

# Distributed Pytorch Performance Analysis using MPI

**Shraddha Naik**

Manipal Institute of Technology  
Manipal, India.

**Harshit Yadav**

Manipal Institute of Technology  
Manipal, India.

***Abstract: Deep Learning and Machine Learning algorithms have become important recently in processing huge amounts of data to train the network. With Increase in Cloud Computing and Distributed Systems in the Data Centers which are tightly connected ,massive supercomputers. Designing an algorithm that can completely utilize the available multicore processing infrastructure is crucially important to reduce the processing time , There is a wide variety of Open Source Frameworks and Libraries but many of them are built as Wrappers using the classic C, C++ libraries which are great for Solving the requirement but they are mostly sequential and does not support new distributed system designs . There are popular frameworks like PyTorch Developed by facebook which are open source and give the User flexibility to Implement it using Multiple Backends like Message Passing Interface (MPI) over default sequential configuration in the Package. We Will evaluate the MNIST datasets using Pytorch Compiled with MPI comparing the performance for both Sequential and Parallel Execution to compare their efficiency.***

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

## I. INTRODUCTION

In Today's world data is being generated at a massive scale of Petabytes each day through various Mobile Devices , Sensors , IoT Devices , Simulations etc . Machine Learning and Deep Learning algorithms are increasingly used to build models which can be used to gain new Insight from the Data and detect anomalies in a wide Range of Domains from field of Physics to Medical Science . Machine Learning and Deep Learning Algorithms are widely used to train on this dataset which can be labelled with ground truth (called the Supervised Machine Learning ) or unlabeled (called as the Unsupervised Machine Learning ) . Both these Supervised and Unsupervised techniques can be

combined together to improve model training accuracy reduce the Noise and train using better algorithm and weights.

There are many software packages which support the ensemble algorithms for Supervised and Unsupervised training.

Deep Learning Algorithms are becoming increasingly popular[1], as given sufficient time and data to process any deep learning model can beat the accuracy and efficiency of Machine Learning Model but with the downside that is extremely computation Intensive Few of these packages are Scikit, Matlab and Numpy , but these packages are built as wrappers around the native C, C++ mathematical library and functions which supports only sequential execution but the Deep Learning frameworks have to deal with large datasets over multiple iterations called epochs to have higher model accuracy .

Deep Learning Algorithms are Modeled to emulate the working of a brain using several layers of Neurons (acting as interconnected Synapses ) and weights for defining the triggering threshold for these synapses called Gradient Descent Method.

There are Several classifications of Deep learning Algorithms such as RNN (recursive neural network which are used on datasets which are time-dependent and are in chronological order)[2], DNN (Deep Neural Networks used to deal with Tabular datasets),CNN (Convolution Neural Network used majorly with Images ).Researchers and Machine Learning Scientists apply these various algorithms to their problem statement and and compare the model accuracy and training time of each method applied for comparison . Usually open Source softwares and frameworks like Theano and Caffe are oriented towards their mathematical capabilities get favoured by the Researchers which are great when building a models but are inefficient for production and Industry Usage

PyTorch is an Machine Learning Library developed by Artificial Intelligence Research Division of Facebook Inc. based on the Torch library for the purpose of Image Processing to handle the Social media Giant need for processing large amount of Social Media feeds in form of user uploaded Images . It is released under the modified BSD licence and free to use . Open source programming language Python Interface is widely used

and accepted standard of Pytorch, It also has C++ frontend which can be tweaked for high level of customization for performance gain and hardware support.

Pytorch provides many useful inbuilt functions and libraries but two of its high level Feature are- Strong support for GPU (Dedicated graphical processing hardware) as DNN are built on tape based automatic differentiation system , Tensor like Computing Similar to Numpy.

## II. LITERATURE REVIEW

While doing the Literature review we came across multiple researches and performance analysis conducted in detail about the performance and efficiency comparisons of various Machine learning algorithms on various sets of data and different types of Hardware with focus on Scalability to Multi Core system execution .

Few Research done by the Industry tech giants such as Facebook , Google , Microsoft considered large scale production level conditions with Datacenter level hardware specification.

