Distributed Pytorch Performance Analysis

**Shraddha Naik**

Manipal Institute of Technology

Manipal, India.

**Harshit Yadav**

Manipal Institute of Technology

Manipal, India.

***Abstract: Deep* Learning and Data Mining (DLDM) algorithms are becoming increasingly important in analyzing large volume of data generated by simulations, experiments and mobile devices. With increasing data volume, distributed memory systems (such as tightly connected supercomputers or cloud computing systems) are becoming important in designing in-memory and massively parallel DLDM algorithms. Yet, the majority of open source DLDM software is limited to sequential execution with a few supporting multi-core/many core execution. In this paper, we extend recently proposed Deep Learning framework by Facebook Pytorch for execution on large scale clusters using Message Passing Interface (MPI). We evaluate our implementation using MNIST** **datasets by sequential and parallel execution using MPI. Our evaluation indicates the efficiency of our proposed implementation.**

Keywords— *Pytorch-Facebook, Message Passing Interface, Distributed Computing, Sequential Execution, Deep Learning.*

# Introduction

Today, simulations, experiments and mobile devices are

generating increasingly large volume of data. Machine Learning and Data Mining (MLDM) algorithms, which can build models, classifiers, and anomaly detectors are being designed and applied in several domains including high energy physics, computational biology, and cyber security. MLDM algorithms are generally classified as supervised (the input dataset is labeled with ground truth) and un-supervised (learning from un-labeled dataset).Base unsupervised /supervised algorithms can be combined together using ensemble methods to remove noise, and possibly learn better models/classifiers.

Several software packages which support supervised, unsupervised and ensemble algorithms have been released publicly. A few well known packages are Weka, Scikit, libsvm and Matlab. However, these packages only support sequential execution. As a result, they are generally used with modest size datasets. At the same time, Deep Learning algorithms - a class of MLDM algorithms are becoming increasingly popular.[1] Deep Learning algorithms emulate brain activity by using several layers of neurons (interconnected with synapses) and learn the weights for the synapses by using gradient descent methods. There are several classes of Deep Learning algorithms Deep Neural Networks-(DNN-typically used on tabular datasets), Convolutional Neural Networks (CNNs - typically used on images) and Recurrent Neural Networks (RNNs - typically used on time-dependent datasets).[2] Several researchers/practitioners have applied Deep Learning algorithms to their problems, and reported better results in comparison to their well published models. Naturally, open source efforts such as Theano, CuDNN, and Caffe have gained traction and wide acceptance among researchers and practitioners alike.

PyTorch is an [open source](https://en.wikipedia.org/wiki/Open_source" \o "Open source) [machine learning](https://en.wikipedia.org/wiki/Machine_learning" \o "Machine learning) [library](https://en.wikipedia.org/wiki/Library_(computing)" \o "Library (computing)) based on the [Torch](https://en.wikipedia.org/wiki/Torch_(machine_learning)" \o "Torch (machine learning)) library, used for applications such as [computer vision](https://en.wikipedia.org/wiki/Computer_vision" \o "Computer vision) and processing. It is primarily developed by [Facebook](https://en.wikipedia.org/wiki/Facebook" \o "Facebook)'s artificial intelligence research group. It is [free and open-source software](https://en.wikipedia.org/wiki/Free_and_open-source_software" \o "Free and open-source software) released under the [Modified BSD license](https://en.wikipedia.org/wiki/Modified_BSD_License" \o "Modified BSD License). Although the [Python](https://en.wikipedia.org/wiki/Python_(programming_language)" \o "Python (programming language)) interface is more polished and the primary focus of development, PyTorch also has a C++ frontend. PyTorch provides two high-level features - Tensor computing (like [NumPy](https://en.wikipedia.org/wiki/NumPy" \o "NumPy)) with strong acceleration via [graphics processing units](https://en.wikipedia.org/wiki/Graphics_processing_unit" \o "Graphics processing unit) (GPU) and [Deep neural networks](https://en.wikipedia.org/wiki/Deep_neural_networks" \o "Deep neural networks) built on a tape-based autodiff system

# Literature review

While doing the literature review we came across several researchers who have conducted in-depth exploration of MLDM algorithms, with a few focusing on scalability to multi-core/many-core systems. A few researchers have considered execution on large scale systems. Recently, several toolkits have become popular for MLDM algorithms. Several MLDM toolkits which support sequential execution such as Weka, Matlab, Scikit, Orange and libsvm have been very widely used for data analysis.[4]

