APMA4302 Methods in Computational Science

Parallelizing Neural Network using MPI

Yaoxuan Liu (yl4578), Shengjie Lin(sl4830)

December 2021

Abstract

The goal of this project is to implement a neural network using parallel computing technique Message Passing Interface (MPI). A basic sequential neural network was coded in Python including one hidden layer and one output layer, and then parallelized by using data parallelism with the help of MPI for python (mpi4py). The improvement in performance with different number of processors has been evaluated and compared. The results showed that parallel computing could significantly speed up the training process. The scalability test was conducted as well.

1 Introduction

With the development of machine learning in recent years, various deep learning models have been generally used to conduct classification, recognition and prediction. Synchronously, more complicated models are proposed and larger datasets are trained, which leads to a higher cost in terms of training time and compute requirements. In order to speed up the learning process, the parallel computing technique MPI has been proved to be successful by applying to a number of applications for neural network parallel training, according to a survey (1). Our project investigates the basic parallel neural network model on the MNIST database by using MPI for python. Sequential and parallel neutral networks are developed and compared. Scalability analysis will also be conducted.

2 Neural Networks

Neural networks, also known as artificial neural networks (ANNs), are a subset of machine learning and the foundation of deep learning. Their name and structures are inspired by the human brain, mimicking the way that real neurons communicate with each other in biological systems (2). Neural networks consist of different node layers that include an input layer, one or more hidden layers, and an output layer. Each node links to another and has its own weight and threshold. If the output of any individual node exceeds the defined threshold value, that node will be activated and send data to the next layer of the network. Otherwise, no data will be send. Figure 1 demonstrates the structure of a simple two-layer neural network model.

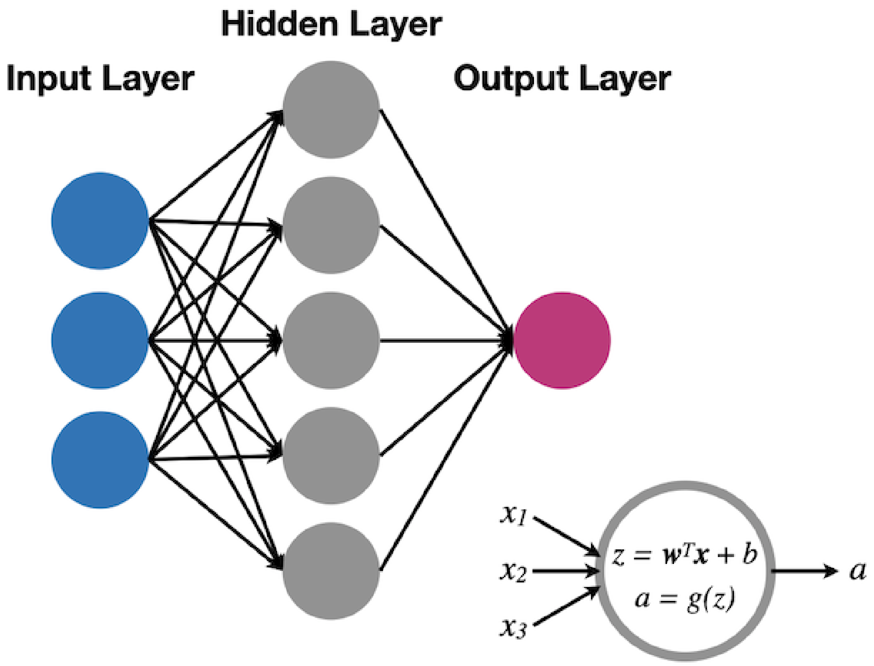


Figure 1: Two-layer Neural Network

The algorithms of neural networks mainly includes two parts: forward propagation and back propagation. Forward propagation sequentially calculates and stores intermediate variables within the computational graph defined by the neural network. Back propagation is a method for fine-tuning the weights of the network to reduce error rates and minimize loss.