Recently Various Machine Learning and Deep Learning toolkits have gained traction and support from enthusiasts for data analysis purpose which use sequential execution like Weka and Matlab , the most popular among the top contenders are Tensorflow by Google and Pytorch by Facebook .

Development and Introduction of Chipsets specifically designed for the Machine Learning algorithms the many of the popular libraries and toolkits have been made compatible to Support Multi Core Execution , Such as Theano and Caffe . Few of the Systems have been designed keeping in consideration Large Scale Systems which Include the Microsoft DMTK (Distributed Machine Learning Toolkit) , MaTex (Machine Learning toolkit for Extreme Scale)and Google Tensorflow[3] which has been optimized and designed to gain huge performance boost on the Custom built Proprietary Hardware called TPU (Tensor Processing Unit).

Pytorch Latest Release supports automatic differentiation, Adaptive Gradient Descent , with easy deployment option across many Multi Core Clusters which allow easy parallelization of computation process across multiple machines in a Distributed environment.

## III. METHODOLOGY

### A. PYTORCH – DEEP LEARNING FRAMEWORK

Facebook Developed Pytorch to answer their need of a Machine Learning Algorithm to analyse the Visual Data generated by the Users on their Social Media Platform ,built using the Machine Learning Library Torch which is based on the programming language LUA which is highly efficient in computing machine learning related tasks but has a high learning curve and does not support many of the Modern features provided by the programming language such as Python Pytorch Library is Completely free and Open source with all its Code available on Github under the modified BSD license. The Package (pytorch.distributed) allows us to easily parallelize the computational process across multiple machines in a cluster setup. It supports this feature by using message passing API and techniques allowing the various interprocess communication . Another package with focus on multiprogramming (torch.multiprocessing ) which allows communication among multiple physically separated machines in a Cluster or Horizontal Scaled setup using various customizable and user user defined communication backends like GLO (Using Network Model Design ), MPI etc.

MPI is a standardised tool in the field of Parallel Computing which provides a set of API to do point to point and collective communication which was the main Inspiration for the Pytorch Library (torch.distributed) . Several Implementation of MPI Exists such as Open-MPI ,Intel MPI and MVAPICH2 for intel based CPU architecture each optimised for different use cases. Some recent implementations also take advantage of the Graphical Processing Unit technologies to avoid the copies through the memory. Unfortunately , Pytorch framework can not be included with MPI implementation because of various proprietary and patented technologies being used in both of them individually owned by separate entities . Therefore Pytorch binaries have to be manually recompiled which is fairly simple and time consuming since the compiling part is done sequentially . On execution the PyTorch the will look by itself for the available MPI backend implementation in the Binaries . The following steps recompile the MPI backend with Pytorch form the Source

### B. MESSAGE PASSING INTERFACE(MPI)

Message Passing Interface (MPI) was introduced in 1992 as an answer to writing standard parallel applications for a variety of major parallel architecture. Keeping in consideration that MPI is just and Interface for writing the methods for writing the message passing applications making it successful and widely used[5].

It enables use of various methods of abstractions for inter-process communication such as point to point communication (send, receive method) and group communication (reduce, scan method). MPI can also be used on a variety of hardware from a large scale

supercomputer to a single compute node for interprocess-communication in a desktop setup.

Unlike Other backends like GLOO and NCCL we used MPI for communication interface due to its performance reasons and easy compatibility with the Intel CPU architecture .

We also observed that MPI has been avoided due to its difficult debugging when implemented on a large scale and no techniques and functionality for fault tolerance . However recent advancements in User Level fault Mitigation (ULFM) and open source implementation has made it possible for fault tolerant algorithms without compromising on performance.

### C. Proposed Model(PYTORCH USING MPI)

To practically implement our proposed approach, we used MNIST dataset which is a standard dataset used in Machine Learning. The dataset was then passed through a CNN network in batches of varying sizes. Using Python as programming language to code. The basic unit in Pytorch is the computational graph. This graph contains nodes, which are operations, and edges which represent tensors (Multiple 2d arrays arranged one after another like a Rubik cube arrangement ). Each node can take multiple inputs and multiple outputs, with tensors created and passed from one node to another which are deleted after use to free up the memory as part of a memory management technique

In addition to carrying tensors, Control dependencies and flow of computation using the edges can be used to enforce relationships such that some computations must be done before others, no matter what parallelization has occurred.

