K-means on Azure

Matthieu Durut
Lokad
70, rue Lemercier, 75018 Paris – France
matthieu.durut@lokad.com

Fabrice Rossi
BILab, Télécom ParisTech
LTCI - UMR CNRS 5141
46, rue Barrault, 75013 Paris – France
fabrice.rossi@telecom-paristech.fr

Abstract

This paper studies how parallel machine learning algorithms can be implemented on top of Microsoft Windows Azure cloud computing platform. The storage component of the platform is used to provide synchronization and communication between processing units. We report the speedups of a parallel k-means algorithm obtained on up to 200 processing units.

1 Parallel K-means

This paper uses the k-means clustering algorithm as a typical example of machine learning methods. This choice is motivated by several reasons. Firstly, while better clustering methods are available, k-means remains a useful tool, especially in the context of vector quantization. For instance, k-means can be used to summarize a very large dataset via a reduced set of prototypes. As such, k-means is at least a standard pre-processing tool. Secondly, k-means has a low processing cost, proportional to the data size multiply by K, the number of clusters. Thus, it is a good candidate for processing very large datasets, even with a sequential implementation on a single processing unit. Thirdly, apart for the number of iterations to convergence, the processing time of 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 dataset with the same dimensions. (This is also true for the communication requirement in classical parallel versions.) Finally, k-means is easy to parallelize on shared memory computers and on local clusters of workstations: numerous publications (see e.g., [1]) report linear speedup up to at least 16 processing units (PU, which can be CPU cores or workstations).

Let us recall the principles of Dhillon and Modha's parallel k-means [1]. K-means consists in alternating two phases (after proper initialization). In the assignment phase, the algorithm computes the Euclidean distance between each of the N data point $X_i \in \mathbb{R}^D$ and each of the K cluster prototypes $m_j \in \mathbb{R}^D$. Each X_i is assigned to its closest prototype, denoted m_{k_i}. In the recalculation phase, each prototype is recomputed as the average of the data points assigned to it in the previous phase.

The assignment phase costs roughly $3NKD$ floating point operations, while the recalculation of the prototypes costs only approximately $(N + K)D$ operations. In addition, 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 N/P data points to P PU. Each PU, called a mapper following [2] terminology, computes a full assignment phase for its N/P points, including the computation of the sum of the data points assigned to each cluster for a total cost of $3NKD/P+ND/P$. Then the PU synchronize and compute the new value of the prototypes by merging the partial results. In [1] this is done via a dedicated "reducing" function of the MPI library [3] which uses $\log P$ parallel rounds of communications between the PU. Each PU to PU communication has to transmit the partial results of one PU to the other one, inducing a communication cost proportional to KD. The final result, i.e., the new values of the prototypes, is broadcasted to all the PU, which starts a new iteration on reception.

The cloud implementation studied in this paper is based on the parallel k-means described above. The main difficulty consists in implementing synchronization and communication between the PU, using the facilities provided by Windows Azure cloud operating system.

2 Microsoft Windows Azure

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*). The Azure components we used are described below.

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 the present article. 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 prototype uses only one worker role, several services and tens of PU. 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 [3], nor a more high level one such as Map Reduce [2] or Dryad [4]. Map reduce could be implemented using Azure components (following the strategy of [5]), yet, as pointed out in e.g. [6], 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.

Azure Queues provide a message delivery mechanism through distributed queues. Queues are designed to store a large amount of small messages (with maximal individual size of 8 KB). Using queues to communicate helps building loosely coupled components and mitigates the impact of individual component failure. Here, queues are used to store job messages. Messages stored in a queue are guaranteed to be returned at least once, but possibly several times: this requires one to design idempotent jobs. When a message is unqueued by a worker, this message is not deleted but it becomes invisible for other workers. If a worker fails to complete the corresponding job (because it throws some exception or because the worker dies), the message becomes available after a certain period of time. Through this process, one can make sure no job is lost because of e.g., a hardware failure.