$$\frac{dA}{dW} = \frac{dZ}{dW}\frac{dA}{dZ} \tag{1}$$

When training the neural network, the information enters into the input layer and forwards in the network to get the output value. Then, the output will be compared with the expected results where the cost function will be calculated. Back propagation computes the gradient of cost function with respect to all the weights in the network (3). As it shown in Eq.1, the back propagation algorithm is based on the chain rule where $\frac{dA}{dZ}$ is calculated and stored in the reversed order, while $\frac{dZ}{dW}$ is obtained by forward propagation. They are working interdependently to train the whole neural network.

$$w := w - \eta \nabla J(w) = w - \frac{\eta}{n}\sum_{i=1}^{n} \nabla J_i(w) \tag{2}$$

In addition, there are many different iterative methods and optimizer for updating the weights and optimizing the cost function. In this project, stochastic gradient descent (SGD) is adopted as Eq.2. The full algorithm of a N-layer neural network is derived as below:

```
Set $A^{[0]} = X$ (Input), $L=$ Total layers
Initialize $W^{[1]} \dots W^{[L]}$, $b^{[1]} \dots b^{[L]}$
for epoch=1 to max iteration

    Forward propagation
        for $l = 1$ to $L-1$
            $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
            $A^{[l]} = g(b^{[l]})$
            Save $A^{[l]}, W^{[l]}$ in memory for later use
        $Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$
        $A^{[L]} = \sigma(Z^{[L]})$

    Cost function $J = -\frac{1}{n}\left(Y \log\left(A^{[L]}\right) - (1-Y)\log\left(1-A^{[L]}\right)\right)$

    Back propagation
        $dA^{[L]} = -\frac{Y}{A^{[L]}} + \frac{1-Y}{1-A^{[L]}}$
        $dZ^{[L]} = dA^{[L]} \sigma'\left(dA^{[L]}\right)$
        $dW^{[L]} = dZ^{[L]} dA^{[L-1]}$
        $db^{[L]} = dZ^{[L]}$
        $dA^{[L-1]} = dZ^{[L]} W^{[L]}$
        for $l = L-1$ to $1$
            $dZ^{[l]} = dA^{[l]} g'\left(dA^{[l]}\right)$
            $dW^{[l]} = dZ^{[l]} dA^{[l-1]}$
            $db^{[l]} = dZ^{[l]}$
            $dA^{[l-1]} = dZ^{[l]} W^{[l]}$

    Update $W$ and $b$
        for $l = 1$ to $L$
            $W^{[l]} = W^{[l]} - \alpha \cdot dW^{[l]}$
            $b^{[l]} = b^{[l]} - \alpha \cdot db^{[l]}$
```

where W is weights matrix, b is bias vector, Z is affine transformations of given inputs, g is activation function, σ is the sigmoid function, A is post-activation output, dW is derivative of the cost function with respect to W, db is derivative of the cost function with respect to b, dZ is derivative of the cost function with respect to Z, dA is derivative of the cost function with respect to A, η is the learning rate.

3 Parallel methods

There are two most common ways to parallelize the neural network: model parallelism and data parallelism. Model parallelism uses the same data for every processor and then splits the model among processors, while data parallelism runs the same model for every processor, but feeds it with different group of sliced data. Figure 2 vividly presents the differences between model parallelism and data parallelism.

For neural networks, data parallelism means that we use the same weights and model but different mini-batches in each processors, and gradients need to be synchronized and averaged after the calculation on each mini-batch. Model parallelism, however, splits the weights of the network equally among the processor, and all processors will work on a single batch of data. The generated output of each part of the model needs to be synchronized and stacked as the input for the next part. Neural networks are computationally intensive because millions of parameters need to be updated numerous times in the training process. Those updates are nothing but large matrix multiplication operations. Therefore, parallelizing the network in either of two ways can significantly reduce the training time. In this project, we will use data parallelism.

