optimal minimizer, the Batch K-Means is a well known algorithm, widely used because of its ease to implement.

The algorithm alternates 2 phases iteratively. The first phase, called the assignment phase, takes as input a given set of prototypes, and assigns each point in the data-set to its nearest prototype. The second phase, referred as the recalculation phase, is run after the assignment phase is completed. During this second phase, each prototype is recomputed as the average of all points in the data set that has been assigned to him. When the second phase is completed, the phase 1 is run again, and phase 1 and 2 are run iteratively until a stopping criterion is met. The following logical code (Algorithm 1) describes the alternation of the two phases.

Batch K-Means is an algorithm that produces by construction better prototypes (in regard of the objective function (1)) for each iteration and that stabilizes on a local minimum of this objective function. In many cases, the prototypes and the corresponding empirical loss are deeply modified in the first iterations of the Batch K-Means then they move much more slowly in the latter iterations, and after several dozens of iterations the prototypes and the corresponding empirical loss are totally fixed. Yet, such a behavior is not systematic and Batch-KMeans may need more iterations before stabilizing (see e.g. [2]). The classical stopping criterion are: wait until the algorithm is completed (prototypes remain unmodified between two consecutive iterations), or run an a-priori fixed number of iterations or run the algorithm until the empirical loss gain between two iterations is below a given threshold.

Algorithm 1 Sequential Batch K-Means

Select K initial prototypes $(w_k)_{k=1..K}$
repeat
  for $t = 1$ to N **do**
    for $k = 1$ to K **do**
      compute $||\mathbf{z}_t - w_k||_2^2$
    end for
    find the closest centroid $w_{k^*(t)}$ from $\mathbf{z}_t$;
  end for
  for $k = 1$ to K **do**
    $w_k = \frac{1}{\#\{t,k^*(t)=k\}} \sum_{\{t,k^*(t)=k\}} \mathbf{z}_t$
  end for
until the stopping criterion is met

Batch K-Means can be rewritten as a gradient-descent algorithm (see [3]). As in many other gradient-descent algorithms, Batch K-Means is very sensitive to the initialization. More specifically, both the time to convergence and the quality of the clustering are strongly impacted by the prototypes initialization. We refer the reader to [20] for an in-depth review of different K-Means initialization techniques. In the following, the K prototypes will be initialized by affecting them the first K points of our data set and in the case of a distributed algorithm, all the computing units will be initialized with the same prototypes.

2.2 Complexity cost

The Batch K-Means cost per iteration does not depend on the actual data values but only on the data size. It is therefore possible to provide a precise cost for a Batch K-Means for a given data size. This cost has already been studied by Dhillon and Modha ([5]). Let us briefly list the different operations required to complete the algorithm. For a given number of iterations I, Batch K-Means requires: $I(K+N)d+IKd$ read operations, $IKNd$ subtractions, $IKNd$ square operations, $IKN(d-1)+I(N-K)d$ additions, IKd divisions, $2IN+IKd$ write operations, IKd double comparisons and an enumeration of K sets whose cumulated size is N.

If the data set is small enough to fit into the RAM memory and specific care is made to avoid most of cache miss (see for example [8]), one can neglect the read an write operation costs for the sequential Batch K-Means version. Making the very rough approximations that additions, subtractions and square operations are

all made in a single CPU clock cycle and that $N >> K$ and $N >> d$, one can model the time to run I iterations of Batch K-Means by:

$$SequentialBatch^{Walltime} = (3KNd + KN + Nd)IT^{flop}$$

where T^{flop} denotes the inverse of CPU cadency (i.e. the time for a floating point operation to be evaluated).

3 Distributed K-Means Algorithm on SMP and DMM architectures

3.1 Distribution scheme

As already observed in [6], the Batch K-Means algorithm is inherently data-parallel: the assignment phase which is the CPU-intensive phase of the sequential algorithm version, consists in the same computation (distance calculations) applied on all the points of the data set. The distance calculations are intrinsically parallel, both over the data points and the prototypes. It is therefore natural to split the computational load by allocating disjoint subsets of points to the different processing units. This property makes the Batch K-Means algorithm suitable for many distributed architectures: the assignment phase is shortened by distributing point assignments tasks over the different processing units.

Let us assume that we own M computing units and that we want to run a Batch K-Means over a data set composed of N data points $\{\mathbf{z}_t\}_{t=1..N}$. The initial data set is split into M parts of homogeneous size $S^i = \{\mathbf{z}_t^i\}_{t=1..n}$ for $1 \leq i \leq M$ with $n = N/M$. The computing units are assigned an Id from 1 to M and the computing unit m processes the data set S^m. Each computing unit is loaded to the exact same amount of computation, and the different processors complete their respective tasks in similar amount of time. When all the computing units are done, a single unit can proceed to the recalculation phase, before the reassignment phase is run again. We detail this distributed algorithm version in the following logical code (Algorithm 2).

As previously stated, this distributed algorithm produces after each iteration the exact same result than the sequential Batch K-Means. The distributed Batch K-Means is therefore only evaluated on a speed-up criterion: how much does this distributed algorithm version reduce the total time of execution? In many implementations, the total time of execution is significantly reduced by the parallelization of the assignments and slightly increased by the communication between the different processors that is induced by the parallelization. We can therefore model the total

Algorithm 2 Distributed Batch K-Means on SMP architectures

Code run by the computing unit m
Get same initial prototypes $(w_k)_{k=1..K}$ than other units
repeat
  for $t = 1$ to n **do**
    for $k = 1$ to K **do**
      compute $||z_t^m - w_k||_2^2$
    end for
    find the closest prototype $w_{k^*(t,m)}$ to $\mathbf{z}_t^m$
  end for
  for $k = 1$ to K **do**
    Set $p_k^m = \#\{t,\ \mathbf{z}_t^m \in S^m \ \&\ k^*(t,m) = k\}$
  end for
  for $k = 1$ to K **do**
    $$w_k^m = \frac{1}{p_k^m} \sum_{\{t,\ \mathbf{z}_t^m \in S^m \ \&\ k^*(t,m)=k\}} \mathbf{z}_t^m$$
  end for
  Wait for other processors to finish the for loops
  if i==0 **then**
    for $k = 1$ to K **do**
      $$\text{Set } w_k = \frac{1}{\sum_{m=1}^{M} p_k^m} \sum_{m=1}^{M} p_k^m w_k^m$$
    end for
    Write w into the shared memory so it is available for the other processing units
  end if
until the stopping criterion is met

time of execution by the equation 2.

$$Distributed\ Batch^{WallTime} = T_M^{comp} + T_M^{comm} \tag{2}$$

where T_M^{comp} refers to the wall time of the assignment phase and T_M^{comm} refers to the wall time of recalculation phase (mostly spent in communications).

The wall time of the assignment phase, T_M^{comp} is independent of the computing platform. This wall time roughly equals the assignment phase time of the sequential algorithm divided by the number of processing units M. Indeed, for this phase the translation of the algorithm toward the distributed version does not introduce nor remove any computation cost. It only distributes this computation load on M different processing units. Therefore, this wall time is modeled by the following equation 3.

$$T_M^{comp} = \frac{(3KNd + KN + Nd)IT^{flop}}{M} \tag{3}$$

As previously stated, the wall time of the reassignment phase is mostly spent in communications. During this phase, the different M prototypes versions computed by the M processing units are merged together to produce an aggregated prototype version (which is exactly the prototype version that would have been produced by the sequential algorithm), and then this prototype version (referred by shared version in the following) is made available to each computing unit before the reassignment phase is restarted. In the following sections, we investigate how the shared prototype version is made available depending of the hardware architecture and how this communication process impacts the wall time of the recalculation phase on Symmetric Multi-Processors (SMP) and Distributed Memory Multi-processors (DMM) architectures.

