Clustering Low-Cost, Cloud-Based Servers to Solve Intensive, Parallel Computations

Nuno B. Brás and Gonçalo Valadão

Instituto de Telecomunicações (IT)
and Universidade Autónoma de Lisboa (UAL)
Lisboa, Portugal.
Email: {nuno.bras, gvaladao} @lx.it.pt

***Abstract*—This paper advocates the usage of available cloud based services for intensive parallel floating point computations, based on clusters of servers installed with Graphical Processing Units (GPUs), in order to run low-cost, High Performance Computing (HPC) tasks.**

It is described a cluster of multiple servers installed with GPU units and running open-source software which works as an easy to scale, low cost platform with centralized Master-Slave control, able to turn on/off the Slaves (GPU server machines) as needed.

The objective is to show that, nowadays, parallelization architectures allow to decrease dramatically the computing time while maintaining or even reducing the intrinsic costs (€/hour) comparing with traditional approaches taken by researchers and industry practitioners.

A computer vision classification problem is used as a toy problem: a deep neural network training using a standard image data library, through known and easy to use parallelization packages. For this problem, comparisons on time performance, scalability and costs, with regard to other traditional approaches, are presented. In this particular case, the computing time was reduced almost 30 times while attaining however, an unexpected cost reduction.

I. Introduction

In the last years a strong changing trend from physical, in-house, infrastructures to cloud based ones has been made within many companies, research institutes, and other organizations, taking most of their software platforms to the cloud.

In research, the ability to do simulations using High Performance Computing (HPC) was not easy for a small team of researchers without a large computing center at their disposal. If the infrastructure was not proprietary, researchers had to have renting hours in super-computers or in similar infrastructures.

In contrast, cloud services are nowadays easily rented without the cost of a large infrastructure or without the complicated procedure for renting a space in a scientific super-computer, allowing to create a budgeted solution with a considerable high performance, for demanding problems in machine learning, optimization or signal processing, for instance.

GPUs are specially well suited to HPC tasks [1] and have firstly arrived as local proprietary installations. However, GPU renting per hour has changed this paradigm. There are available nowadays several services based on GPUs for HPC: Amazon Web Services (AWS) [2], NIMBIX [3], Penguin Computing [4] among others. Typically, these are called Infrastrucure as a Service (IaaS), that is, services where servers are totally available through command line (or other) and the user has (almost) full control over each server.

Typical costs for an On Demand GPU based server are in the range of 2€ to 4€/hour/server, depending on the kind of GPU Servers. Some cloud services (such as Google's Preemptible VMs [5] and AWS Spot Services [2]) offer a new modality which presents much lower prices. Essentially it consists of serving unused processing power and its price can be much lower than of a normal, reserved, GPU server instances [6]. With a typical average cost of 0.06 €/hour (for AWS and without much fluctuation), the drawback of this kind of server instances is that data and instance itself could be turned off at any moment, although in practice it could take more than a week for that to happen. Besides, this kind of instances are not saved, which means they should be configured from the ground each time one intends to use them. In this context, it is essential to have an automated system that is able to build the cluster from scratch, allocating server instances and managing their work, enabling also a shared infrastructure among servers. With this architecture and corresponding constrains in mind, we have tested the use of a toy problem to understand the advantages of such approach.

The paper is organized as follows: in the next section we present the cloud computing cluster and its architecture. Then, in Section 3, we provide a description of the toy problem, applied to test the cluster performance. In Section 4 performance and assessment results are presented, benchmarked with two standard approaches: a personal laptop approach and cloud based CPU only server. Also in Section 4 conclusions are discussed, focusing in particular the cluster limits and further improvements.

II. The Cluster Architecture

In this section we address three fundamental issues, namely: i) the chosen cloud based service; ii) the cluster architecture; and iii) the software language and used packages.

