MPI Learn: distributed training

AUGUST 2018

AUTHOR:
Filipe
Magalhães

CERN EP-CMG

SUPERVISORS(S):

Maurizio Pierini
Jean-Roch Vlimant

Project Specification

MPI Learn is a framework for distributed training of Neural Networks. Machine Learning models can take a very long time to train. This can be improved using parallelism, by distributing the training over several processes and several hardware resources. Implementing parallelism requires expertise and is time consuming.

MPI Learn is aimed at machine learning users, who need to speedup the training of their models. A user should input a model, training and validation data, and tune other training parameters. MPI Learn will internally distribute the training over the specified number of processes, and output results, abstracting all the parallelism from the user.

MPI Learn is intended to be part of a bigger project, MPI Opt which aims to perform hyperparameter optimization, in a distributed fashion. This framework will search for the best hyperparameters in a user defined search space. The search will be parallelized, with several executions of MPI Learn being run in parallel.

MPI Learn is currently implemented and being used in some practical projects. The work developed over the course of this summer focused on optimizing the framework, and analyzing its execution with the objective of increasing performance.

Abstract

MPI Learn is a framework for the distributed training of neural networks. This platform is aimed at machine learning users, who can use it to train models faster, without dealing with the complexity of parallelizing the execution. This framework is implemented and in active use. During the months of July and August, the work developed focused on profiling and analyzing the performance of the execution. Features and improvements were developed based on this analysis.

The profiling was done using varied examples of models and datasets, allowing for the analysis of very different execution behaviours. A single process mode was developed for the framework, to serve as a base for scalability studies. In order to reduce the performance impact caused by the validation of a model at the end of an epoch, the validation was parallelized by using threads. Additionally, the optimizer used was optimized, as it was deemed a bottleneck in some executions.

The analysis of the execution flow as also led to a number of considerations regarding future developments and the most efficient use of the framework.

Contents

Contents **iv**

List of Figures **v**

1 Introduction **1**

2 Profiling **2**
2.1 Data input files 2

3 Features and Improvements **3**
3.1 Single Process Mode 3
3.2 Validation Thread 3
3.2.1 Distributed Validation 3
3.3 Adam optimizer 4

4 Conclusions **6**
4.1 Acknowledgments 6

Bibliography **7**

List of Figures

3.1 Average execution time with and without the use of the validation thread. Using 5 worker in the GRU example. 4
3.2 Average time taken by each implementation of the ”apply update” step. 5

1. Introduction

Neural Networks (NNs) have been revolutionizing many fields in technology and science for the past years, and High Energy Physics is no exception. NNs can be faster and more accurate than other traditional methods. However, training NNs can be a very slow process, requiring hours days or weeks to achieve results.

It is highly desirable to speed up the training process, so that users could iterate on their models faster, and achieve results in a timely manner. Using specialized hardware, such as GPUs, that parallelize the training in a single machine, is fundamental to speedup the computation. However, even with high end hardware, the full process may take a long time. In order to further reduce the waiting time, the computation can be distributed, so that several machines, and several GPUs collaborate to make training faster. There are several approaches to parallelize training.

At CERN, machine learning models are being widely used by physicists, with knowledge in machine learning. As such, it is important to have a way to speedup computation that does not require deep knowledge of parallelism or distributed computing techniques. Most of the users are already familiar with popular packages used for machine learning, such as Keras or Pytorch, and have models implemented with these tools. In order to reduce the time spent with boilerplate code, it is important to be able to use this existing knowledge and code with parallelized approaches, avoiding as much extra work as possible. It is also important to explore as many available machines as possible, without the requirement of a high speed network, or hardware in the same physical machine.

In response to this need, *MPI Learn*[5] was developed, a framework that speeds up the training, while abstracting the distribution of computation from the user. This framework aims at being easy to use while providing enough flexibility. Users need to provide the model to train and tune any relevant parameters. The distributed training happens without intervention or any need for custom code. Provided models can be exported from popular platforms such as TensorFlow, Keras or Pytorch. The code of the framework is highly modular, and therefore can be customized to different needs.