3.2 Communication costs in SMP architectures

SMP refers to a multiprocessor computer architecture where several identical processors are connected to a shared memory and controlled by a single OS instance. A typical SMP configuration is a multi-core processor, where each core is treated as a separate processor. In SMP architectures, the different processing units, the main memory and the hard-disks are connected through dedicated hardware such as buses, switches, etc. Provided that the quantity of communication does not exceed the bandwidth of the interconnection device, the communication are very

fast.

In such an architecture, the communication costs of the recalculation phase are very small compared to the processing costs of the reassignment phase. This brings us to neglect the communication costs in the case of a SMP architecture ($T_M^{comm,SMP} = 0$). One can then simplify the Batch K-Means cost on SMP architecture by the simplifying equation 4.

$$Distributed\ Batch_{SMP}^{WallTime} = T_M^{comp} = \frac{(3KNd + KN + Nd)IT^{flop}}{M} \quad (4)$$

This modeling of Batch K-Means on SMP architectures provides a perfect theoretical model where the neglect of SMP communication costs leads to a theoretical perfect speed-up of a factor M for M processing units.

3.3 Communication costs in DMM architectures

In contrast to SMP architectures, a DMM system refers to a multi-processor in which each processor has a private memory and no shared memory is available. In such a system, the prototypes version computed by the processing unit i is accessible by the processing unit j (with $i \neq j$), if and only if the processing unit i explicitly sends its result to the processing unit j. Since all the communications in DMM architectures need to be explicit, several frameworks have been devised to provide communication and synchronization primitives. These frameworks disburden the application developer from specific communication handling by providing a higher level communication layer. Among these frameworks, the most well known are Message Passing Interface (MPI) (see e.g. [21]) or PVM ([22]).

The wall time of the synchronous distributed Batch K-Means algorithm on DMM has been studied by Dhillon and Modha (see [5]) then by Joshi (see [19]). In both cases, Batch K-Means is distributed on DMM architecture using a MPI framework. In [5], this wall time is modeled by the following equation 3.3.

$$\begin{aligned} Distributed\ Batch_{DMM}^{WallTime} &= T_M^{comp} + T_M^{comm,DMM} \\ &= \frac{(3KNd+KN+Nd)IT^{flop}}{M} + O(dKIT_M^{reduce}) \end{aligned}$$

where T_M^{reduce}, following the notation of [5], denotes the time required to perform a sum or an average operation of M doubles distributed on M processing units.

To determine the recalculation phase wall time on DMM architectures using MPI, one therefore need to determine the time to perform a sum or an average operation

distributed on M processing units, referred as T_M^{reduce}. This quantity is determined by the design of MPI: the MPI framework performs communication and broadcasts data between the processing units using a tree-like topology that is described in the following section (3.5). For such a tree-like topology, the communication and broadcasting primitives are reported to be performed in $O(log(M))$. By consequence, T_M^{reduce} is also reported to be performed in $O(log(M))$ (see e.g. [11]).

In many cases, the hardware architecture provides enough bandwidth for MPI to be very efficient. In the experiments made in [5], the communication costs are very acceptable and have little impact on the distributed Batch K-Means speed-up (see section 3.7). Yet, the communication latency and the bandwidth between processing units should not be neglected in many other cases. The following sections provide a deeper evaluation of T_M^{reduce} and computes some bandwidth condition to prevent communications to become a bottleneck of the algorithm.

3.4 Modeling of communication real costs

A lot of work has been made to provide precise modeling of the MPI primitives performances during the last two decades. In [11], numerical values of many MPI execution performances are provided. As outlined in [9], achieving such numerical measurements is a very difficult task that highly depends on the underlying hardware topology, on the synchronization methods, on the network congestion, on the different MPI implementations, etc. In this section we do not aim to provide satisfactory numerical values of the MPI behavior in our clustering context but rather to explicit qualitative patterns of the constraints brought by communications.

Let us consider the communication scheme described in figure 1. This overly-simplified communication scheme highlights the tree-like nature of the communication patterns adopted by many MPI implementations. More specifically, the figure highlights the logarithmic nature of the communication mechanism: to sum or average values across computing units, many MPI implementations have developed a structure where data chunks are send in multiple steps. Each step of this communication pattern consists in two actions: a processing unit sends its data to a seconde one, the receiving unit merges the two data chunks into a data chunk of the same size. After each step, the number of data chunks that are send in parallel is divided by 2. Such a pattern induces $\lceil log_2(M) \rceil$ communication steps.

In the following equations, we neglect communication latencies as well as the the time to merge the data chunk (which is small in comparison of data communication between the two processing units) and model communication costs as the ratio of

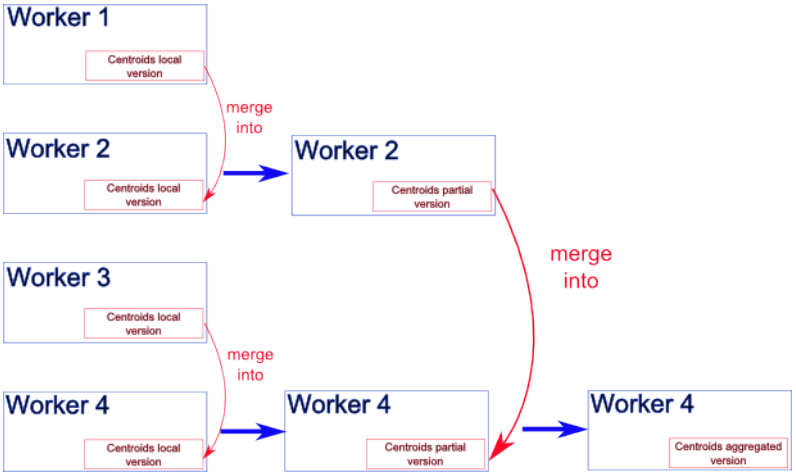

Figure 1: This non-realistic scheme of the merging prototypes logic highlights the tree structure of many MPI primitives. In the first step, the first machine sends a data chunk to the second machine, while the third machine sends simultaneously its data chunk to the fourth machine. A second step follows the first one in which the merging result of the first and second machines are send to the fourth machine that already owns the merging result of the third and fourth machines. In two steps, the data chunks of the four workers are merged.

the quantity of data to be send by the bandwidth. We therefore provide a theoretic modeling of $T_M^{comm,DMM}$ by:

$$T_M^{comm,DMM} = \sum_{m=1}^{\lceil log_2(M) \rceil} \frac{IKdS}{B_{\frac{M}{2^m}}^{DMM,MPI}}$$

where S refers to the size of a double in memory (8 bytes in the following) and $B_x^{DMM,MPI}$ refers to the communication bandwidth per machine while x processing units are communicating at the same time. The number of processing units that are communicating at the same time x has a strong impact on $B_x^{DMM,MPI}$ because of a phenomenon referred as aggregated bandwidth boundaries.

The aggregated bandwidth refers to the maximum number of bytes the actual network can transfer per time unit. We refer the reader to the section 5.2 for a more precise explanation. In the context of our model, we make the rough simplification that the bandwidth per worker only depends on the hardware context and on the total number of processing units, and not on the actual number of processing units

communicating at the same time. We therefore note this bandwidth $B_M^{DMM,MPI}$.

The time spent in communication is then written:

$$\begin{aligned} T_M^{comm} &= \sum_{m=1}^{\lceil log_2(M) \rceil} (\frac{IKdS}{B_M^{DMM,MPI}}) \\ &= \lceil log_2(M) \rceil \frac{IKdS}{B_M^{DMM,MPI}} \end{aligned}$$

We can then deduce an estimation of the speed-up rate on DMM architectures more detailed than in [5], where T_P^{comm} is specified :

$$Speedup = \frac{KMeans\ Sequential\ Cost}{KMeansDMMDistributedCost} \tag{5}$$

$$= \frac{T_1^{comp}}{T_P^{comp} + T_P^{comm}} \tag{6}$$

$$= \frac{3nKdIT^{flop}}{\frac{3nKdIT^{flop}}{M} + \frac{IKdS}{B_M^{DMM,MPI}} \lceil log_2(M) \rceil} \tag{7}$$

$$= \frac{3nT^{flop}}{\frac{3nT^{flop}}{M} + \frac{S}{B_M^{DMM,MPI}} \lceil log_2(M) \rceil} \tag{8}$$

3.5 Comments

The speed-up modeling provided in the previous section is too rough to provide a satisfactory forecasting tool to anticipate the actual speed-up provided by a distributed Batch K-Means implementation on a given architecture. Yet, this naive modeling allows us to draw some qualitative conclusions shared by many distributed Batch K-Means implementations on shared-nothing architectures:

- The speed-up does depend neither on the number of prototypes (K), nor on the data dimension (d) or on the law from which the data are drawn. It only depends on the number of points in the data set (N) and on architecture characteristics such as bandwidth, CPU frequency or whether the OS is a 32-bit or a 64-bit OS.