A. The Cloud-Based Service

As referred above, there are already several cloud-based services that can be used to run GPU parallel processing. From those, we have chosen AWS mainly for two reasons: reliability and the offering of a spot business model. Regarding the former, AWS is one of the world's leading provider of cloud services, as well as one of the pioneers. A multitude of world-class organizations (including government organizations) run much of their software through AWS. Regarding the latter, as already referred to above, it translates into an extreme drop in price, at the cost of allowing the data and programs to be, suddenly, gone without warning.

B. *The Architecture*

We define a master-slave structure for the employed cluster, where the master server is responsible for coordinating slave servers and making data persistent across them. This paper adopts StarCluster [7] which exactly implements such a system. The master server does not need to have strong processing abilities, but it must be able to run scripts to manage the slave servers, namely, in terms of macro tasks such as the hyper-parameters determination, that will be referred to in section III. As we embrace the usage of plot instances, as already referred, the master server must be reserved and solid, in order to keep in disk the simulated data in case of a problem with any of the slave servers. The chosen system also should ensure the existence of a frequent backup of intermediate results from the slaves to the master, to face the possibility of unexpected shutdowns. For these reasons, the master was chosen to be a AWS CPU Only Server instance. Since the first AWS instance is free (if sufficiently small - AWS Free Tier), costs are only applied over the usage of spot instances (and storage, eventually).

To benchmark the defined solution with other options we also consider (i) a personal laptop and (ii) a unique CPU Server without GPUs.

Setup and Data Sharing: In terms of usage, the system should be prepared in the beginning, building (or instantiating) from scratch all the slave servers. Notice that the servers are created based on linux images previously parameterized, with every programs/packages pre-installed in order to run the code, particularly all the GPU packages and drivers. Data and Code should be also previously available in the master, which in turn should share them "immediately" among slave servers when they boot. In order to share data across all slave servers from the master, a shared network drive is created from master to all slaves at boot time.

Finally, it is important that any modification in code (or data) in a local laptop (where primary tests are run) could be interchanged with the master. A cloud-based file synchronization service (Dropbox) [8] was previously configured in the master, inside its shared network drive in order to spread info across all slave servers, allowing to have an on-line copy of all data and code. Code synchronization takes less than 5 seconds. (A similar mechanism would be feasible with a version control system, such as git [9], but data will have to be synchronized by other method).

In order to manage all this infrastructure the program StarCluster [7] was used. This allows us to easily build and parameterize all servers from a unique interface. Although this setup is done in parallel for all machines it could take up to 15 minutes to have the system prepared.

C. *Software Languages and Packages*

In a Infrastructure as a Service (IaaS) cloud based service it is up to the user to choose the programming language. Python has been one of the most popular choices, since it is freely available (and open source), it is a high level language that allows a very fast development, and has a huge range of application specific communities developing all types of useful modules.

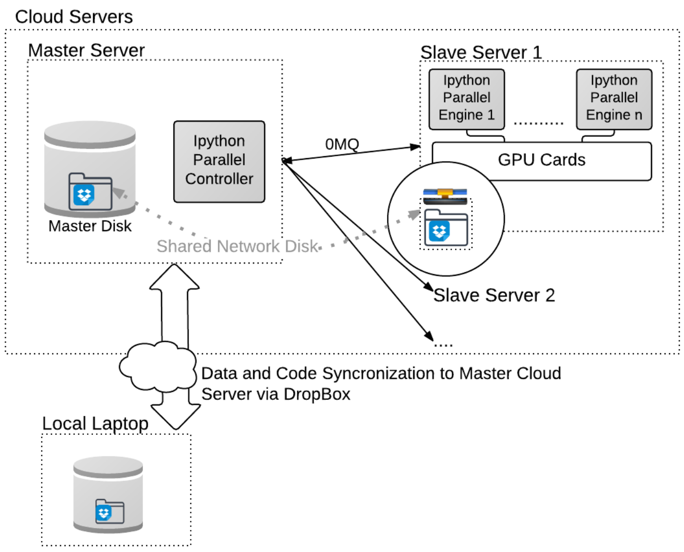
Cloud Servers
Master Server
Slave Server 1
Ipython Parallel Engine 1
....
Ipython Parallel Engine n
Ipython Parallel Controller
0MQ
GPU Cards
Master Disk
Shared Network Disk
Slave Server 2
....
Data and Code Syncronization to Master Cloud Server via DropBox
Local Laptop

Figure 1. System Architecture: the cloud infrastructure and the used components are represented.

