A Discussion on Parallelization Schemes for Vector Quantization Algorithms

Matthieu Durut[1,3], Benoit Patra[2,3], Fabrice Rossi[4]

1- Telecom ParisTech - INFRES, 46 rue Barrault - Paris - France

2- Université Pierre et Marie Curie - LSTA, 4 place Jussieu - Paris - France

3- Lokad , 10 rue Philippe de Champaigne - Paris - France

4- Université Paris I Panthéon-Sorbonne - SAMM, 90 rue de Tolbiac - Paris - France

Abstract. This paper studies parallelization schemes of the Vector Quantization algorithm in order to obtain time speed-ups using distributed resources. We show that the most intuitive parallelization scheme does not lead to better performance than the sequential algorithm. Another distributed scheme is therefore introduced which recovers the expected speed-ups. Then, it is improved to fit implementation on distributed architectures where communications are slow and inter-machines synchronization too costly. The schemes are tested with simulated distributed architectures and, for the last one, with Microsoft Windows Azure platform obtaining speed-ups up to 32 VMs.

1 Introduction

Motivated by the problem of executing clustering algorithms on very large datasets, we investigate parallelization schemes of the Vector Quantization (VQ) method (also called online k-means). This procedure is known for its good statistical properties but, as any on-line procedure, it is inherently slow. In this paper, we assume to have access to a satisfactory sequential implementation of the VQ algorithm. Then, our goal is to distribute efficiently this algorithm in order to bring speed-ups, i.e., gain of time execution brought by more computing resources compared to a sequential execution. Such theoretical parallel algorithms are developed in [**?**]. However, they may cover many different practical implementations. Practical considerations lead us to the present analysis.

The VQ technique computes a summary of a dataset $\{\mathbf{z}_t\}_{t=1}^n$ of d dimensional samples using the following iterations on a vector $w = (w_1, \ldots, w_\kappa) \in (\mathbb{R}^d)^\kappa$, whose components are called the prototypes,

$$w(t+1) = w(t) - \varepsilon_{t+1} H\left(\mathbf{z}_{\{t+1 \bmod n\}}, w(t)\right), \quad t \geq 0. \tag{1}$$

In the equation above, $w(0) \in \left(\mathbb{R}^d\right)^\kappa$, the ε_t are positive reals called *steps* and $H(\mathbf{z}, w)$ is defined by

$$H(\mathbf{z}, w) = \left((w_\ell - \mathbf{z})\, \mathbb{1}_{\left\{l = \operatorname{argmin}_{i=1,\ldots,\kappa} \|\mathbf{z} - w_i\|^2\right\}}\right)_{1 \leq \ell \leq \kappa}, \quad \mathbf{z} \in \mathbb{R}^d \text{ and } w \in (\mathbb{R}^d)^\kappa. \tag{2}$$

The mod operator stands for the remainder of an integer division operation. For a discussion on ties that may appear in equation (2) and for a satisfactory almost sure convergence theorem of the VQ procedure, the reader is referred to [**?**]. Remark that the term $H(\mathbf{z}, w)$ is accessible with a simple distance computation and can also be thought as a sample of the gradient of the clustering criterion, namely the distortion (see for instance [**?**]). Therefore, the VQ technique also belongs to the class of stochastic gradient descent algorithms.

This paper follows the VQ ideas presented in [**?**]. We assume having access to M computing entities, each of them executing concurrent VQ procedures. These executions are performed on a dataset, split among the local memory of the computing instances, and represented by the sequences $\{\mathbf{z}_t^i\}_{t=1}^n$, $i \in \{1, \ldots, M\}$. The prototype iterations computed by the multiple VQ techniques are denoted by $\{w^i(t)\}_{t=0}^\infty$ and called *versions*. We use the following normalized criterion to measure the speed-up ability of our investigated schemes.