- The speed-up is a monotonically increasing function of N. In particular, the previous equation leads to $\lim_{N \to +\infty} SpeedUp = M$. From this theoretical

result, one can draw two conclusions: the more the RAM memory of workers is loaded, the more the workers are efficient. Provided the processing units are given enough RAM, it is possible to get an efficiency per worker arbitrary close to 100%.

- The actual speed-up of the distributed implementation might even be higher than the theoretical speed-up modeling since we do not have take into account that the RAM is bounded. When the data is split across the RAM of the multiple processing units in such a way that it fits into the multiple RAM while it would not have fit into the RAM of a single machine, then the speed-up achieved by the distributed version on M processing units can be drastically higher than M.

- The previous equation shows that speed-up are a monotonically decreasing function of T^{flop} and a monotonically increasing function of $B_M^{DMM,MPI}$. This result shows that the parallelization is best suited on slow machines with high communication bandwidth.

3.6 Bandwidth Condition

In this section we briefly determine some theoretical conditions that guarantee that the recalculation phase will be small in comparison of the reassignment phase. This condition translates into:

$$T_M^{comm} << T_M^{comp}$$

Such a condition translates into:

$$\frac{IKdS}{B_M^{DMM,MPI}}\lceil log_2(M)\rceil << \frac{(3NKd + NK + Nd)IT^{flop}}{M}$$

A sufficient condition for equation 3.6 to be verified is that the following equation 9 is verified:

$$\frac{IKdS}{B_M^{DMM,MPI}}\lceil log_2(M)\rceil << \frac{3nKdIT^{flop}}{M} \tag{9}$$

Finally we get the following condition:

$$\frac{N}{M\lceil log_2(M)\rceil} >> \frac{S}{3T^{flop}B_M^{DMM,MPI}}$$

For example, in the context of a DMM architecture with 50 computing units composed of a single mono-core retail processor (2Ghz) and a network with $100MB/sec$ bandwidth per processing unit, the condition turns into:

$$N >> 16,000$$

3.7 Dhillon and Modha case study

The experiments of [5] have been run on a IBM SP2 platform with a maximum of 16 nodes. Each node is a IBM POWER2 processor running at 160MHz with 256 MBytes of main memory. Processors communicate through the High-Performance Switch (HPS) with HPS-2 adapters. Performance of the HPS has been discussed in [23] and [7]. In [7] for example, point-to-point bandwidth in a SP2 with HPS and HPS-2 adapters using MPI is reported to be about 35 MBytes/sec.

It is difficult to estimate bandwidth actually obtained during the experiments of [5], as it is also difficult to estimate effective Flops. In [5], no comments are made about bandwidth, yet Flops are reported to be very fluctuent: 1.2 GigaFlops in some experiment, 1.8 GFlops in another (maximum would be 16 * 160MFlops = 2,5GFlops). Yet, on smaller data sets Flops might be very much lower than the reported numbers. Since we cannot guess the actual bandwidth and Flops during these experiments, theoretical bandwidth and Flops are used in the following to provide approximative results.

Dhillon reports that he has obtained a speed-up factor of 6.22 on 16 processors when using 2^{11} points. Using condition (5), we can estimate speed up when using 2^{11} points :

$$\begin{aligned} EstimatedSpeedup &= \frac{3NT^{flop}}{\frac{3NT^{flop}}{P} + \left(2\frac{SoD}{B_P^{DMM,MPI}} + 5T^{flop}\right)\lceil log_2(P)\rceil} \\ &\simeq 11 \end{aligned}$$

We see the estimated speedup is not accurate, but we are in the range where communication is important enough to prevent a perfect speedup, but yet small enough so a significant speedup is observed.

When using 2^{21} points, we get :

$$EstimatedSpeedup = \frac{3nT^{flop}}{\frac{3nT^{flop}}{P} + \left(2\frac{SoD}{B_P^{DMM,MPI}} + 5T^{flop}\right)\lceil log_2(P)\rceil}$$
$$\simeq 15.996$$

[5] reports that for 2^{21} points, a 15.62 speed-up is observed. Again, anticipated speed-up indicates there will be little issue to parallelize, and observed speed-up is indeed excellent.

4 Implementing Distributed Batch K-Means on Azure

4.1 Recall of some Azure specificities

Windows Azure Platform is Microsoft's cloud computing solution, in the form of Platform as a Service (PaaS). The underlying cloud operating system (Windows Azure) provides services hosting and scalability. It is composed of a storage system (that includes Blob Storage and Queue Storage) and of a processing system (that includes *web roles* and *worker roles*). We refer the reader to the chapter **??** for an in-depth overview of Azure.

The architecture of an Azure hosted application is based on two components: **web roles** and **worker roles**. Web roles are designed for web application programming and do not concern our present work. Worker roles are designed to run general background processing. Each worker role typically gathers several *cloud services* and uses many *workers* (Azure's processing units) to execute them.

Our cloud distributed Batch K-Means prototype uses only one worker role, several services and tens of workers. The computing power is provided by the workers, while Azure storage system is used to implement synchronization and communication between workers. It must be noted indeed that Azure does not offer currently any standard API for distributed computation, neither a low level one such as MPI, nor a more high level one such as Map Reduce ([4]) or Dryad ([12]). Map reduce could be implemented using Azure components (following the strategy of [16]), yet, as pointed out in e.g. [17], those high level API might be inappropriate for iterative machine-learning algorithms such as the k-means. We rely therefore directly on Azure queues and blob storage.

Our prototype uses *Lokad-Cloud*[1], an open-source framework that adds a small abstraction layer to ease Azure workers startup and life cycle management, and storage access.

4.2 The cloud Batch K-Means algorithm

As already outlined, the distributed Batch K-Means can be reformulated as an iterated MapReduce where the assignment phases are run by the mappers and the recalculation phases are performed by the reducers. The design of our distributed cloud Batch K-Means is therefore vastly inspired of the MapReduce abstraction and follows the considerations introduced in the section **??** of chapter **??**.

Following the MapReduce terminology, we split our algorithm into three cloud services (setup, map and reduce services), each one matching a specific need. A queue is associated to each service: it contains messages specifying the storage location of the data needed for the tasks. The processing units regularly ping the queues to acquire a message. Once it has acquired a message, a worker starts running the corresponding service, and the message becomes invisible till the task is completed or timeouts. Overall, we use $M + \sqrt{M} + 1$ processing units in the services described below.

The **SetUp Service** generates M split data sets of $n = N/M$ points in each and put them into the BlobStorage. It is also generating the original shared prototypes which are also stored in the BlobStorage. Once the processing units in charge of the set-up have completed the data generation, they pushes M messages in the queue corresponding to the "Map Service". Each message contains a taskId (from 1 to M) related to a split data set to be processed and a groupId (from 1 to $\sqrt{M}$), which is described above. The same processing unit also pushes $\sqrt{M}$ messages in the queue corresponding to the "Reduce Service". In the current implementation[2], the SetUp service is executed by M processing units to speed-up the generation process.

Once the set-up is completed, the Map queue is filled with M messages (corresponding to the M map tasks) and the Reduce queue is filled with $\sqrt{M}$ messages (corresponding to the $\sqrt{M}$ reduce tasks). Each processing unit pings the different queues to acquire a Map task or a Reduce task.