Regarding our application with a master and slaves architecture, it is possible to write in python the execution code but also the master scripts to control slaves, using iPython Parallel module. This brings a seamless experience across implementation of execution code and master scripts.

For each CPU in each server that composes the cluster it is created an iPython Parallel Engine [10], and in the master, it is initialized an iPython Controller [10]. This is done automatically in each cluster initialization. Now we can send direct iPython parallel messages (using pre-configured zeroMQ [11]) from the master to any set of engines instances running in the slaves.

There are two kinds of direct messaging between the master and the engines: (i) sending the instantiated objects to each engine; (ii) sending remote instructions that are evaluated locally. We have used the second one. Each Server have CUDA installed (if GPUs are available) and Theano [12]. Theano is always used, even when GPUs are not available, and Theano itself chooses what should be used: CPU or GPUs.

Our cloud-based virtual setup is schematically represented in Fig. 1.

To get a specific example of the achieved flexibility and its easiness of implementation a toy problem is next explained

Table I
LAPTOP, AWS-EC2 ON-DEMAND INSTANCES AND AWS-EC2 SPOT INSTANCES

Equipment	Cost	Description
Laptop/Desktop Computer	800 €(Total Acquisition Value)	OSX i5, 2.4 GHz CPU, 10GB RAM, SSD disk
Single CPU Instance	Free (AWS Free Tier)	Linux t2.micro, 1 2.5 GHz vCPU, 1 GB RAM
CPU cluster on demand Instance	0.017 €/On-Demand Instance Hour	Linux t2.micro, 1 2.5 GHz vCPU, 1 GB RAM
GPU Cluster on demand Instance	0.702 €/On Demand Instance Hour	Linux g2.2xlarge, 8 CPU, 4GB, SSD Disk, GPU K520
CPU Cluster Spot Instance	0.016 €/Instance Hour	Linux c3.large, 3 CPU, 4GB, SSD Disk
GPU Cluster Spot Instance	0.065 €/Instance Hour	Linux g2.2xlarge, 8 CPU, 4GB, SSD Disk, GPU K520

and tested.

III. THE TOY PROBLEM

A. The Underlying Challenge

The underlying problem is a character classification problem, over the MNIST dataset [13]. This problem was often used to benchmark algorithms because it was already deeply studied. The MNIST has a training set of 60 000 characters, a testing set of 10 000 characters, having each image 28 $\times$ 28 pixels.

Here, the problem is solved using a deep neural network described in [14], composed of the following layers:

- 2 layers, each composed of a convolutional layer and a pool layer (2×2) (half size image reduction in each Pool layer). This takes as input 784 pixels, and at the end of the last layer, one have 100 values;
- 1 Fully Connected Layer using a ReLU activation function, input layer size: 25 values and an output of 25 values;
- 1 Final Softmax Layer giving statistical meaning to the obtained results, with a 25 input arrays and with 10 output results (one for each numerical digit);

where a similar Network was used to classify the ImageNet Dataset. The resulting optimization problem is regularized using an L1 kernel, although other methods could be used such as Dropout techniques.

The network weights are trained through the use of mini-batches with variable sizes over the training dataset. The degree of confidence is calculated over the testing set. An epoch is defined as a complete training step where a set of mini-batches equivalent to all the training data set has been used. Each epoch training is based on linear matrix calculations, easily parallelized through GPUs. Most of the calculations can be done using GPU CUDA package and performance can be ultimately increased from 4 to more than 80 times for matrix operations [15], [16]. Hyper-parameter tunning is intrinsically distributable, through several servers.