With recent developments in Deep Learning algorithms, several implementations of Deep Learning algorithms have become available for multi-core and many-core systems such as Theano, CuDNN and Caffee. A few other toolkits support execution on large scale systems. These toolkits include Microsoft DMTK and Machine Learning Toolkit for Extreme Scale (MaTEx), TensorFlow.[3]

Recently released Pytorch supports MLDM algorithms with automatic differentiation. It is readily available for deployment with multi-core and many-core clusters. It contains several optimizations such as Adaptive Gradient Descent (Ada-Grad), Dropout for regularization among others. The distributed package included in Pytorch (i.e. torch.distributed) enables researchers and practitioners to easily parallelize their computations across processes and clusters of machines.

# Methodology

## PYTORCH – DEEP LEARNING FRAMEWORK

Pytorch is an [open source](https://en.wikipedia.org/wiki/Open_source" \o "Open source) [machine learning](https://en.wikipedia.org/wiki/Machine_learning" \o "Machine learning) [library](https://en.wikipedia.org/wiki/Library_(computing)" \o "Library (computing)) based on the [Torch](https://en.wikipedia.org/wiki/Torch_(machine_learning)" \o "Torch (machine learning)) library, used for applications such as vision and [natural language processing](https://en.wikipedia.org/wiki/Natural_language_processing" \o "Natural language processing). It is primarily developed by [Facebook](https://en.wikipedia.org/wiki/Facebook" \o "Facebook)'s artificial intelligence research group .It is [free and open-source software](https://en.wikipedia.org/wiki/Free_and_open-source_software" \o "Free and open-source software) released under the [Modified BSD license](https://en.wikipedia.org/wiki/Modified_BSD_License" \o "Modified BSD License). The distributed package included in Pytorch (i.e. torch.distributed) enables researchers and practitioners to easily parallelize their computations across processes and clusters of machines. To do so, it leverages the messaging passing semantics allowing each process to communicate data to any of the other processes. As opposed to the multiprocessing (torch.multiprocessing) package, processes can use different communication backend and are not restricted to being executed on the same machine.

The Message Passing Interface (MPI) is a standardized tool from the field of high-performance computing. It allows to do point-to-point and collective communications and was the main inspiration for the API of torch.distributed. Several implementations of MPI exist (e.g. [Open-MPI](https://www.open-mpi.org/), [MVAPICH2](http://mvapich.cse.ohio-state.edu/), [Intel MPI](https://software.intel.com/en-us/intel-mpi-library)) each optimized for different purposes. The advantage of using the MPI backend lies in MPI’s wide availability - and high-level of optimization - on large computer clusters.[1] [Some](https://developer.nvidia.com/mvapich) [recent](https://developer.nvidia.com/ibm-spectrum-mpi) [implementations](https://www.open-mpi.org/) are also able to take advantage of CUDA IPC and GPU Direct technologies in order to avoid memory copies through the CPU. Unfortunately, PyTorch’s binaries cannot include an MPI implementation and we’ll have to recompile it by hand. Fortunately, this process is fairly simple given that upon compilation, PyTorch will look by itself for an available MPI implementation. The following steps install the MPI backend, by installing PyTorch [from source](https://github.com/pytorch/pytorch" \l "from-source).

## MESSAGE PASSING INTERFACE(MPI)

Message Passing Interface (MPI) provides a rich set of abstractions for inter-process communication. MPI supports pair-wise communication (such as using send, receive) and group communication (such as using reduction, barrier). MPI has become the de facto communication interface for legacy scientific applications. The primary reason for MPI's success is its wide availability.[5] MPI is available on large scale supercomputers, cloud computing systems and it can also be used for interprocess-communication on a single compute node if other shared memory programming models are not available. Unlike other runtimes such as Spark and GRPC,. Due to the performance reasons, we considered MPI to be the primary communication interface instead of other communication subsystems. Specifically, we have used several MPI routines for our large scale implementation. We have used All-to-all reduction (an MPI primitive which allows operations such as sum on user's data, and disseminates the final result among all the processes in a group) for averaging weights and biases and point-to-point operations for data distribution. We also observed that MPI has been criticized for its lack of support for fault tolerance. However, with recent advancements such as User-level Fault Mitigation (ULFM) and open source implementations, it is possible to design fault tolerant MLDM algorithms using MPI, without losing performance and "continued execution" in the presence of hardware faults. We expect that with ULFM (or its variants) becoming available with mainstream implementations, MPI would find its wide acceptance in the MLDM community.