[1] `http://code.google.com/p/lokad-cloud/`

[2] available at http://code.google.com/p/clouddalvq/

When executing a **Map Service** task, a processing unit is referred as a mapper. Each mapper first downloads the corresponding partial data set it is in charge of (once for all). Then the mapper loads the initial shared prototypes and starts the distance computations that form the assignment phase. When the assignment phase is completed, the mapper builds a local version of the prototypes according to the assignment phase locally run. This prototypes version is sent to the BlobStorage, and the mapper waits for the Reduce service to produce a shared version of the prototypes. When the shared version of the prototypes is made available, the mapper downloads it from the storage and restart the assignment phase using the new prototypes version thus obtained.

When executing a **Reduce Service** task, a processing unit is referred as a partial reducer. Each partial reducer downloads from the storage multiple prototypes versions that come from different map tasks, and merge them into an average prototypes version. More specifically, each Reduce task consists in merging $\sqrt{M}$ prototypes versions. The Reduce task message holds an Id called groupId that refers to the group of prototypes versions that this task needs to collect. When the $\sqrt{M}$ prototypes versions it is in charge of are retrieved, the partial reducer merges the prototypes versions using weighted averages and pushes the merged result into the storage. When all the $\sqrt{M}$ Reduce tasks have been completed, a last reducer, referred as the final reducer, downloads the $\sqrt{M}$ merged results thus produced and merge all of them into a single prototypes version called shared version. This shared version is pushed into the storage to be made available for the mappers[3]. When this shared version is read by the mappers, a new iteration of the algorithm is started and the assignment phase is re-run by the mappers.

The two-steps design of the Reduce service (partial reducers and the final reducer) is of primary importance. This design ensures no processing unit needs to download more than $\sqrt{M}$ prototypes versions per iteration. In the same manner, $\sqrt{M}$ copies of the shared prototypes version are made available in the storage instead of one, to ensure no blob is requested in parallel by more than $\sqrt{M}$ processing units per iteration.

For each iteration of the algorithm, there are 3 synchronization barriers. Firstly, each partial reducer needs to get the results of the $\sqrt{M}$ mappers it is related with before merging them. Secondly, the final reducer needs to get the results of all the $\sqrt{M}$ partial reducers before merging them into the shared prototypes version. Finally, each mapper needs to wait for the final reducer to push into the BlobStor-

[3]More specifically, the final reducer pushes $\sqrt{M}$ copies of the shared prototypes version in the storage instead of one. Each of this copies is read by all the mappers sharing the same groupId.

age the shared version before restarting the reassignment phase. As outlined in the chapter **??**, the synchronization primitive is one of the few primitives that is not directly provided by Azure. Let's consider two different designs to implement this synchronization process.

A first solution consists in using a counter to keep track of the number of tasks that need to be completed before the synchronization barrier could be removed. This counter needs to be idempotent and designed such as the contention is limited. Such a counter has been already described in the BitTreeCounter section of chapter **??**. For example, all the mappers that process tasks sharing the same groupId could own such a counter initially set to $\sqrt{M}$; when a mapper is done, it decrements the counter, and when a mapper decrements the counter from 1 to 0, it knows all the other mappers of the same groupId are also done, so that the partial reduce could be started.

A second solution comes from the ability for each processing unit to ping a specific blob in the storage to determine if the blob exists or not. Let us re-examine the previous example. Instead of using a specific counter and starting the reducer only when the counter hits 0, the partial reducer could be started from the beginning, in parallel of the mappers. This partial reducer would then regularly query the storage to detect whether the prototypes versions produced by the mappers have already been made available. When all the $\sqrt{M}$ versions have been retrieved, then the partial reducer can start the merging operation. This second solution requires that each blob put into the storage has a corresponding blobName that is pre-defined using a fixed rule; in the case of prototypes versions made by mappers, the corresponding blobNames are built using the following addressing rule: prefix/iteration/groupId/jobId.

Both the solutions presented provide idempotency and avoid contention. The second solution has been chosen because it helps reducing straggler issues (see section 5.4) by overlapping latest computations of mappers with retrieval of first available results by the partial reducer. This second solution comes with the drawback of running more workers ($M + \sqrt{M} + 1$) than the first solution (M). This drawback is not of primary importance since, as this will be highlighted in the following sections, the scale-up of our algorithm will be more limited by the bandwidth/CPU power of our machines than by the actual cost of these machines.

The figure 4.2 provides a scheme of the cloud implementation of our distributed Batch K-Means. The algorithms 3 (resp. 4 and 5) reproduce the logical code run by the mappers (resp. the partial reducers and the final reducer).

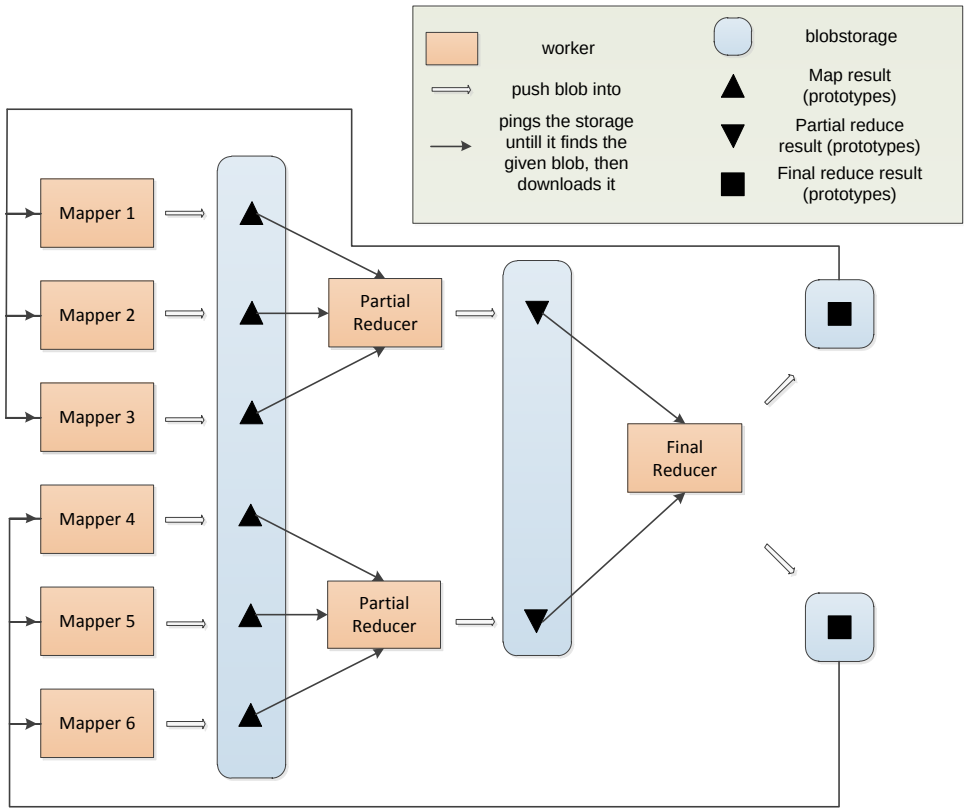

Figure 2: Distribution scheme of our cloud distributed Batch K-Means. The communications between workers are conveyed through the BlobStorage. The recalculation phase is a two-step process run by the partial reducers and the final reducer to reduce I/O contention.