Azure Blob Storage enables applications to store large objects, up to 50 GB each. It supports a massively scalable blob system, where hot blobs will be served from many servers to scale read access. Blobs are composed of a string (that is used as a key to store the value), a value composed of a binary object, and a timestamp (etag) (that indicates the last write on this blob). In addition of Get and Put methods, blobs whose key shares a given prefix can be listed. Optimistic non locking atomic read-modify-write operations can be implemented using a timestamp matching condition: a write succeed if and only if the timestamp of the storage matches the one provided by the write operation.

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.

3 Proposed implementation

Our prototype consists in 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 jobs. Workers are stored in a pool and regularly ping the queues to acquire a message. Once it has acquired a message, a worker starts running the service related to the queue where the message was stored, and the message becomes invisible till the job is completed or timeouts. Overall, we use $P + \sqrt{P} + 1$ processing units in the services described below.

The **SetUp Service** generates P split datasets of N/P points in each and put them into the BlobStorage. It is also generating the original shared prototypes which are also stored in the BlobStorage.

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

Once completed, it pushes P messages in the queue corresponding to the "Map Service". Each message contains a jobId related to a split dataset to be processed and a groupId used in the "Reduce Service". It also pushes $\sqrt{P}$ messages in the queue corresponding to the "Reduce Service". In the current implementation, the SetUp service is executed by P workers (PU) as the dataset is randomly generated. For a real application, the SetUp service could be executed by one worker or by several workers, depending on the availability and storage format of the data in the BlobStorage.

When executing a **Map Service** job, a worker first downloads the dataset it is in charged of (once for all). Then the worker loads the initial shared prototypes and starts the computation step of attributing each point in the dataset to the closest prototype. New local prototypes version are build on the fly as the points are being processed. When all the map jobs have been completed, reducers must be notified to aggregate the results. We considered two designs for this synchronization process.

A first solution consists in using the atomic read-modify-write operation to store a shared counter in the BlobStorage. Each worker increases the counter once it has completed its iteration. When the counter reaches P, reduction is started. A second possibility uses directly the reduce service. It queries on a regular basis the BlobStorage for the results of the Map Service. When it finds the required results, the reduction is started.

We chose to use the second solution for several reasons. Firstly, this solution is compatible with multiple execution of the same map service job: if a map job is run twice, then the result is written twice at the same place and it does not affect synchronization. Secondly, it could help reducing straggler issues by overlapping latest computations with retrieval of first available results. Thirdly, it handles both synchronization and communication between mappers and reducers. Finally, it proved to be much faster when the number of workers reaches between 50 and 100. Indeed, optimistic non locking read-modify-write schemes are not adapted to high contention situations: for instance, we observed counter updating times up to 20 seconds with 70 workers.

In practice, when all the points of a map service job have been processed once, the local prototypes version and their weights are pushed into the storage according to the following addressing rule: iteration/groupId/jobId. The worker then starts waiting, pinging the shared prototypes of the current iteration until it becomes available.

Each **Reduce Service** job has a groupID, obtained from one of the $\sqrt{P}$ messages pushed into the reduce queue. As explained above, each reducer is listing blobnames with prefix: iteration/groupId in the BlobStorage until it gets all the results expected within this groupId. Map results are downloaded as soon as available. Once all the map results expected have been retrieved and loaded locally, the reducer merges the results into a partial merged result, and pushed this in a partial reduce results directory. One last reducer runs the same process on the partial reduce results directory. Once the $\sqrt{P}$ partial reduce results are retrieved, the last reducer builds a new shared version out of the partial reduce results and pushes the shared prototypes into the storage. After several seconds, every map worker has been pinging the now available shared result blob, and the map step can be run again.

4 Expected performances

The assignment phase of the k-means is perfectly parallelized in [1] and in our implementation; its speedup is P, the maximal value. Global performances are only hindered by the reduction phase. On the high performance distributed memory system used in [1], MPI's reduce is in $O(log(P))$. Neglecting this cost leads to a linear speedup confirmed in the large dataset case by [1].