Parallel computation using Pytorch is done on basis of task in which each processing is assigned to a processing element for computation, rather than running the whole graph, in parallel, on multiple devices using the greedy algorithms . in the beginning , Pytorch executes a check to estimate the size of the graph to determine approximately how long each node will take to compute the size of each chunk of data that needs to be assigned to each processing element for the computation and assign each chunk to the node using the greedy algorithm approach based on which device and processing element is free for next task execution . Finally, Pytorch inserts the Point to point MPI communication API like send and receive between devices to transfer the tensors. It does this in a way to minimize communication and modifies the graph assignment accordingly if it changes the total execution time more than the estimated ones

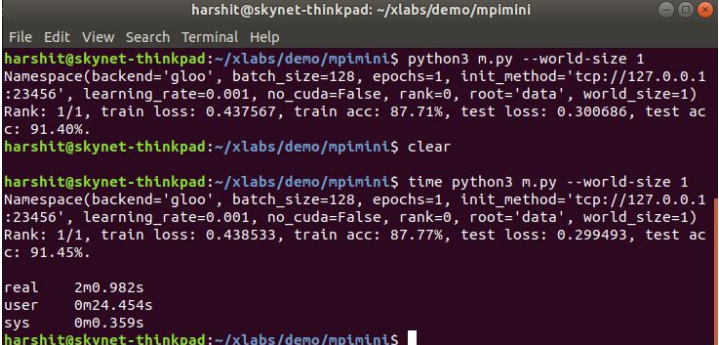
We followed an approach where the machine learning model is replicated on each processing element . Each device learns the model independently using standard backpropagation algorithm. This approach scales well in computation

and communication, even though the model is replicated on each device. At each iteration of the model , the total number of FLOPs (floating point operations) is  $m/p$  where  $m$  is the number of samples and  $p$  is the number of processes defined as an argument in parameters during the time of the execution.

The system was tested on physical 4 core CPU Device due to hardware limitations for the scope of this research . 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.

## IV. EXPERIMENTAL

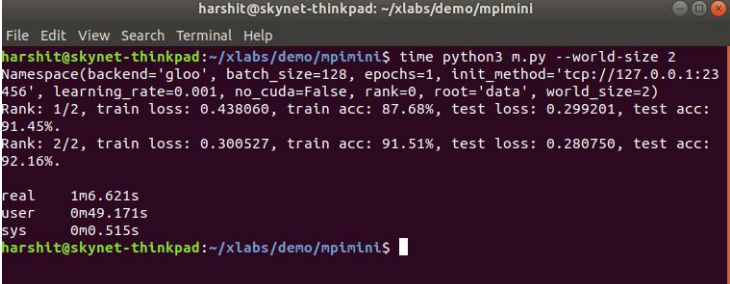
The MNIST database of handwritten numbers from range (0-9) with a collection of 60,000 images each of size 28X28 pixels is widely as a standard benchmark test in Machine Learning. For our evaluation on sequential and parallel execution performance of the compiled binaries we considered the MNIST data for measurement .



```
harshit@skynet-thinkpad: ~/xlabs/demo/mpimini
File Edit View Search Terminal Help
harshit@skynet-thinkpad:~/xlabs/demo/mpimini$ python3 m.py --world-size 1
Namespace(backend='gloo', batch_size=128, epochs=1, init_method='tcp://127.0.0.1:23456', learning_rate=0.001, no_cuda=False, rank=0, root='data', world_size=1)
Rank: 1/1, train loss: 0.437567, train acc: 87.71%, test loss: 0.300686, test acc: 91.40%.
harshit@skynet-thinkpad:~/xlabs/demo/mpimini$ clear
harshit@skynet-thinkpad:~/xlabs/demo/mpimini$ time python3 m.py --world-size 1
Namespace(backend='gloo', batch_size=128, epochs=1, init_method='tcp://127.0.0.1:23456', learning_rate=0.001, no_cuda=False, rank=0, root='data', world_size=1)
Rank: 1/1, train loss: 0.438533, train acc: 87.77%, test loss: 0.299493, test acc: 91.45%.

real    2m0.982s
user    0m24.454s
sys     0m0.359s
harshit@skynet-thinkpad:~/xlabs/demo/mpimini$
```