Algorithm 3 Distributed Cloud Batch K-Means : Mapper

Dequeue a message from the Map Queue.
Get taskId, groupId and IterationMax from the message.
Set m=taskId
Retrieve the partial data set $S^m = \{\mathbf{z}_i^m\}_{i=1..N}$ from the storage
Retrieve the initial prototypes shared version $\{w_k^{srd}\}_{k=1..K}$
Initialize w^m as $w^m = w^{srd}$
for It=0; It < IterationMax ; It++ **do**
  for $\mathbf{z}_i^m \in S^m$ **do**
    for $k = 1$ to K **do**
      Compute $||\mathbf{z}_i^m - w_k||_2^2$
    end for
    Find the closest prototype $w_{k^*(i,m)}$ to $\mathbf{z}_i^m$
  end for
  for $k = 1$ to K **do**
    Set $p_k^m = \#\{t,\ \mathbf{z}_t^m \in S^m \ \&\ k^*(t,m) = k\}$
  end for
  for $k = 1$ to K **do**
    Set $w_k^m = \dfrac{1}{p_k^m} \displaystyle\sum_{\{t,\ \mathbf{z}_t^m \in S^m\ \&\ k^*(t,m)=k\}} \mathbf{z}_t^m$
  end for
  Send w^m into the storage in a location depending of the iteration It
  Send p^m into the storage in a location depending of the iteration It
  Ping the storage every second to check if w^{srd} is available for iteration It
  Download it when it becomes available
  Replace w^m by the new downloaded shared version
end for

Algorithm 4 Distributed Cloud Batch K-Means : Partial Reducer

Dequeue a message from the Partial Reduce Queue.
Get groupId and IterationMax from the message.
Set g= groudId
for It=0; It < IterationMax ; It++ **do**
  Retrieve the prototypes version $\{w^m\}_{m=g\sqrt{M}..(g+1)\sqrt{M}}$ corresponding to iteration It
  Retrieve the corresponding weights $\{p^m\}_{m=g\sqrt{M}..(g+1)\sqrt{M}}$ corresponding to iteration It
  for $k = 1$ to K **do**
    Set $p_k^g = \sum_{m=g\sqrt{M}}^{(g+1)\sqrt{M}} p_k^m$
    Set $w_k^g = \frac{1}{p_k^g} \sum_{m=g\sqrt{M}}^{(g+1)\sqrt{M}} p_k^m w_k^m$
  end for
  Send w^g into the storage in a location depending of the iteration It
  Send p^g into the storage in a location depending of the iteration It
end for

Algorithm 5 Distributed Cloud Batch K-Means : Final Reducer

Dequeue the single message from the Final Reduce Queue.
Get IterationMax from the message.
for It=0; It < IterationMax ; It++ **do**
  Retrieve the prototypes version $\{w^g\}_{g=1..\sqrt{M}}$ corresponding to iteration It
  Retrieve the corresponding weights $\{p^g\}_{g=1..\sqrt{M}}$ corresponding to iteration It
  for $k = 1$ to K **do**
    Set $p_k^{srd} = \sum_{g=1}^{\sqrt{M}} p_k^g$
    Set $w_k^{srd} = \frac{1}{p_k^{srd}} \sum_{g=1}^{\sqrt{M}} p_k^g w_k^g$
  end for
  Send w^{srd} into the storage in a location depending of the iteration It
end for

4.3 Comments on the algorithm

In the previous proposed algorithm as well as in the DMM implementation of [6], the assignment phase of Batch K-Means is perfectly parallelized. As already outlined in the modeling of speed-up in the case of DMM architectures, the global performances are hindered by the reduction phase wall time duration. In the case of a SMP architecture, the recalculation phase is approximately instantaneous and the parallelization in such architectures are very efficient. On DMM architectures, we have already showed how the MPI framework provides very efficient primitives to perform the reduction phase in amounts of time that are in most cases negligible (with a $O(log(M))$ cost). In such cases, the DMM implementations lead to a linear speedup nearly as optimal as in the SMP case, as confirmed in the large data set experiments of [6].

In the case of our Azure implementation that relies on the storage instead of direct inter-machines communications, the recalculation phase is proportionally much longer than in the DMM case. The two-steps design of our algorithm produces a recalculation phase cost of $O(\sqrt{M})$ because the partial reducers as well as the final reducer need to download $\sqrt{M}$ prototypes versions per iteration. Such a cost is asymptotically much higher than the asymptotical cost of the MPI framework ($O(log(M))$). While it would have been possible for our reduction architecture to complete in $O(log(M))$ by implementing a $log(M)$-steps architecture, such a design would have led to higher recalculation costs because of frictions. Indeed, the latency between a task to being completed and the consumer of the former task result to acknowledge that the result is available is rather high. This phenomenon highlights a previously stated remark (see chapter **??**): the cloud platforms are probably not suited to process very fine-grained tasks.

We also draw the reader's attention on the importance of the data set location and download. The previous algorithm is inspired from an iterated MapReduce with the noticeable exception that each map task does not correspond to a single iteration of K-Means but to all the iterations. Since Azure does not provide a mechanism to run computations on the physical machines where the data are stored through the BlobStorage, each processing unit need to download the data set from the storage (i.e. from distant machines). To prevent each of the workers from re-loading a data set at each iteration, we have chosen that each processing unit was processing the same data chunk for each iteration.

4.4 Optimizing the number of processing units

A very important aspect of our cloud implementation of Batch K-Means is the elasticity provided by the cloud. On SMP or DMM architectures, the quantity of CPU facilities is bounded by the actual hardware, and the communication are fast enough so the interest of the user is to run the algorithm on all the hardware available. The situation is significantly different for our cloud implementation. Indeed, the cloud capabilities can be redimensionned on-demand to better suit our algorithm requirements. Besides, the communication costs induced show that oversizing the number of workers would lead to increase the overall algorithm wall time because of these higher communication costs. Running Batch K-Means on cloud computing therefore introduce a new interesting question: what is the (optimal) amount of workers that minimize our algorithm wall time?

To answer this question, we re-tailor the speed-up formula detailed in section 3.5. As already explained in section 4.2, the symbol M in the following will not refer to the total number of processing units but to the number of processing units that will run the Map tasks. The total amount of processing units could be lowered to this quantity, but as outlined in the same section, it would come at the cost of a slightly slower algorithm. As in the DMM case, the distance calculations wall time stays unmodified:

$$T_M^{comp} = \frac{(3NKd + NK + Nd)IT^{flop}}{M}$$

where N, as in the DMM architecture case, stands for the total number of points in all the data sets gathered. In addition to the distance calculations, a second cost is introduced to model the reassignment phase wall time: it is the time to load the data set from the storage. Let us introduce T_{Blob}^{read} (resp. T_{Blob}^{write}) that refers to the time needed by a given processing unit to download (resp. upload) a blob from (resp. to) the storage per memory unit. The cost to load the data set from the storage is then modeled by the following equation:

$$T_M^{Load} = \frac{NdST_{Blob}^{read}}{M}$$

We recall that by design this loading operation need to be performed only once, even when $I > 1$. This loading operation can neglected in speed up model if $T_M^{Load} << T_M^{Comp}$. A sufficient condition for this to be true is:

$$\frac{ST_{Blob}^{read}}{3IKT^{flop}} << 1$$

This condition is true in almost all the case. We now provide a new wall time modeling of the recalculation phase performed by the two-steps reduce architecture provided in section 4.2. This recalculation phase is composed of multiple communications between workers and storage as well as average computation of the different prototypes versions. More specifically, each iteration requires:

- M mappers to write their version "simultaneously" in the storage
- Each of the partial reducers to retrieve $\sqrt{M}$ prototypes versions
- Each of the partial reducers to compute an average of the versions thus retrieved
- Each of the partial reducers to write "simultaneously" its result in the storage
- The final reducer to retrieve the $\sqrt{M}$ partial reducer versions
- The final reducer to compute the shared version accordingly
- The final reducer to write the $\sqrt{M}$ shared versions in the storage
- All the mappers to retrieve the shared version

The recalculation phase per iteration can therefore be modeled by the following equation:

$$\begin{aligned} T_P^{comm,periteration} &= KdST_{Blob}^{write} + \sqrt{M}KdST_{Blob}^{read} + 5(\sqrt{M})KdT^{flop} \\ &+ KdsT_{Blob}^{read} + KdST_{Blob}^{write} + \sqrt{M}KdST_{Blob}^{read} \\ &+ 5(\sqrt{M})KdT^{flop} + \sqrt{M}KdST_{Blob}^{write} \end{aligned}$$