## Proposed Model(PYTORCH USING MPI)

To implement our proposed approach, we used MNIST dataset which is widely used in Machine Learning. The dataset was passed through a Convolution Neural network in batches of varying sizes. We used Python as our programming language to code. We imported Pytorch, mpi4py libraries to carry out our execution. The basic unit in Pytorch is the computational graph. This graph contains nodes, which are operations, and edges which represent tensors (arbitrary dimensional arrays). Each node can take multiple inputs and give multiple outputs, with tensors created and passed from one node to another

and generically, deleted after use to avoid memory clutter. In addition to carrying tensors, edges can also be used to control the flow of a computation. Control dependencies can be used to enforce relationships such that some computations must be done before others, no matter what parallelization has occurred.

Parallelization in Pytorch is done in a task-based manner. That is, each node is assigned to a device for computation, rather than running the whole graph, in parallel, on multiple devices. The way that it assigned is via a greedy algorithm. First, Pytorch runs a simulation of the graph to determine approximately how long each node will take to compute and to determine the computation order as above. Then, the greedy algorithm assigns nodes to devices based on whether or not there is a kernel for that operation on that device (not all operations have GPU implementations, for instance) and based on which device is expected to be free when the computation is ready to be done. Finally, Pytorch inserts send and receive nodes between

devices to transfer the tensors. It does this in a way to minimize communication (given the assignment of the graph) and modifies the graph assignments slightly if it changes the total execution time to change where communication happens.

We considered several methods for parallelism. Firstly, we considered the methods where the matrices belonging to each layer (neurons connected with synapses) were distributed among multiple compute nodes, possibly with block/row decomposition. However, this approach requires significant communication for each sample and hence other approaches were considered.

We considered an approach where the model is replicated on each device. Each device learns the model independently using standard backpropagation algorithm. This approach scales well in computation

and communication, even though the model is replicated on each device. To support this argument, let us consider a simple performance model of computation and communication at each epoch during the training process. Let m be the number of samples, and p be the number of processes. For simplicity, let n be the number of neurons in each layer and l be the number of layers. Hence, at each

epoch, the total number of FLOPs (floating point operations) is m/p.

. l, while the total communication volume is . l, Naturally, with strong scaling - the work per device reduces - however for reasonable work distribution, the overall time in communication can be managed. By using MPI and high performance communications, the overall fraction of time spent during computation is increased. Hence, we implement this form of parallelism for our implementation.

# EXPERIMENTAL

The MNIST database of handwritten numbers is a widely used data set in Machine Learning. We consider MNIST dataset for our evaluation on sequential and parallel execution.