In the proposed algorithm, the two phases reduce has not a $O(log(P))$ cost but rather a $2\sqrt{P}KDB$ cost (per iteration), where B is the time needed to read a double from the BlobStorage (this cost neglects both the latency of the synchronization process and the issue of aggregated bandwidth, see details below). This adds to the time needed to perform the P parallel map jobs, given by $3NKDF/P$ (per iteration), where F is the time needed to perform one floating point operation. As in [1], increasing N brings the speedup closer to its P limit. However, the main advantage of cloud computing over traditional clusters is the possibility to adapt the processing power to the data. In our case, given a dataset with fixed values of N, D and K, the optimal number of mapper PU is given by $P^* = (3NF/B)^{2/3}$. For this value of P^*, which depends neither on K nor on D, the total processing time consists in one third of mapping and two third of reducing.

5 Experimental results

We tested the performances of the proposed implementation, as well as the limits of the theoretical cost model on synthetic data generated uniformly in the unit hypercube (as pointed out above, timings are independent of the actual data values). We report the results of one out of several benchmarks that show the same general trends. The dataset consists of $N = 500000$ observations in dimension $D = 1000$ (4 GB). We look for $K = 1000$ clusters. The algorithm is run for 10 iterations to get stable timing estimates. The observed floating point performances of our code on a small instance worker in Azure is 669 Mflop/s. The bandwidth of a single threaded worker is about 8 MB/s. Neglecting loading time and memory issues (a small instance has only 1 GB of memory), a sequential 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 mapping PU (P). We report the speedup over the theoretical total running time, the efficiency (speedup divided by $P + \sqrt{P} + 1$, the total number of PU) and the theoretical efficiency predicted by the model.

P	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 P (here 95), for which one third of the total time is spent in the map phase (as predicted by the model). This is lower than the predicted value (170 for the chosen constants) as a consequence of the overestimation of the storage efficiency by the theoretical model. Further investigations are needed to get a more accurate cost model for large P. In particular, we need both to take into account latency and bandwidth aggregation issues. Indeed, Microsoft reports that the total aggregated bandwidth of a PU group tops at 800 MB/s: in our model, B is independent of P, which underestimates the reduce cost for large values of P. In addition, we have observed a large latency between a write completion to the BlobStorage by a worker and the availability of the blob to other workers (up to two minutes in some cases). This latency should be modeled and taken into account to predict the optimal P^*.

Nevertheless, the obtained performances are very satisfactory for a reasonable number of mapping processing units (around 60 for this data size). 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 native high performances API, communication aspects will probably remain a major concern in cloud implementations.

References

[1] I. S. Dhillon and D. 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.

[2] J. Dean and S. 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.

[3] M. Snir, S. Otto, S. Huss-Lederman, W. David, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Boston, 1996.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. 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, USA, 2007. ACM.

[5] H. Liu and D. 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/.

[6] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

K-means on Azure
Supplementary material

Matthieu Durut
Lokad
70, rue Lemercier, 75018 Paris – France
matthieu.durut@lokad.com

Fabrice Rossi
BILab, Télécom ParisTech
LTCI - UMR CNRS 5141
46, rue Barrault, 75013 Paris – France
fabrice.rossi@telecom-paristech.fr

1 Azure base performances

One can find some Azure performance benchmarks on *AzureScope*[1]. Azure performance largely depends on our algorithm, our implementation and the framework (Lokad-cloud) we use. In order to use our model as a predictive tool to determine the optimal number of workers P^* and the expected speedup, we estimated Azure performances by running some isolated jobs and recording time.

1.1 Read bandwidth

We pushed 8 blobs of 8 MB into the storage (this is the size of map results we are going to load back in reducers), and we measured the time spent to retrieve them. For a single worker using a single thread, we retrieved the blobs in 7.79 seconds, implying a 8.21 MB/sec read bandwidth. We tried to use multiple threads on a single worker as advised by AzureScope to speedup the download process, and we found the best read bandwidth was obtained using 5 threads: we retrieved the 8 blobs in 6,13 seconds, implying a 10.44 MB/sec read bandwidth. The multithreads read bandwidth we observed differs from what is achieved in AzureScope. This may be explained by two potential factors: we observed average bandwidth whereas AzureScope observed peak performances and we did not have the same experiment environment. Especially, all the parallel read requests were run on a single container instead of being distributed on several BlobStorage containers.

