

# Implementation of Coded Elastic Computing

Raj Patel & Jake Maschoff

## Abstract

Cloud computing is an ever-growing platform. Using multiple servers have helped distribute computational heavy tasks. Machine learning, in particular, is benefiting greatly from cloud computing. Distributing machine learning tasks across multiple nodes can reduce the training time. However, the distribution of the task across multiple nodes is challenging. Nodes can leave and join the task unexpectedly. One of the proposed methods for machine learning tasks is *coded elastic computing* [1]. In this paper, we summarize our implementation of the proposed method.

## Introduction

Coded elastic computing allows a user to train machine learning models over preemptable machines [1]. In this proposed framework, a machine can leave or join the task unexpectedly. The framework is different from others (like error-correcting codes) because coded elastic computing can adapt to machines leaving or joining at run time. One of the benefits of the proposed framework is that it reduces the workload of a single machine. This method can also be used with learning tasks that use the number of machines as a hyper-parameter.

This framework is achieved through codes as mentioned before. As long as there are less than  $L$  preempted machines, the coded data will allow for the original data to be carefully calculated codes. In the example provided in the paper, they used 6 machines. The way that the data is encoded is shown in Figure 1. The data is sub-sectioned into 3 parts since in this scenario  $L = 3$ . Redundancy is removed in the encodings by having  $P$  sub-sections (where  $P$  represents the number of machines in the network once it starts) but each machine only uses  $L$  of them. If new machines join the network, they download either subblocks lost by previously preempted machines or a new linear combination of the data. The coefficients are chosen in the linear combinations randomly but predetermined.

It is important to allow for machines to come and go from the network because cloud service providers are then able to provide a cheaper level of services. AWS, for example, offers an Amazon Spot Instances which can be up to 90% cheaper than normal prices, but only provides services when at a low system load [1, 2]. When using such a framework, multiple machines will come and go throughout the entire machine learning algorithm. It is important to account for this in an efficient manner.