Internally, MPI Learn follows a Master-Slave architecture, where a main model is kept on the master process, and each worker is responsible for training on a sub-set of the training set. Each worker will contribute updates to the master, which will update the main model accordingly, and give the updated model to the workers.

The framework is currently implemented, and being used for research. The current focus of the project is in maintenance, adding new features as needed by the users, and improving the efficiency of the framework. The work on the performance of the platform focuses on profiling and analysis of its execution, in order to improve any inefficiencies detected.

2. Profiling

The main focus of the work developed over the months of July and August was the optimization of the MPI Learn implementation. This objective was achieved by profiling the execution of representative examples in MPI Learn, and implementing features and improvements according to that analysis.

In order to analyze the execution profile, the built-in tracing option was used, which provides a trace file readable with Google Chrome's ”Trace Event Profiling Tool”.

The data and models used had a big variation. The starting point was the standard MPI Learn examples based on the MNIST and CIFAR10 datasets. Models closer to realistic high-energy physics usecases were also tested. In particular, we used so-called jet-tagging classifiers based on GRU and (1D and 2D) convolutional layers.

Profiling was used to motivate the addition of features and improvements. Additionally, it was used to measure the effectiveness of the changes made. These will be discussed in further detail in Chapter 3. The execution profiles obtained were also useful to analyze how configuration options change the performance, such as the configuration of the input files.

2.1 Data input files

MPI Learn accepts a list of input files for training. These are distributed among the workers as evenly as possible. Individual files are not split among processes, no matter their content or if it is the only input file. Opening and processing a file involves some overhead. It is pertinent to ask if there is an ideal number of files in which to split the computation, in order to maintain performance.

Since the content of a file is not split, it is important to have a number of files divisible by the number of workers used, so that the work distribution is made evenly. Experiments were made using convolutional network example, using both a very large number of files, each containing a batch (100 examples), and the minimum number of files, where all the training data were split into 5 files. The global overhead does not change significantly when changing the number of files, favouring very slightly having as many files as worker processes.

It may be interesting to explore parallelism between file I/O and training. At the moment, in the worst examples examined the overhead represents less than 10% of the execution time.

3. Features and Improvements

Motivated by the results of the execution profiles of different examples, different features were developed, which will be laid out in the following sections. As part of the maintenance of MPI Learn, small code issues were fixed, which will not be presented for lack of practical interest.

3.1 Single Process Mode

In order to make an analysis of the scaling of MPI Learn, and understand the performance impact of using several processes in the training, it is useful to execute the program in a single process, where communication latency is not a factor to consider.

With this in mind, a single process execution mode was developed (Implemented and submitted in [2]). When executing in a single process, the same steps that would run in a distributed execution will be performed, save for the communication, which is substituted by updates to the relevant variables. In particular, the optimizer that runs in the master, applying the updates of each worker to the main model, is still used in this version.

3.2 Validation Thread

In some examples, the validation, at the end of each epoch, can take up a significant percentage of run-time. In particular, when executing the GRU example with 5 workers, validation would take 33% of the execution time. Validation is performed in the master and, in a regular execution, it will not do anything else at the same time, implying that all workers must wait until the validation is done in order to submit new updates to the master and continue their training.

In order to avoid wasting computing resources, the workers must be able to keep on training during the validation. With this objective in mind, the validation was moved to a dedicated thread in the master, while the main thread keeps on answering the workers and applying the updates being received. (Implemented and submitted in [4]) The main thread provides the validation thread with the values of the weights at the time of validation, via a queue. The validation thread then uses those weights and saves all results accordingly, staying idle the remainder of the time.

The implementation of the thread is impactful when the validation represent a significant part of the execution time. In particular, for an execution where the validation takes up 28% of the time (89s in a total of 313s), the use of the thread reduces the overall execution time by 23% (65s). This represents a good speedup, and reduces almost completely the impact of the validation on the performance of training.

3.2.1 Distributed Validation

Using a thread in the master to perform validation allows for a better use of the GPU in the master process, and allows workers to train continuously, without waiting for the validation. However, if the validation takes enough time, it will still become the bottleneck of the computation. If the validation thread is always working, the computation time will depend on it.