Figure 1: Sequential Execution



```
harshit@skynet-thinkpad: ~/xlabs/demo/mpimini
File Edit View Search Terminal Help
harshit@skynet-thinkpad:~/xlabs/demo/mpimini$ time python3 m.py --world-size 2
Namespace(backend='gloo', batch_size=128, epochs=1, init_method='tcp://127.0.0.1:23456', learning_rate=0.001, no_cuda=False, rank=0, root='data', world_size=2)
Rank: 1/2, train loss: 0.438060, train acc: 87.68%, test loss: 0.299201, test acc: 91.45%.
Rank: 2/2, train loss: 0.300527, train acc: 91.51%, test loss: 0.280750, test acc: 92.16%.

real    1m6.621s
user    0m49.171s
sys     0m0.515s
harshit@skynet-thinkpad:~/xlabs/demo/mpimini$
```

Figure 2: Parallel Execution with 2 Cores

```

harshit@skynet-thinkpad: ~/xlabs/demo/mpimini
File Edit View Search Terminal Help

harshit@skynet-thinkpad:~/xlabs/demo/mpimini$ time python3 m.py --world-s
e 4
Namespace(backend='gloo', batch_size=128, epochs=1, init_method='tcp://12
0.0.1:23456', learning_rate=0.001, no_cuda=False, rank=0, root='data', wo
rld_size=4)
Rank: 1/4, train loss: 0.435157, train acc: 87.77%, test loss: 0.299042,
st acc: 91.45%.
Rank: 2/4, train loss: 0.300431, train acc: 91.55%, test loss: 0.280774,
st acc: 92.03%.
Rank: 3/4, train loss: 0.284590, train acc: 92.06%, test loss: 0.275477,
st acc: 92.28%.
Rank: 4/4, train loss: 0.276633, train acc: 92.33%, test loss: 0.273121,
st acc: 92.35%.

real    1m22.175s
user    1m37.066s
sys     0m0.652s
harshit@skynet-thinkpad:~/xlabs/demo/mpimini$

```

Figure 3: Parallel Execution with 4 Cores

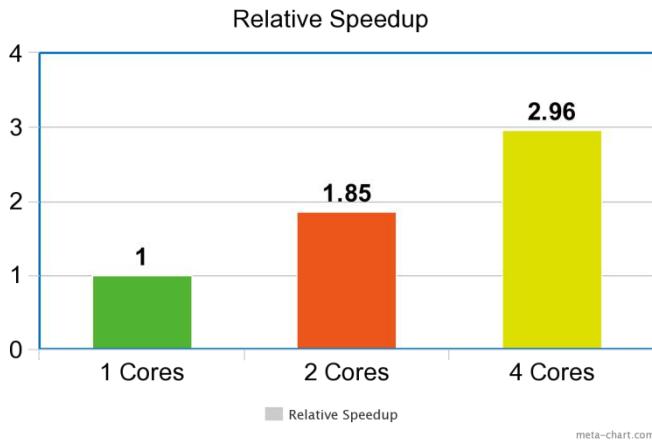


Figure 4: Relative speedup of 1 core upto 4 cores

## V. 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.

|                                | Execution Time |
|--------------------------------|----------------|
| Sequential Execution           | 4m0.616        |
| Parallel Execution with 2 core | 2m10.635       |
| Parallel Execution with 4 core | 1m 21.486      |

## VI. CONCLUSION

After implementing the Standard data set using our proposed specification improve the parallel memory management limitations of Pytorch. It was observed that the behavior of Pytorch on the dataset using various core count . Evaluating the MNIST dataset gives the best speedup when executed parallelly on 4 core using Pytorch with MPI. we checked the speedup i.e. sequential execution with parallel execution using 4 cores and we got 69% 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 NCCL and other standard datasets to 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>,