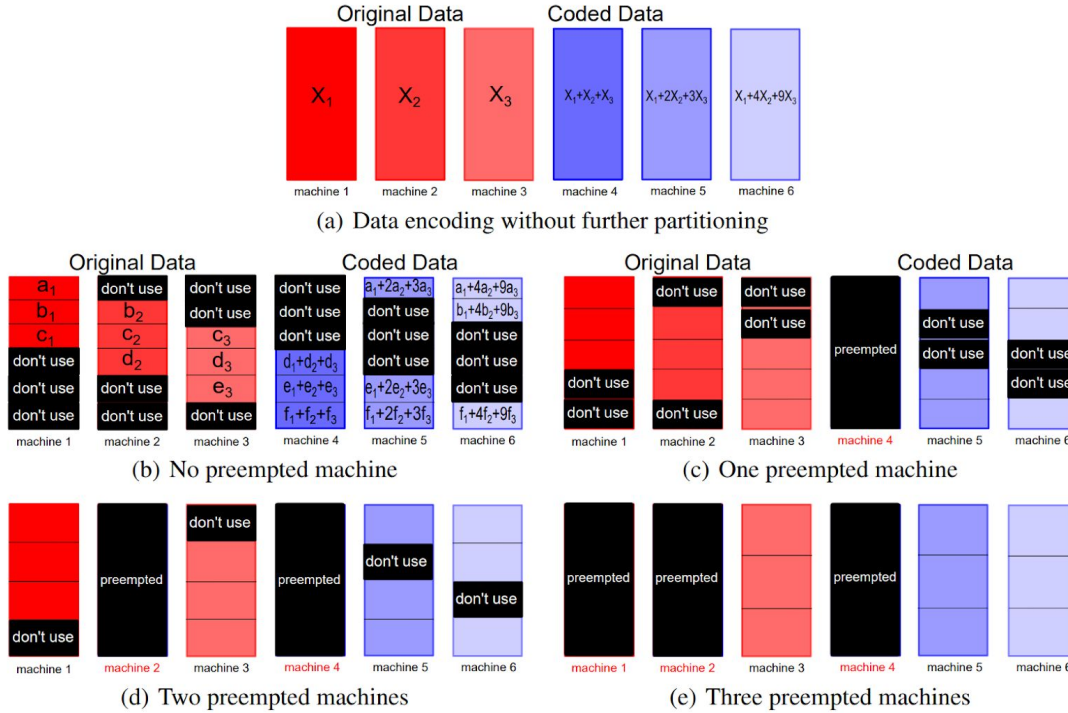


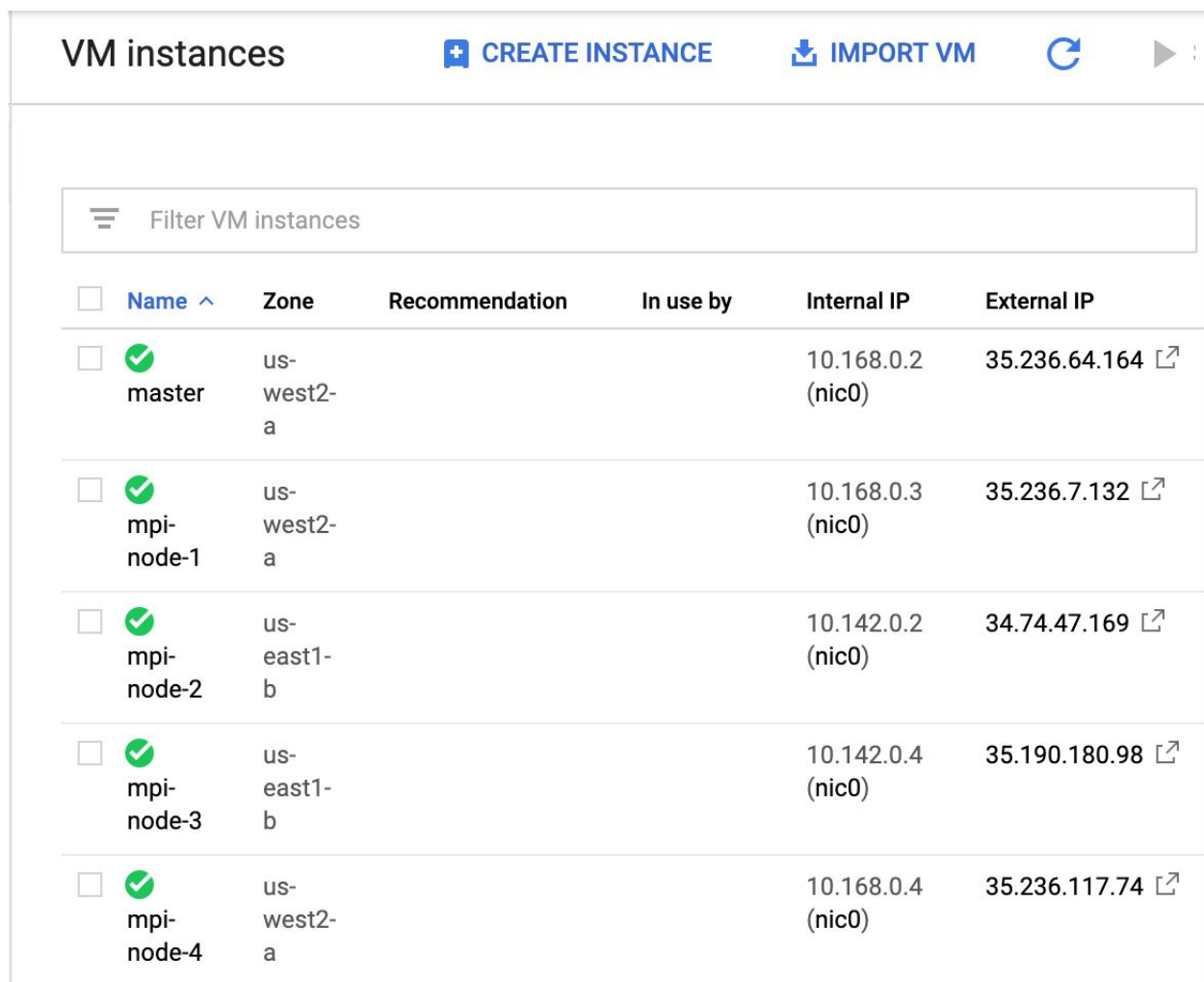
Figure 1. Data is distributed to each machine in elastic coded machines. As the machines leave, elastic coded machines adapt at run-time distribute data accordingly.