Keeping only the most significant terms, we get:

$$T_M^{comm} \simeq I\sqrt{M}KdS(2T_{Blob}^{read} + T_{Blob}^{write})$$

Then we can deduce an "idea" of the speed-up factor :

$$SpeedUp = \frac{T_1^{comp}}{T_M^{comp} + T_M^{comm}} \tag{10}$$

$$\simeq \frac{3IKNdT^{flop}}{\frac{3IKNdT^{flop}}{M} + I\sqrt{M}KdS(2T_{Blob}^{read} + T_{Blob}^{write})} \tag{11}$$

$$\simeq \frac{3NT^{flop}}{\frac{3NT^{flop}}{M} + \sqrt{M}S(2T_{Blob}^{read} + T_{Blob}^{write})} \tag{12}$$

Using this modeling, which is very naive, one can see there is an optimal number of workers to use. This number is obtained by having time spent by communication equals time spent by computation.

$$M^* = \sqrt[2/3]{\frac{3NT^{flop}}{S(2T_{Blob}^{read} + T_{Blob}^{write})}} \tag{13}$$

In this model, the best number of processing units does not scale linearly with the number of data points n (whereas it was the case in the DMM implementation cost model). It directly follows that our model provide a theoretical impossibility to provide an infinite scale-up with the two-steps reduce architecture. The section 5.6 investigates how our cloud distributed Batch K-Means actually perform in terms of practical scale-up.

One can verify that for the previous value of M^*, $T_{M^*}^{comp} = T_{M^*}^{comm}$. This means that when running the optimal number of processing units, the reassignment phase and the recalculation phase have the same duration. In such a situation the efficiency of our implementation is rather low: with $M^* + \sqrt{M^*} + 1$ processing units used in the algorithm, we only get a speed-up of $M^*/2$.

5 Experimental results

5.1 Azure base performances

In order to use our cost models as a predictive tool to determine the optimal number of workers M^* and our implementation performance, one needs to evaluate the Azure services performances. These performances have already been briefly analyzed in section **??** of chapter **??**. We refer the reader to this section for more explanations and just recall here the recorded performances.

- **BlobStorage Read Bandwidth:** 8MBytes/sec
- **BlobStorage Write Bandwidth:** 3MBytes/sec
- **CPU performance while performing distance calculations:** 670 MFlops

5.2 The two-steps reduce architecture benchmark

The aggregated bandwidth refers to the maximum number of bytes the actual network can transfer per time unit. In most cases, such a quantity does not equal the product of maximal bandwidth per processing unit by the number of processing units, since the network may not sustain such maximal bandwidths for each workers when all of them are communicating at the same time. The aggregated bandwidth measurement is a difficult task that is hardware and task dependant. In this section we focus on a specific case raised by our clustering algorithm developed in chapter **??**.

In the case of the two-steps recalculation phase implementation developed in section 4.2, theoretically determining how long would the recalculation phase be is too difficult and inaccurate, partly because of the previously mentioned aggregated bandwidth boundaries. We therefore produce a custom benchmark to evaluate the time spent in the recalculation phase as follows: the prototypes are designed as a data chunk of 8MBytes and the recalculation phase is run 10 times($I = 10$). For different values of M, the clustering implementation is run but the processing part is replaced by waiting a fix period of time (15 seconds), so we could record communication time without being affected by straggler issues, as reported in section 5.4. The following table provides wall time of this benchmark (for 10 iterations), and the amount of time spent in the recalculation phase (Wall Time - 10*15 seconds).

M	5	10	20	30	40	50	60	70	80
Wall Time (in sec)	287	300	335	359	392	421	434	468	479
Communication (in sec)	137	150	185	209	242	271	284	318	329
$2T_{Blob}^{read} + T_{Blob}^{write}, (in\ 10^{-7} sec/Byte)$	7.64	5.92	5.16	4.76	4.78	4.78	4.59	4.73	4.58

M	90	100	110	120	130
Wall Time (in sec)	509	533	697	591	620
Communication (in sec)	359	383	547	441	470
$2T_{Blob}^{read} + T_{Blob}^{write}, (in\ 10^{-7} sec/Byte)$	4.71	4.77	6.51	5.02	5.13

First of all, one can notice that the quantity $2T_{Blob}^{read} + T_{Blob}^{write}$ does not grow with M (at least for $M < 100$). This result proves that we do not suffer in our experiment from aggregated bandwidth boundaries before using 100 workers. Secondly, we note that the obtained value is smaller than the value that would be obtained following the values provided in section 5.1: indeed, the parallelization of downloads

and uploads in each machine (through multiple threads) reduces enough the communication to unbalance the frictions introduced by our two-step design. Finally, the whole process is sometimes behaving much worse than expected (see the case $M = 110$). The case $M = 110$ has been re-launched 1 hour later, obtaining the value 4.85.

5.3 Experiment settings

In the following experiments, Batch K-Means are run on synthetic data. As explained in the introduction of this chapter, the Batch K-Means wall time depends on the data size but not on the actual data values, except for the number of iterations to convergence. Thus, the synthetic nature of the data has no impact on the conclusion that we draw. Besides, the synthetic nature of our data has allowed us to easily modify parameters such as d to highlight some results. The synthetic data are generated uniformly in the unit hypercube using the following settings: the dimension d is set to 1000 and the number of clusters K is set to 1000. The number of points in the total data set depends on the experiment run: for scale-up experiments, the total data set is composed of $500,000$ data points that are evenly split among the multiple processing units, while the data set total amount of points grows with the number of processing units in the scale-up experiments. In all our experiments, the algorithm is run for 10 iterations to get stable timing estimates.

Theoretically, K and d should have the same impact on map and reduce phases since map and reduce costs are supposed to be proportional of KD. This result traduces the fact that speed-up is theoretically agnostic to K or d. Yet, since our model does not take into account latency, having very small values of K and D would lead to underestimate communication costs by neglecting latency. Provided that K and d are kept big enough, our speed-up/scale-up results remain strongly close.

As explained in the introduction of the chapter, the distributed Batch K-Means produce for each iteration the exact same results than a sequential Batch K-Means would run. Thus, the following experiments only focus on the speed-up provided by the parallelization and not the function loss improvements.

5.4 Straggler issues

In this section we investigate the variability of CPU performances of the multiple processing units and show that some tasks are processed in amounts of time signifi-

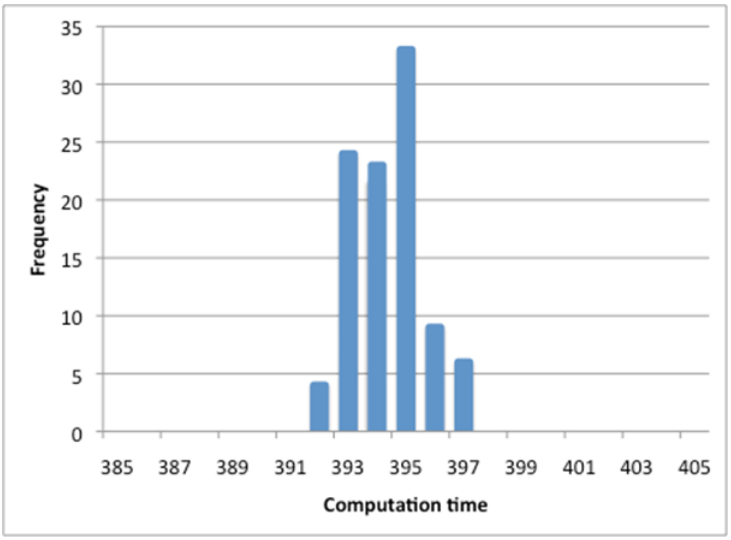

Figure 3: Distribution of the processing time (in second) for multiple runs of the same computation task for a single VM. As expected, the distribution is tightened on a specific value.

cantly higher than expected, a phenomenon referred as stragglers (see e.g. [4] or [15]).