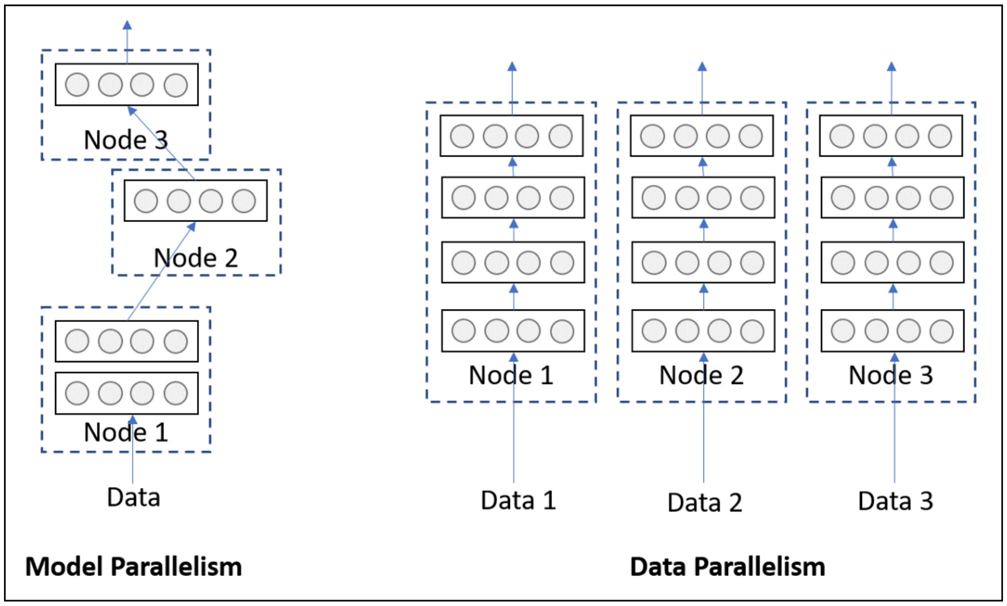


Figure 2: Model Parallelism and Data Parallelism

4 MPI

Message Passing Interface (MPI) is a standardized and portable message-passing standard for parallel computing systems. The MPI standard specifies the syntax and semantics of library routines that could be used by a wide variety of users who are creating portable message-passing applications in C, C++, or Fortran. MPI for Python (mpi4py) provides Python bindings for the MPI standard, allowing Python applications to exploit multiple processors on workstations, clusters and supercomputers (4).

In this project, some methods and operations in mpi4py communicator (mpi4py.MPI.Comm) are used to communicate the data from the address space of one process to that of another process. For example, `Bcast` will broadcast the same piece of data from one process to all other processes, and `Gather` will gather together values from a group of processes. `Scatter` is the operation of scattering chunks of an array to different processes. `Barrier` will barrier the synchronization at the specific point (5).

5 Dataset and Application

The dataset we applied in this project is from MNIST handwritten digit consisting of a training set of 5000 images. Each image is 20*20 grayscale presenting number 0-9 which will be classified into 10 labels.

Then, we started to parallelize neural network model and train the dataset by adopting data parallelism. First of all, rank 0 (master) will slice the data into different subsets and initialize the parameters. Then, these parameters will be broadcasted to other ranks (workers), and data subsets will be scattered as well. Next, the cost function and gradients will be calculated using sliced data in each rank and gathered in parameter server. The cost and gradients from each rank will be averaged and weights of network will be updated. Finally, new weights will be sent to the corresponding ranks. This process above will be repeated in every iteration. Our neural network model is running on Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz with 4 cores and maximum 8 logical processors.