$$C_{n,M}(w) = \frac{1}{nM} \sum_{i=1}^{M} \sum_{t=1}^{n} \min_{\ell=1,\ldots,\kappa} \left\| \mathbf{z}_t^i - w_\ell \right\|^2, \quad w \in (\mathbb{R}^d)^\kappa. \tag{3}$$

Algorithms are tested using simulated distributed architecture and synthetic vectorial data[1] in this paper. In addition, it is organized as follows. First, Section 2 provides empirical evidences that the most simple scheme cannot bring speed-ups. Then, some insights to explain the previous non satisfactory situation are provided in Section 3. Consequently, we design a new scheme and prove by practice its ability to bring speed-ups. Finally, in Section 4, we present an asynchronous adaptation of this latter scheme which fits better slow communication architectures such as Cloud Computing.

2 A first distributed scheme

Our investigation starts with the most intuitive parallelization scheme given by the system of equations (4). All versions are set equal at time $t = 0$, $w^1(0) = \ldots = w^M(0)$. For $i \in \{1, \ldots, M\}$ and $t \geq 0$, we have the following iterations:

$$\begin{cases} w_{temp}^i = w^i(t) - \varepsilon_{t+1} H\left(\mathbf{z}^i_{\{t+1 \bmod n\}}, w^i(t)\right) & \\ w^i(t+1) = w_{temp}^i & \text{if } t \bmod \tau \neq 0 \text{ or } t = 0, \\ \begin{cases} w^{srd} = \frac{1}{M} \sum_{j=1}^{M} w_{temp}^j \\ w^i(t+1) = w^{srd} \end{cases} & \text{if } t \bmod \tau = 0 \text{ and } t \geq \tau. \end{cases} \tag{4}$$

The averaging phase described by the braced inner equations is executed only whenever τ points have been processed by every concurrent processors. It produces a shared version of the prototypes, namely w^{srd}, which is broadcasted

[1] Source code is available at the address `http://code.google.com/p/clouddalvq/`

to each processing unit. The Figure 1 shows that multiple resources do not bring speed-ups for convergence. Even if more data are processed, no gain in term of wall clock time is provided using this distributed scheme.

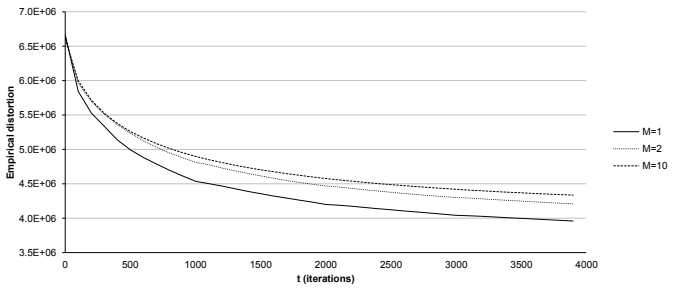

Figure 1: Charts of performance curves for iterations (4) with $\tau = 10$ and different number of computing entities: M = 1, 2, 10.

3 Towards a better scheme

The investigation of the previous non-satisfactory result starts by rewriting both the sequential and the distributed scheme. On the one hand, the sequential VQ iterations (1) can be rewritten:

$$w(t+1) = w(t-\tau+1) - \sum_{t'=t-\tau+1}^{t} \varepsilon_{t'+1} H\left(\mathbf{z}_{\{t'+1 \bmod n\}}, w(t')\right), \quad t \geq \tau. \quad (5)$$

On the other hand, if $t \bmod \tau = 0$ and $t > 0$ then, for all $i \in \{1, \ldots, M\}$, iterations (4) write

$$w^i(t+1) = w^i(t-\tau+1) - \sum_{t'=t-\tau+1}^{t} \varepsilon_{t'+1} \left(\frac{1}{M} \sum_{j=1}^{M} H\left(\mathbf{z}^j_{\{t'+1 \bmod n\}}, w^j(t')\right) \right) \quad (6)$$

Assuming that $w^j(t') \approx w^i(t')$, for all $(i,j) \in \{1, \ldots, M\}^2$ and $t' \geq 0$, the mean in parenthesis is an estimator of the gradient of the distortion at $w^i(t')$. Consequently, the two algorithms can be thought as stochastic gradient descent procedures with different estimators but driven by the same learning rate which is given by the sequence $\{\varepsilon_t\}_{t=1}^{\infty}$.