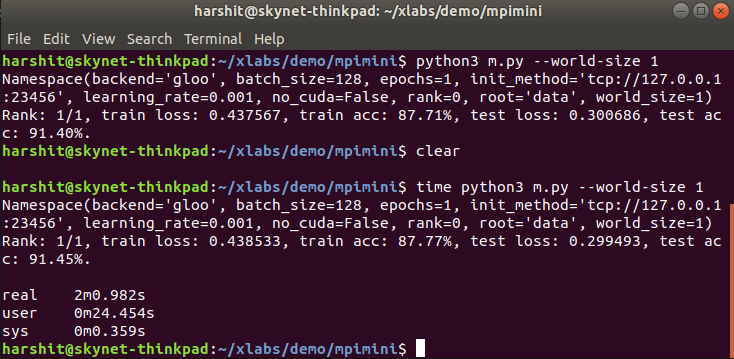


Figure 1: Sequential Execution

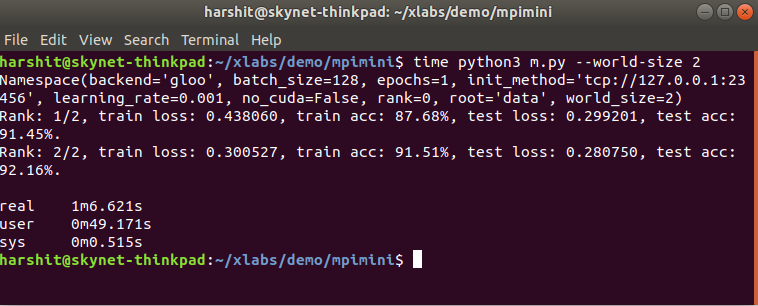


Figure 2: Parallel Execution with 2 Cores

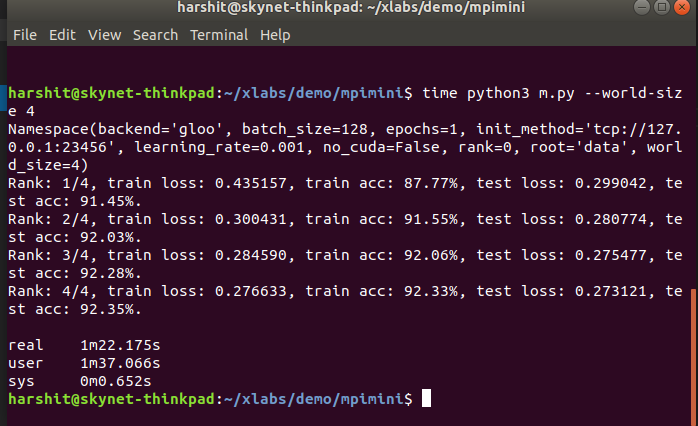


Figure 3: Parallel Execution with 4 Cores

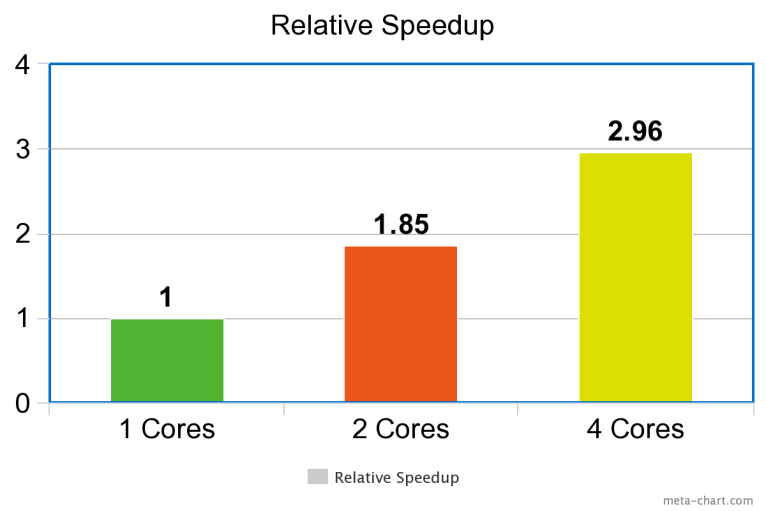


Figure 4: Relative speedup of 1 core upto 4 cores

# Results and analysis

Table 1 shows the comparison between execution times with 1 core, 2 cores and 4 cores. It is observed that we get the best results on MNIST database with parallel execution with 4 cores.

|  |  |
| --- | --- |
| Model | Execution Time |
| Sequential Execution | 4m0.616 |
| Parallel Execution with 2 cores | 2m10.635 |
| Parallel Execution with 4 cores | 1m21.486 |

# Conclusion

In this paper, we have proposed a design to alleviate the distributed memory limitations of Pytorch. We observed the behavior of Pytorch on the dataset using various cores. Evaluating the MNIST dataset gives the best speedup when executed parallely on 4 core using Pytorch with MPI. We verified the speedup i.e. sequential execution with parallel execution using 4 cores and we got 98% efficiency.

These results provide encouragement to develop Pytorch using MPI for evaluating handwritten Digit Recognition. Further research can be carried out to implement Pytorch using CUDA with large datasets and analyze its efficiency.

REFERENCES

[1] Abhinav Vishnu, Charles Siegel Jeff Daily, Distributed TensorFlow with MPI, Macrh 7, 2016.

[2] P. J. Sadowski, D. Whiteson, and P. Baldi, \Searching for higgs boson decay modes with deep learning," in Advances in Neural Information Processing Systems 27, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds. Curran Associates, Inc., 2014.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, \Ca\_e: Convolutional architecture for fast feature embedding," arXiv preprint arXiv:1408.5093, 2014.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel,B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and

E. Duchesnay, \Scikit-learn: Machine learning in python," J. Mach. Learn. Res., vol. 12, Nov. 2011.

[5] [https://pytorch.org](https://pytorch.org/), <https://en.wikipedia.org/wiki/PyTorch>