In the first experiment, we run 100 times the same task that consists in a heavy distance calculations load (each task corresponds to a reassignment phase). The task is expected to be run in 7 minutes. The results are provided in figure 5.4.
In the second experiment, the same task is run 10 times in a row by 85 mappers. Each of the 850 records therefore consists in the same computation load, performed on processing units supposedly of the same performance (Azure small role instances). The same experiment has been run on 2 different days, on 2 different hours, on different workers, and the same following patterns have been observed. The following figure provides the empirical distribution of these computation durations.

From this experiment, we deduce that:

- *The 3 modes distribution* : 90% of the tasks are completed between 390 and 500 seconds. For the tasks completed in this interval, we observe 3 different modes of our empirical distribution.The 3 modes may be due to hardware heterogeneity or multiple VM hosted on the same physical machine.

- *A worker can be affected by temporary slowness* : The three longest runs

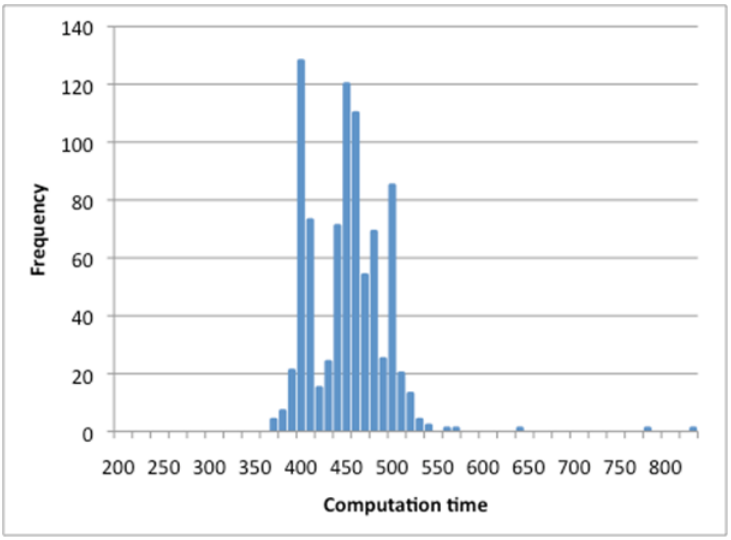

Figure 4: Distribution of the processing time (in second) for multiple runs of the same computation task for multiple VM. One can note the "3 modes" distribution and outliers (tasks run in much more time).

(823 seconds, 632 seconds, 778 seconds) have been performed by the same VM, which has also performed very well on other iterations: (360 seconds, 364 seconds, ...). This could be explained by very different reasons, such as the fact the physical machine hosting our VM has been hosting temporarily another VM, a temporary downsizing of the size of cache memory for our VM, etc.

Straggler issues have already been pointed out, for example in the original MapReduce article [4] by Google, yet they were observed while running thousands of machines. We show that straggler issues are also observed on as such a small pool of workers as 100 VM. [4] describes a monitoring framework to detect tasks taking too much time, and use backup workers to re-launch tasks that has been detected to be too long. Yet, this approach leads to wait for the standard duration of the task before detecting straggler tasks and launching again the corresponding tasks. This situation severely limits the potential speed-up that can be achieved.

As pointed out by Graphlab [17], the ease of access of MapReduce frameworks and the great help it provides to design scalable applications has driven part of the machine-learning community to think their algorithms to fit in a MapReduce framework. However, the combination of stragglers and of a synchronous framework like MapReduce prevent in many cases users from obtaining overall good

speed-ups.

5.5 Speed-up

In this section we report the performances of the cloud implementation proposed in section 4.2 in term of speed-up. In other terms, we investigate whether the proposed implementation allows us to reduce the Batch K-Means execution wall time for a given data set. In addition, we compare the observed speed-up to the theoretical speed-up obtained by the equation 10. We report the results of one out of multiple experiments that we run. The other experiments showed the same general patterns and qualitative conclusions.

The proposed implementation is tested using the settings described in section 5.3. The algorithm is run for 10 iterations to get stable timing estimates. Neglecting loading time and memory issues (a small instance has only 1 GB of memory), a sequential Batch K-Means implementation would use approximately 6 hours and 13 minutes to run the 10 iterations.

The following table reports the total running time in seconds (including data loading) of the proposed implementation for different numbers of mappers (M). We report the speedup over the theoretical total running time, the efficiency (speedup divided by the total number of processing units $M + \sqrt{M} + 1$) and the theoretical efficiency predicted by the model.

M	10	50	60	70	80	90	95	100	110	120	130	140	150	160
Time	2223	657	574	551	560	525	521	539	544	544	574	603	605	674
SpeedUp	10.0	34.1	39.0	40.6	40.0	42.6	43.0	41.5	41.2	41.2	39.0	37.1	37.0	33.2
Efficiency	0.67	0.58	0.57	0.51	0.44	0.42	0.41	0.37	0.34	0.31	0.27	0.24	0.23	0.19
Theo. Eff.	0.63	0.61	0.60	0.55	0.53	0.49	0.48	0.47	0.43	0.41	0.37	0.36	0.33	0.32

As expected, the total processing time is minimal for a specific value of M (here 95), for which half of the total time is spent in the map phase and for which the speed-up (43.0) is approximately $M^*/2$ (as predicted by the model). With the values of T^{flop} reported in section 5.1 and the values of $2T^{read}_{Blob} + T^{write}_{Blob}$ evaluated in section 5.2, the estimated optimal number of mappers is 70. While the theoretical optimal number of workers slightly underestimate the actual optimal number of workers, the equation 13 provides a first estimate of this number before running the experiments. More specifically, once the values $2T^{read}_{Blob} + T^{write}_{Blob}$ and T^{flop} have been once evaluated, our equations provide an a-priori tool to estimate what would be the optimal number of machines to use.

5.6 Optimal number of processing units and scale-up

We recall that the scale-up is the ability to cope with more data in the same amount of time, provided that the number of processing units is increased accordingly. In our cloud Batch K-Means, the theoretical optimal number of mappers M^* is not proportional to N (see equation 13), contrary to the DMM/MPI model cost (because of the communication cost in O(log(M))). As a consequence, our cloud version cannot hope to achieve linear scale-up.

As a consequence, the scale-up challenge turns into minimizing growth of wall time as N grows, using $M^*(N)$ mappers. Theoretically, our model gives a processing cost (T_M^{comp}) proportional to N/M and a communication cost (T_M^{comm}) proportional to $\sqrt{M}$. As a consequence, the algorithm execution total time ($T_M^{comp} + T_M comm$) is proportional to $\sqrt[1/3]{N}$ and the optimal number of workers M^* is proportional to $\sqrt[2/3]{N}$.

In the following experiment, the values of K and D are kept constant ($K = 1000$ and $D = 1000$). For various values of N, our implementation is run on different values of M to determine the effective optimal values of M for a given N.

	N	M^*	Wall Time	Sequential theoretic time	Effective Speedup	Theoretical Speedup ($= \frac{M^*}{2}$)
Experiment 1	62500	27	264	2798	10.6	9
Experiment 2	125000	45	306	5597	18.29	15
Experiment 3	250000	78	384	11194	29.15	26
Experiment 4	500000	95	521	22388	43.0	31.6

As expected, one can see that $M^*(N)$ and $T_M^{comp} + T_M comm$ do not grow as fast as N. Between the experiment 1 and the experiment 4, N is multiplied by 8. Our theoretical model anticipates that M^* should grow accordingly by $8^{2/3} = 4$. Indeed, M^* grows from 27 to 95 (that is a 3.51 ratio). In the same way, our model anticipates that the execution wall time should grow by $8^{1/3} = 2$. Indeed, the execution wall time grows from 264 seconds to 521 seconds. The figure 5.6 provides the detailed experiments results of the speed-up obtained for multiple values of N and M.