The convergence speed of a non-fixed step gradient descent procedure is essentially driven by the decreasing speed of the sequence of steps (see for instance [**?**]). The choice of this sequence is subject to an exploration/convergence trade-off. Since the two procedures above share the same learning rate with respect to the iterations $t \geq 0$, they share the same convergence speed with respect to the

wall clock time (time measured by an exterior observer). Yet, the distributed scheme of Section 2 has a much lower learning rate with respect to the number of samples processed, favoring exploration to the detriment of the convergence. The multiple resources therefore lead to better exploration but to similar convergence speed with respect to wall clock time.

Remind that we suppose to have a satisfactory VQ implementation. Consequently we seek for a distributed scheme that have the same learning rate evolution in term of processed samples and which convergence speed with respect to iterations is accelerated.
Set,

$$\Delta^j_{t_1 \to t_2} = \sum_{t'=t_1+1}^{t_2} \varepsilon_{t'+1} H\left(\mathbf{z}^j_{\{t'+1 \bmod n\}}, w^j(t')\right), \quad j \in \{1, \ldots, M\} \text{ and } t_2 > t_1 \geq 0. \tag{7}$$

At time $t = 0$, $w^1(0) = \ldots = w^M(0) = w^{srd}$. For all $i \in \{1, \ldots, M\}$ and all $t \geq 0$, consider the distributed scheme given by

$$\begin{cases} w^i_{temp} = w^i(t) - \varepsilon_{t+1} H\left(\mathbf{z}^i_{\{t+1 \bmod n\}}, w^i(t)\right) & \\ w^i(t+1) = w^i_{temp} & \text{if } t \bmod \tau \neq 0 \text{ or } t = 0, \\ \begin{cases} w^{srd} = w^{srd} - \sum_{j=1}^{M} \Delta^j_{t-\tau \to t} \\ w^i(t+1) = w^{srd} \end{cases} & \text{if } t \bmod \tau = 0 \text{ and } t \geq \tau. \end{cases} \tag{8}$$

The results of the simulations are displayed in the charts of Figure 2. The charts show that substantial speed-ups are obtained with distributed resources. The acceleration is greater when the reducing phase (described by the braced inner equations) is frequent. Indeed, if τ is large then more autonomy has been granted to the concurrent executions, they could be attracted to different regions that would slow down the consensus and the convergence.

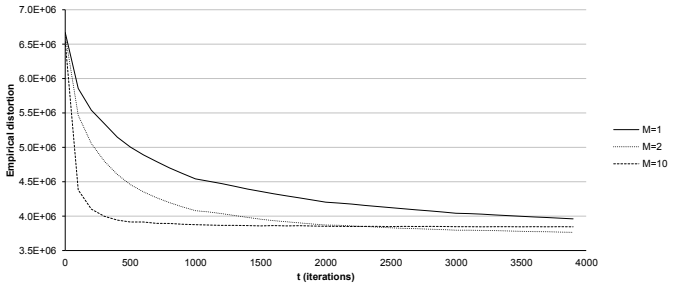

Figure 2: Charts of performance curves for iterations (8) with $\tau = 10$ and different number of computing entities: M = 1, 2, 10.