## Implementation

During our implementation, issues arose immediately. The paper used a framework called Apache Reef for their communication framework, which was uncompileable (even the GitHub repository reported build errors). This was unfortunate because Apache Reef was an Elastic Group Communication (EGC) framework, which allowed for machines to leave the message network whenever they pleased. This functionality was greatly needed for the implementation because machines need to preemptively leave and come back into the network to prove our coded elastic communication is working.

Since Apache Reef was unavailable, we researched for a different message passing interface (MPI) EGC. Unfortunately, in our research, we could not find other MPIs that offered this capability. With no framework with this functionality, it seemed we needed to develop our own framework. After speaking with the TA, Nick, MPI4PY seemed like a good alternative library for communicating between different processes/hosts. Our own preemption framework would need to be built on top of this, which will be addressed later in the paper.

Compiling and building an MPI4PY instance was easy compared to Apache REEF. Once it was compiled, having MPI4PY communicate across multiple hosts was challenging. Our implementation consists of four “worker” nodes and one “master” node that needed to share data with each other as shown in Figure 2.



The screenshot shows the 'VM instances' page in the Google Cloud console. At the top, there are buttons for 'CREATE INSTANCE', 'IMPORT VM', and a refresh icon. Below the header is a search bar labeled 'Filter VM instances'. The main content is a table with the following columns: Name, Zone, Recommendation, In use by, Internal IP, and External IP. There are five rows, each representing a VM instance. The first row is the 'master' instance, and the next four rows are worker instances labeled 'mpi-node-1' through 'mpi-node-4'. Each instance has a green checkmark icon next to its name, indicating it is running. The 'Internal IP' column shows the IP address and the network interface (nic0). The 'External IP' column shows the public IP address and a link icon to view details.











<input type="checkbox"/> Name ^	Zone	Recommendation	In use by	Internal IP	External IP
<input type="checkbox"/>  master	us-west2-a			10.168.0.2 (nic0)	35.236.64.164 
<input type="checkbox"/>  mpi-node-1	us-west2-a			10.168.0.3 (nic0)	35.236.7.132 
<input type="checkbox"/>  mpi-node-2	us-east1-b			10.142.0.2 (nic0)	34.74.47.169 
<input type="checkbox"/>  mpi-node-3	us-east1-b			10.142.0.4 (nic0)	35.190.180.98 
<input type="checkbox"/>  mpi-node-4	us-west2-a			10.168.0.4 (nic0)	35.236.117.74 

Figure 2. GCloud snapshot of one master and four works.

To begin, we created a master node on the GCloud infrastructure. MPI4PY was installed in this server along with its dependencies. Initially, this started with a simple test configuration. The purpose of this test configuration was to learn and observe how MPI4PY master/slave communication worked. This started with five processes on the same machine, one master and four workers. The master node sent out dummy data to the worker nodes, which then was printed to the console. Once this was working, the configuration across multiple physical servers needed to be set up.

To save time installing MPI4PY along with its dependencies, we cloned our master node into four worker nodes. We also created these four nodes in different regions of the United States to make this project interesting. In order for MPI library to communicate between the master node and worker nodes, SSH keys were needed to be generated for all of the nodes to communicate with each other via SSH. Google Cloud has a metadata configuration which allowed us to share SSH public keys to all servers at once.

We encountered a major bug during running our code across multiple nodes. It seemed we missed installing a major underlying MPI library. To fix this bug, an underlying framework is installed, such as OpenMPI or MPich. This bug would not allow us to launch processes on the remote hosts. The error message was of the form, “bash: unable to find command ortd”. Once OpenMPI was downloaded on all nodes and the appropriate paths set to the /openmpi/bin in the .bashrc file, this issue was solved. MPI4PY runs on top of OpenMPI, which was overlooked.