For our last experiment, we aim to achieve the nominal highest value possible for speed-up. As explained in section 3.5, for a fixed number of mappers M, the best achievable speed-up is obtained by filling the RAM of each machine with data so each machine is in charge of a heavy computation load. While the table **??** and the figure 5.6 show how the speed-up grows with N (using $M^*(N)$), the following figure 5.6 shows how the speed-up grows with M (using the highest value of N that do not oversize the RAM of our VM). For this experiment, we set $K = 1000$,

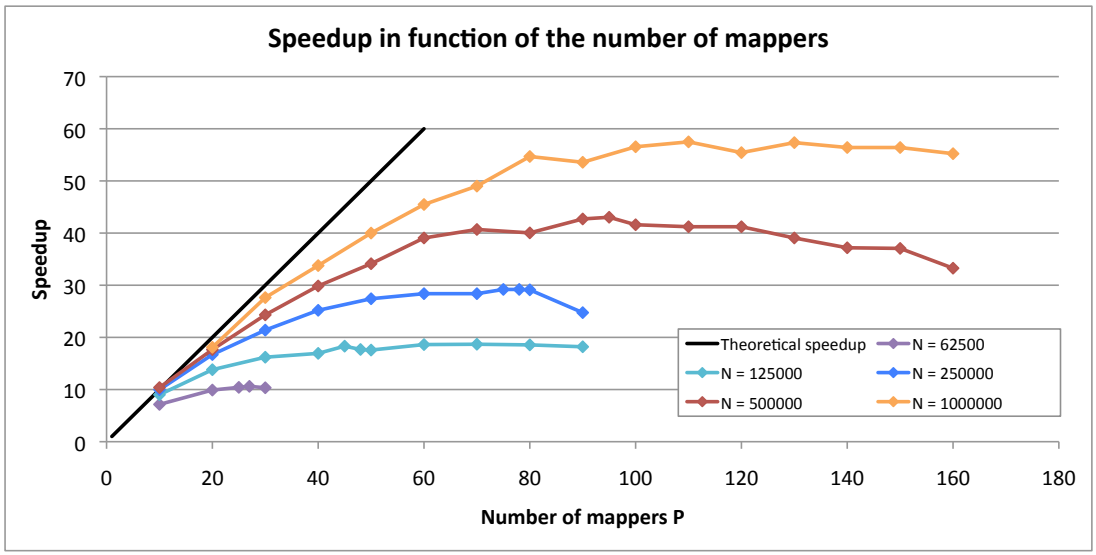

Figure 5: Charts of speed-up performance curves for our cloud Batch K-Means implementation with different data set size. For a given size N, the speed-up grows with the number of processing units until M^*, then the speed-up slowly decrease.

$D = 1000$, and set N in such a way that each mappers is given 50.000 data points $N = M * 50.000$[4].
Overall, the obtained performances are satisfactory and the predictive model provides reasonable estimates of the execution wall time and of the optimal number of processing units that need to be used. While there is room for improving our implementation, the latency issues might prevent resorting on a tree like $O(log(P))$ reducer as available in MPI without using direct inter-machines communications. Without native high performances API, communication aspects will probably remain a major concern in Azure implementations.

5.7 Price

Without any commitment and package, 1 hour of CPU on small VM is charged 0.1 dollar and 1000000 transactions between workers and storage (QueueStorage or BlobStorage) are charged 1 dollar. Therefore, one clustering experiment with 10 workers (decreasing wall time from 6 hours to 37 minutes) is charged less than 2

[4]The value of $n = 50.000$ corresponds to 400 MBytes in RAM, while the RAM of the small role instances are supposed to have 1.75GBytes. In theory, we could have therefore loaded much more our instances. In practice, when we run this experiment in 2010, the VM crashed when we used higher values for n

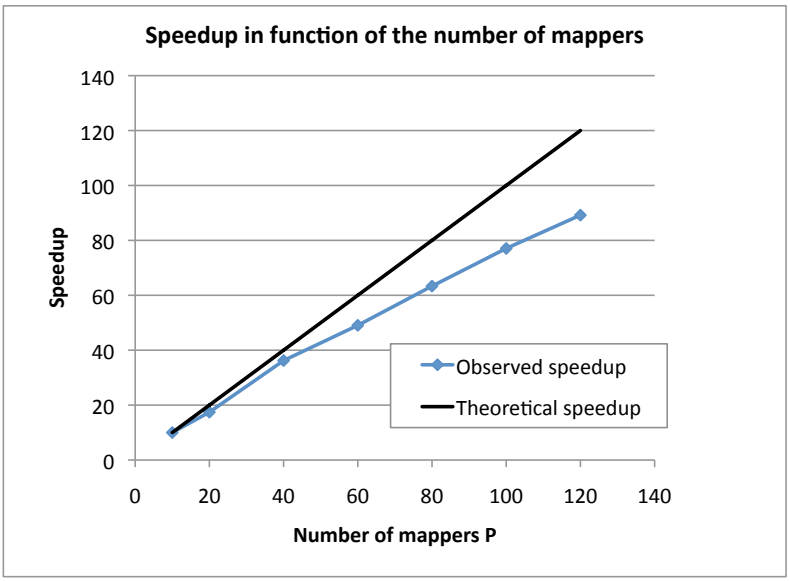

Figure 6: Charts of speed-up performance curves for our cloud Batch K-Means implementation with different number of processing units. For each value of M, the value of N is set accordingly so that the processing units are heavy loaded with data and computations. When the number of processing units grows, the communication costs increase and the spread between the obtained speed-up and the theoretical optimal speed-up increases.

dollars, and our biggest experiment running 175 workers is charged less than 20 dollars. Since our experiments are not one hour long but 10 minutes long at worst, if one can recycle the 50 other minutes running other computations, then the cost of our biggest experiment drops to 4 dollars on average. For bigger uses, some packages are available to decrease charges.

References

[1] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Eng., 8(6):962-969*, 1996.

[2] David Arthur and Sergei Vassilvitskii. How slow is the k-means method?

[3] Léon Bottou and Yoshua Bengio. Convergence properties of the k-means algorithms. In *Advances in Neural Information Processing Systems 7*, pages 585–592. MIT Press, 1995.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[5] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. *Large-scale Parallel KDD Systems Workshop, ACM SIGKDD*, August 1999.

[6] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 245–260, London, UK, 2000. Springer-Verlag.

[7] On Ibm Sp (draft, Gang Cheng, and Marek Podgorny. The high performance switch and programming interfaces on ibm sp2, 1995.

[8] Ulrich Drepper. What every programmer should know about memory, 2007.

[9] William Gropp and Ewing Lusk. Reproducible measurements of mpi performance characteristics. pages 11–18. Springer-Verlag, 1999.

[10] C. Hennig. Models and methods for clusterwise linear regression. In *Proceedings in Computational Statistics*, pages 3–0. Springer, 1999.

[11] Torsten Hoefler, Rajeev Thakur, and Jesper Larsson Träff. Toward performance models of mpi implementations for understanding application scaling issues.

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[13] R. Agrawal J. C. Shafer and M. Mehta. A scalable parallel classifier for data mining. *Proc. 22nd International Conference on VLDB, Mumbai, India*, 1996.

[14] A K Jain, M N Murty, and P. J. Flynn. Data clustering: A review. In *ACM computing surveys, Vol.31, no.3, September*, 1999.

[15] Jimmy Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce.

[16] Huan Liu and Dan Orban. Cloud mapreduce: a mapreduce implementation on top of a cloud operating system. Technical report, Accenture Technology Labs, 2009. `http://code.google.com/p/cloudmapreduce/`.

[17] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[18] Kasturi Varadarajan Meena Mahajan, Prajakta Nimbhorkar. The planar k-means problem is np-hard. *?*, - -.

[19] Manasi N.Joshi. Parallel k-means algorithm on distributed memory multiprocessors. *?*, spring 2003.

[20] A. D. Peterson, A. P. Ghosh, and R. Maitra. A systematic evaluation of different methods for initializing the k-means clustering algorithm. Technical report, 2010.

[21] Marc Snir, Steve Otto, Steven Huss-Lederman, Walker David, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Boston, 1996.

[22] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[23] John Sobolewski Vasilios Georgitsis. Performance of mpl and mpich on the sp2 system.

[24] Liao W. Parallel k-means data clustering. 2005.