B. The Toy Problem: Hyper-Parameters Tuning

A very large set of hyper-parameters surrounds the fine tuning of a deep neural network, namely, the learning rate, the early stopping, mini-batch size among many others [17]. The resulting optimization problem to be solved is regularized using a L1 kernel. Although other methods could be used, such as Dropout techniques, a hyper-parameter emerge from the regularization tuning.

The proposed Toy Problem is a simple hyper-parameter optimization, training networks for a set of possible hyper-parameter values. Since each training could be done in parallel, this is a perfect problem to use the proposed distributed computing power for separate runs, together with the GPU ability to parallelize stochastic gradient calculation steps.

This problem is suited to be solved in a GPU server based environment and some characteristics could be used to show the performance of the system.

IV. RESULTS AND CONCLUSIONS

As said, to benchmark the defined solution with other options we consider (i) a personal laptop, (ii) a CPU cloud Server with and (iii) without GPUs and (iv) a cluster of CPU only servers, with the same architecture as the GPU cluster (see Table I).

The following data was used to benchmark performance and costs:

- Throughput is defined as training epochs per minute;
- Calculation costs and execution time are compared in a set of 1000 training epochs, using total costs in Euros;

A. Throughput Benchmarking

Throughput tests are shown in Figure 2. This graphic allow

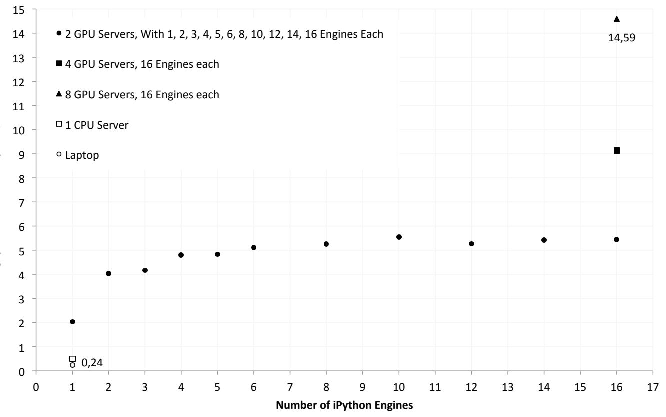

Figure 2. Throughput in Number of Epochs/Minute

us to take some conclusions: it is possible to see that the performance of a laptop (0,24 epochs/min) or a unique CPU server (0,49 epochs/min) is definitely slower than a simple Server with a GPU (2,03 epochs/min), even considering the intrinsic master-slave communication delays, due to the GPU acceleration of the matrix calculation. Another important conclusion has to do with the "log style" performance evolution, as the number of engines are added for the same number of servers, showing that although the CPUs (16) are not completely in use, the shared memory, disk and GPUs degrade performance before all the available engines are working (one per CPU), actually starting to degrade the overall performance. This means that a balanced number of engines per server should be considered to get the best performance: one could see that, for the same number of 16 engines the performance increase from 5,44 epochs/min with 2 Servers to 9.13 with 4 and then to 14,59 with 8.

The best performance was achieved by a a set of 8 servers with GPUs, each working with 8 iPython engines, in a total of 64 iPython engines, sharing, a set of 3000 GPU cores per server. It can be seen that comparing the throughput of a laptop (0,24 epochs/min) or a unique CPU server (0,49 epochs/min) with this architecture (14,59 epochs/min) took an increase of performance of respectively 60,8 and 29,4 times. This increase of performance was, of course, expected. What was not expected was the price evolution, discussed in the next session.

B. Costs And Execution Time Benchmark

In order to understand how the price moved from a simple CPU computer to a GPU Cluster, we have created a graph where the cost in € and the time spent is used to map each setup. This is shown in Figure 3.