4 A model with stochastic delays

The previous parallelization schemes do not deal with communication costs introduced by update exchanges between machines. In the context of cloud computing, no efficient shared memory is available and these costs introduce delays. The effect of delays for parallel stochastic gradient descent has already been studied (see for instance [**?**]). However in the previous paper the computing architecture is endowed with an efficient shared memory. Moreover, the unreliability of the cloud computing hardware introduces strong straggler issues and makes the synchronization process inappropriate. In this subsection, we improve the model of iterations (8) with random communication costs that follows a geometric distribution and we remove the synchronization process of reducing phase, resulting in the more realistic iterations (9) below. For each time $t \geq 0$, let $\tau^i(t)$ be the latest time before t when the unit i finished to send its updates and received the shared version. At time $t = 0$ we have $w^1(0) = \ldots = w^M(0) = w^{srd}$, and for all $i \in \{1, \ldots, M\}$ and all $t \geq 0$,

$$\begin{cases} w^i_{temp} = w^i(t) - \varepsilon_{t+1} H\left(\mathbf{z}^i_{\{t+1 \bmod n\}}, w^i(t)\right) & \\ w^i(t+1) = w^i_{temp} & \text{if } t \neq \tau^i(t) \\ w^i(t+1) = w^{srd}(\tau^i(t-1)) - \Delta^i_{\tau^i(t-1)\to t} & \text{if } t = \tau^i(t) \\ w^{srd}(t+1) = w^{srd}(t) - \displaystyle\sum_{j, t=\tau^j(t)} \Delta^j_{\tau^j(\tau^j(t-1)-1)\to\tau^j(t-1)} & \end{cases} \tag{9}$$

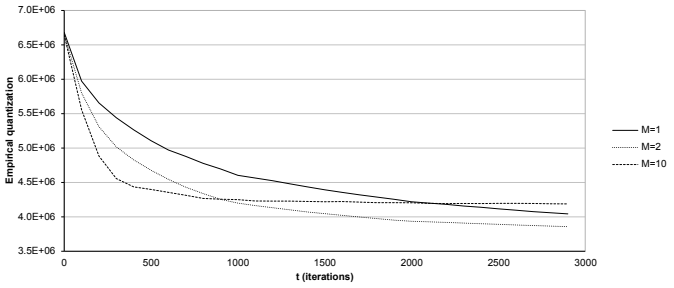

Figure 3: Charts of performance curves for iterations (9) with $\tau = 10$ and different number of computing entities: M = 1, 2, 10.

There are no more synchronization between processing units: each machine uploads its updates and downloads the shared version as soon as its previous uploads and downloads are completed. A dedicated unit permanently modifies the shared version with the latest updates received from the other machines without any synchronization barrier. The figure 3 shows that the introduction of small delays and asynchronism only slightly impacts performances, compared

to the scheme given by equations (8). The figure 4 shows the results obtained by our cloud implementation of the iterations (9) using 32 real processing units. A future paper will describe more precisely this cloud implementation.

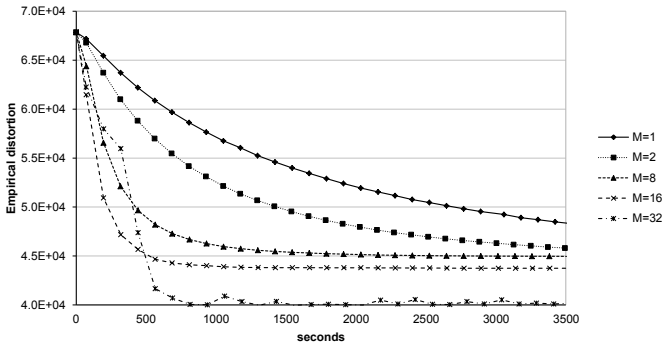

Figure 4: Charts of performance curves for iterations (9) on our cloud implementation and different number of computing entities.

5 Conclusion

In this paper we show that the naive parallelization scheme proposed in section 2 does not provide better performance than the sequential scheme. This surprising result derives from the fact that our first parallel scheme leads to a decrease of the learning rate per data points processed. We therefore propose a new parallelization scheme relying on asynchronous updates of a common "shared version". This latter algorithm is very well suited for parallel computation on slow communication networks such as cloud computing platforms. Our implementation on Azure show significant scale-up, up to 32 machines.