This benchmark is very optimistic as opposed to our clustering experiments: the bandwidth we estimated was recorded while no other workers were reading or writing into the storage. Other parallel reads and writes can have two effects on the read bandwidth performance. Firstly, they can limit the bandwidth because of aggregated bandwidth boundaries (AzureScope is reporting 800 MB/sec bandwidth for reading operations). Secondly, they can increase the latency between a write completion in the blob storage and the availability of the blob to other workers. When running experiments with a large number of workers reading and writing in parallel, we sometimes experienced blobs already pushed into the storage becoming available only after 1 or 2 minutes. To use our model as a predictive tool to determine P^*, we used a value of bandwidth equals to 8 MB/sec, which is a very conservative assumption that leads us to underestimate communication costs.

1.2 Workers Flops performances

All our experiments were run on Azure small VM that are guaranteed to run on 1.6 GHz CPU. Yet because of virtualization we do not have any warranty in term of floating point operations speed. Therefore, to fit our predictive speedup model, we ran some map jobs (where the number of floating point operations is known) to determine how fast could our algorithm be run on these VM. The code was first run on an Intel Core 2 Duo T7250 2*2GHz using only one core. We estimated our map jobs to be run at approximatively 750 MFlops on average. We then ran some

[1] http://azurescope.cloudapp.net/

of our map jobs in Azure and we get for the same code performance of 669 MFlops on the small VM.

It is worth mentioning that with the number of workers we used (at max 160 mappers), we observed very little differences in processing time between workers. Flops performance may slightly move with time, but at a given time, workers performed at very similar speed.

2 Benchmarks

2.1 Speedup

The next table shows speedups with varying values of P. All the runs listed below were using N = 500000, K = 1000, D = 1000 and I = 10. 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. Yet, since our model does not take into account latency, having very small values of K and D would lead to underestimate communication cost by neglecting latency. We have set K and D to be much smaller than N, but also big enough to neglect latency.

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

P	10	50	60	70	80	90	95	100	110	120	130	140	150	160
Wall 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 Efficiency	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 P (here 95), for which one third of the total time is spent in the map phase (as predicted by the model). One can notice that the speedup curve is relatively flat in a neighborhood of P^*. This value of P^* is lower than the predicted value (170 for the chosen constants) as a consequence of the overestimation of the storage efficiency by the theoretical model. As explained above, our model made the assumption that bandwidth is scalable with the number of workers, and this is no more true when the number of workers is too high. For example, when running 160 workers, some workers spend one minute to load the dataset from the storage, and for 2 experiments we ran with more than 100 workers, one blob took more than 2 minutes to be retrieved by a reducer after being pushed by a mapper.

2.2 Scaleup

Since in our model P^* is not proportional to N anymore (as this was the case using MPI's broadcast), we cannot hope to achieve linear scaleup. Therefore, the scaleup challenge turns into minimizing growth of wall time as N grows, using $P^*(N)$ mappers. Theoretically, as our model gives a processing cost proportional to $\frac{N}{P}$ and a communication cost proportional to $\sqrt{P}$, for $P = P^* = (3NF/B)^{2/3}$ (where B is the time needed to read a double from the BlobStorage and F is the time needed to perform one floating point operation), we should get a global cost proportional to $N^{\frac{1}{3}}$. Therefore, if N is expected to be multiplied by 8, wall time of the algorithm running $P^*(N)$ mappers should be multiplied by 2. We estimated by running experiments for some given values of N the best value $P^*(N)$ and we observed the corresponding amount of wall time. Values of K and D (K=1000 and D=1000) were kept constant for all the experiments to ease comparisons.

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

One can see that $P^*(N)$ does not grow as fast as N. For all the experiments, speedup achieved with P^* is approximatively $\frac{P^*}{3}$, and reducing step took about twice the time spent in map step for this number of mappers, as predicted by our cost model. While the size of the dataset is multiplied by 8, wall time is exactly multiplied by 2, as predicted by the model.

2.3 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 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.