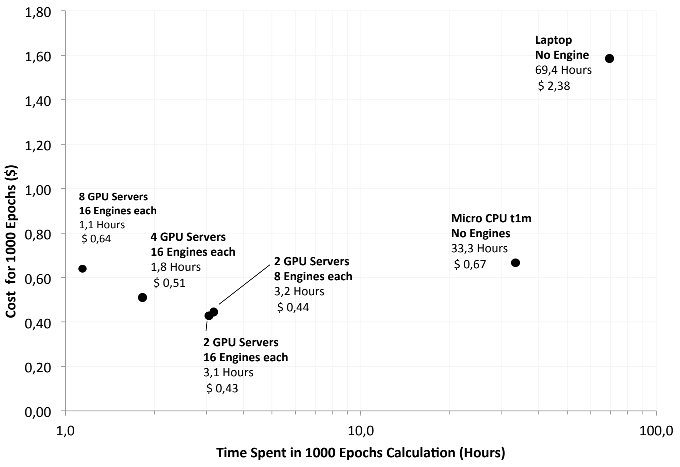

Figure 3. Cost *vs* Time map for each setup, considering 1000 training epochs.

It was expected that the cost would increase as time decreases. That would make sense since time improvements on processing is normally an expensive asset. What happened was that the cost actually decreased when a GPU cluster based setup was used, from 0,67 €, when using a unique CPU Server to 0,64 €. Of course the laptop could also be compared, having a reduction of almost 4 times in cost, when considering its price amortized in 4 years. However, it wouldn't be a fair comparison, since the laptop is typically a necessary asset, even if no cloud simulations are carried out.

C. Conclusions

This paper advocates the usage of cloud computing services for high performance computing tasks. Supporting this view are the experimental results which are summarized in Figures 2 and 3. The former shows the throughput of solved tasks (epochs) per minute vs number of engines, and the latter shows the cost of solving 1000 tasks *vs* the time spent with it. From both it is evident that it is highly beneficial to use multiple engines equipped with GPUs, *i.e.* the parallelization they allow brings an exceptional improvement on the computational task at hand. Furthermore, Fig. 2 shows clearly that this improvement has its limitations, *i.e.*, from a certain point, increasing the number of parallel engines does not bring any noticeable value. Figure 3 confirms the benefits of using parallel engines and, furthermore, it shows that, with the existing cloud computing services, it actually can cost less to use clusters of engines instead of using one CPU-only engine. Furthermore, it shows that increasing the number of engines in the cluster not only brings down the time spent in the computation, but also decreases associated costs for all presented setups.

Also, the creation of such a cloud based cluster allowed the authors to show how it could be centrally managed using 3rd party open-source software.

References

[1] M. Silberstein, "GPUs: High-performance accelerators for parallel applications: The multicore transformation (ubiquity symposium)," *Ubiquity*, vol. 2014, no. August, pp. 1:1–1:13, Aug. 2014.
[2] "Amazon Web Services," http://aws.amazon.com, accessed: 2015-06-04.
[3] "Nimbix," http://www.nimbix.net/, accessed: 2015-06-04.
[4] "Penguin Computing," http://www.penguincomputing.com/, accessed: 2015-06-04.
[5] "Theano," http://googlecloudplatform.blogspot.pt/2015/05/Introducing-Preemptible-VMs-a-new-class-of-compute-available-at-70-off-standard-pricing.html, accessed: 2015-06-04.
[6] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing amazon EC2 spot instance pricing," *ACM Trans. Econ. Comput.*, vol. 1, no. 3, pp. 16:1–16:20, Sep. 2013.
[7] "Star cluster," http://star.mit.edu/cluster/, accessed: 2015-06-04.
[8] "Dropbox," https://www.dropbox.com/, accessed: 2015-06-04.
[9] "git," https://git-scm.com/, accessed: 2015-06-04.
[10] "ipython," http://ipython.org/ipython-doc/stable/parallel/parallel_intro.html, accessed: 2015-06-04.
[11] "zeromq," http://zeromq.org/, accessed: 2015-06-04.
[12] "Theano," http://deeplearning.net/software/theano/, accessed: 2015-06-04.
[13] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.
[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.
[15] F. Vazquez, E. Garzon, J. Martinez, and J. Fernandez, "The sparse matrix vector product on gpus," in *Proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering*, vol. 2, 2009, pp. 1081–1092.
[16] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.
[17] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 437–478.