Another error occurred which stated “Error: unable to perform system call ‘Execve’”. This was caused by formatting the mpiexec command incorrectly. When running mpiexec, we didn’t specify which interpreter to use, such as Python2 or Python3. Everything worked as expected after changing the command to the following:

```
mpiexec --hostfile ./hostfile python3 ~/CodedElasticComputingImplementation/main.py
```

Adding the **python3** lets the hosts know what version of python to use to run the specified code. Also, the --hostfile flag must be added with a path to the host file in order to gain access to the IP addresses of the remote hosts.

Finally, a host file needed to be created for remote hosts communicating with each other. The host file stores the IP (or domain names if available) to each of the nodes with the corresponding number of processes to launch on the remote host. Our host file had the following format:

```
localhost
10.168.0.3
```

10.142.0.2  
10.142.0.4  
10.168.0.4

The localhost is included since the program is run on the master node, and it must be included in the host file so it spawns a process on the machine itself. We excluded the number of processes to run from the host file forcing it to default to a single process.

To prove that the coded implementation works, we decided on implementing an SVM classifier. The dataset used was UCI's credit card dataset. The dataset consisted of 219 data features, 20,000 training examples, and 10,000 test examples. The values represented in this data set were binary values and the classification was also binary. The data set classifies whether a bank customer will default on their credit card payment or not. The dataset was represented in lib linear format which was parsed accordingly.

Initially, we trained the task without any machines leaving to see if we could achieve the same accuracy as running the task on a single machine. Our results were more than satisfactory, as we were able to achieve nearly identical results as what we had received with just a single machine. This is all shown in Table 1. We were able to compare the results to what we had received when we ran a similar model in our Machine Learning course.

Table 1: SVM Single Machine vs Multiple Machines

	Single Machine	Multiple Machines (4 for us)
Average Precision	66.62%	70.25%
Recall	22.43%	19.93%
F1	33.42%	34.05%
Accuracy	80.39%	80.28%

We also added a few of our own ideas into this step. Each worker was sent its own batches of data, where it would first perform cross-validation on the data using 12 different hyper-parameters. Using the best hyper-parameters on the data, it would then use those to train on the data and calculate the weights for the particular iteration. After all of the machines performed their calculations, an average of all the weights was taken as the overall weights for the model. This model was then tested on the entire testing data set.

Once sure that the distributed machine learning was in fact working, we began working on coding according to the Coded Elastic Computing paper. First, though, we had to implement preemptive machines into our code. We had to change our implementation to do so. In order to accommodate for this we began implementing a slight framework that takes in a command-line argument integer with the max number of computers that can possibly leave the network. The command-line argument takes the form:

```
mpiexec --hostfile ./hostfile python3 ~/CodedElasticComputingImplementation/main.py 2
```

If an integer is passed, the network has the possibility of not sending data to that many machines. It does so by sending a particular flag to each of the worker nodes. If the flag is present, the machine will not compute the gradients and not send any data back to the master node. This was made sure to work by running the SVM algorithm multiple times and seeing that nodes would not send back data when told not to.

With the preemption working, we began working on the coding used by the Elastic Coding Computing paper. After much toil and effort, we were unable to replicate this coding technique and still have the machine learning algorithm work. If given more time, we're sure that we could get this working.

## Conclusion

In conclusion, we were able to implement the proposed coded elastic computing framework successfully without the encodings. We did have to tweak the framework in order for it to work since the applications used in their paper would not compile on our machines. Preempted machines were created programmatically in order to overcome the frameworks that would not compile or work. The paper mentioned that they were able to achieve similar results on multiple machines compared to a single machine. In our implementation, we confirmed that we also had similar results running SVM on multiple machines compared to a single machine. However, more time was needed to completely implement the encoding techniques used in the paper. Overall, this project has been very helpful for us understanding coded elastic computing and distributed machine learning. The implementation process did get complicated with bugs, however, we were able to overcome most of them successfully.

## Resources

- [1] Yang, Y., Interlandi, M., Grover, P., Kar, S., Amizadeh, S. and Weimer, M. (2019). *Coded Elastic Computing*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1812.06411> [Accessed 5 May 2019].
- [2] AWS Spot Instances Prices. <https://aws.amazon.com/ec2/spot/pricing/>, 2018.