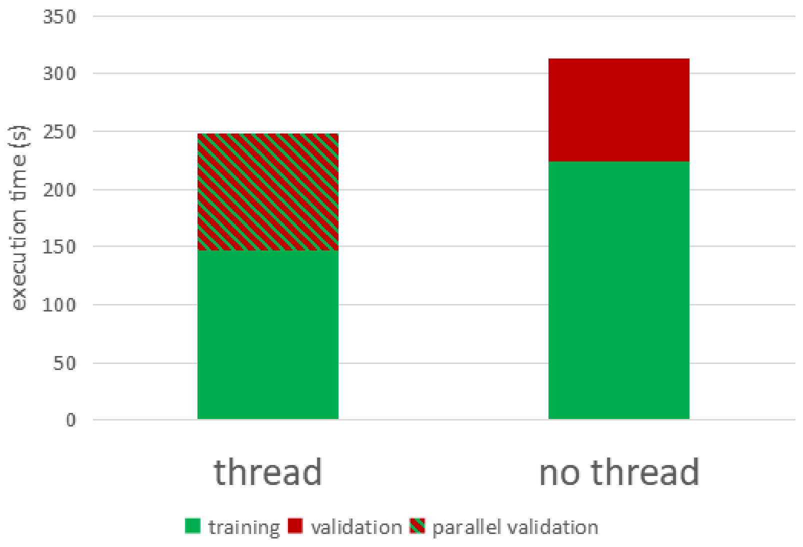

Figure 3.1: Average execution time with and without the use of the validation thread. Using 5 worker in the GRU example.

This can be solved by distributing the validation. Each worker can be attributed a fraction of the validation set, and perform the validation at the request of the master. This can be incorporated as falg sent when the worker receives weights from the master, to signal that validation should be performed, rather than training. Each worker would simply have to run its fraction of the validation and return the results to the master, which aggregates the results.

This problem has only been relevant in examples where the validation set is very big and the number of workers large. Since these were not deemed realistic at the moment, distributed validation has not been implemented.

3.3 Adam optimizer

The profiling done with the CIFAR10 example presented a poor scalability with a small number of workers. The master process spent most of the time applying the updates sent by the workers, while these spent most of the time waiting for a response from the master. In order to solve the issue, it was necessary to increase the performance of applying each update at the master.

The original code implemented the adam optimizer in *numpy*. The computation inside numpy itself was taking too much time to execute. The first attempt made was to use different numpy methods to solve the problem. The objective was to reduce the number of calls to numpy, since the function call itself and the use of intermediate variables introduces overhead. However, this approach did not yield any particular speedup.

NumExpr [1] is a numerical evaluator for python. It is a library that receives a string representing the expression to compute, which is subsequently processed and executed. This library can be faster than numpy, since it avoids storing intermediate results. Aside from that, it would also use all available CPUs, automatically parallelizing the computation.

Using this library instead of numpy to implement the computation proved to be much faster, reducing the execution time of each update from $20msec$ to $6msec$. This improvement reduced the computation time, but it is still a limitation to scalablity, since training in the workers is very fast.

In order to further improve the execution time, it is interesting to explore the parallelism provided by the GPU. As such, an implementation of the Adam optimizer was developed, using tensorflow. The final version wraps Tensorflow's own implementation of the Adam optimizer, making it compatible and modular in relation to the rest of MPI Learn. This implementation proved to be even faster, reducing the execution time of each update to around $3msec$. This was the final version implemented, and submitted to the main code base. [3]

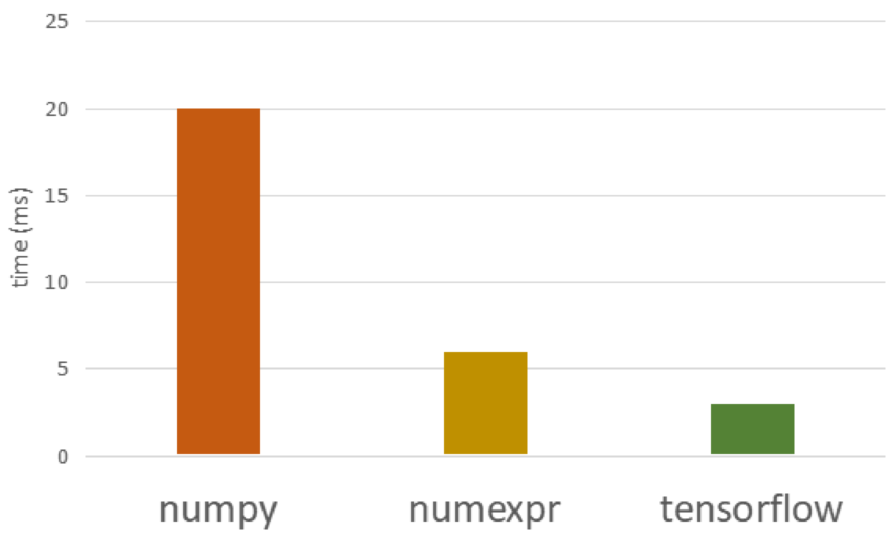

Figure 3.2: Average time taken by each implementation of the "apply update" step.

4. Conclusions

The work developed over the course of this CERN openlab experience focused on the profiling of MPI Learn, as well as addition of new features and efficiency improvements.

The profiling work resulted in a better understanding of the current performance challenges, and of how model dependent the execution time is. The features implemented resulted in faster execution times, making it viable to parallelize and speedup more use cases. In particular, using a thread in the validation removes one of the main synchronization points from the workers. The improvement of the incorporation of updates in the master, on the other hand, allows for the efficient parallelization of models that take a small time to train each batch.

As MPI Learn is implemented and in active use, it will remain in continuous improvement, with features being added as users require. It will be important to improve the experience based on user feedback regarding supported features and user interaction with the framework. Furthermore, the optimization work should continue, in order to ensure that the computational resources are used as much as possible.

In particular, if the use of heterogeneous machines is considered, or the use of machines where the resources may be in concurrent use by other processes, it makes sense to implement load balancing in the training. At the moment, each worker received an equal part of the training data, and performs its training over it. However, should a worker be slower to process its share, that will not be compensated by other workers, which will stay idle at the end of the computation, until the slower worker finishes. Training data could be dynamically distributed by the master, without significant overhead, as long as a queue of input files to use is kept in the master.

Additionally, it is important to explore GPU parallelism, aside from multi-process parallelism. Increasing the batch size to maximum supported gives a free speedup by better exploring GPU parallelism. Any downsides of increasing the batch size should also be visible when adding workers. It should be part of the instructions of use of MPI learn to keep the batch size per process as high as possible. It would be also interesting if the framework could automatically selects a suitable batch size to better explore the available hardware.

As the computational effort is better distributed and time performance increases, it is important to maintain the accuracy of the training performed in a distributed fashion. To this end, algorithms that control the instability associated with distributing the training will be explored, such as GEM[6].

4.1 Acknowledgments

This work was mainly conducted at ”*iBanks*”, the AI GPU cluster at Caltech. We acknowledge NVIDIA, SuperMicro and the Kavli Foundation for their support of ”*iBanks*”.

This project was received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement no 772369).

Bibliography

[1] Numexpr library. `https://github.com/pydata/numexpr`. Accessed: 2018-08-28. 4

[2] Single process mode pull request. `https://github.com/svalleco/mpi_learn/pull/22`. Accessed: 2018-08-23. 3

[3] Tensorflow adam optimizer pull request. `https://github.com/svalleco/mpi_learn/pull/27`. Accessed: 2018-08-28. 5

[4] Threaded validation pull request. `https://github.com/svalleco/mpi_learn/pull/23`. Accessed: 2018-08-23. 3

[5] Dustin Anderson, Jean-Roch Vlimant, and Maria Spiropulu. An mpi-based python framework for distributed training with keras. *CoRR*, abs/1712.05878, 2017. 1

[6] Joeri Hermans and Gilles Louppe. Gradient energy matching for distributed asynchronous gradient descent. *CoRR*, abs/1805.08469, 2018. 6