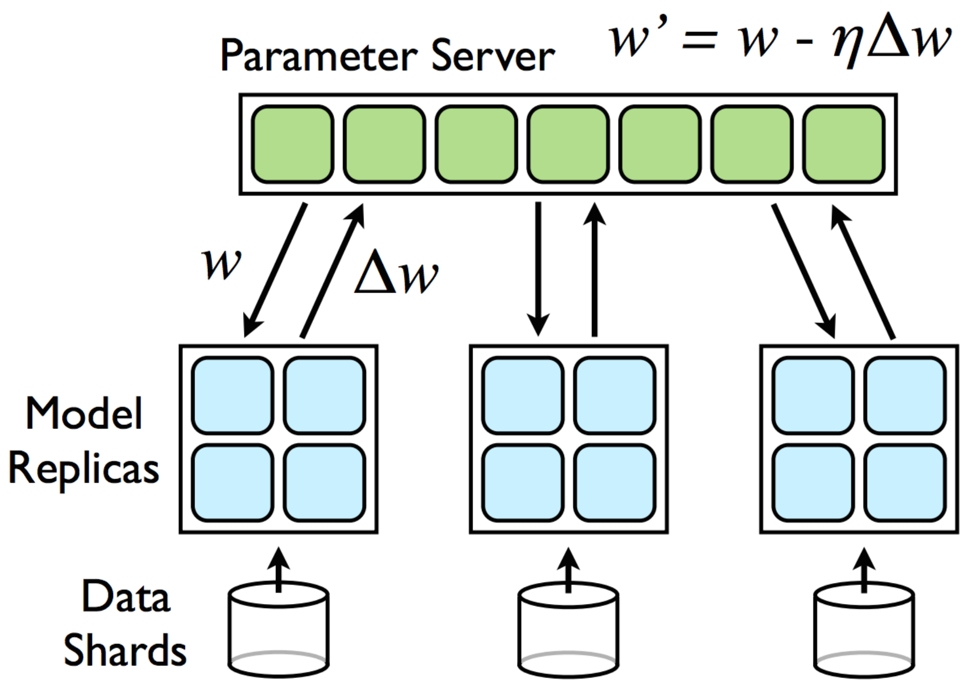


Figure 3: Data parallelism for updating parameters

6 Results and Discussion

To make the amount of data sufficient and meet the requirement of our study, we replicate the original data for multiple times. Figure 4 shows the execution time of the training process (10 iterations) including sequential and parallel neural network models. With the increasing number of processors adopted, the overall running time of our model has been significantly decreased, which means that our parallelization is successful.

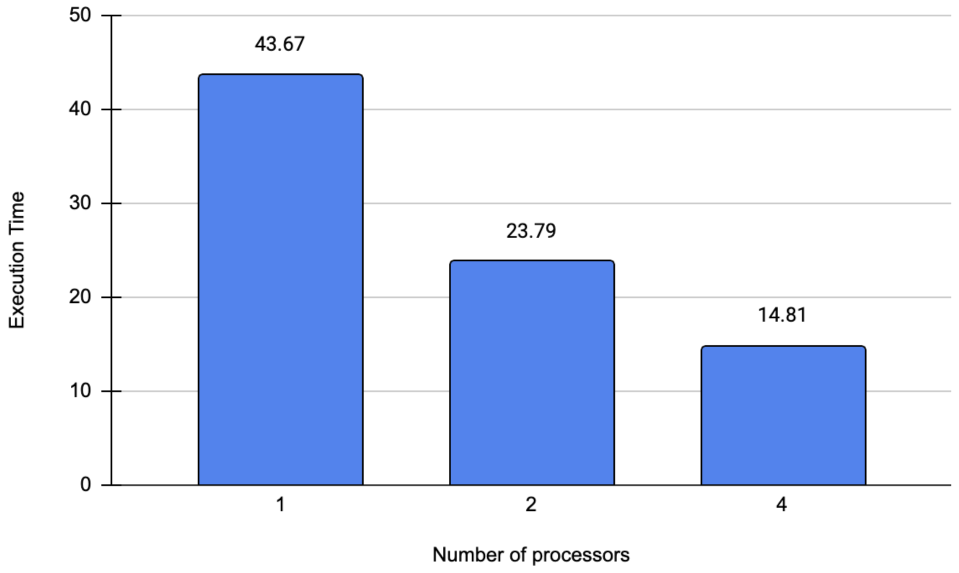


Figure 4: Comparison of parallel computing performance

In addition, scalability needs to be tested which is widely used to indicate the ability of hardware and software to deliver greater computational power when the amount of resources is increased. For software, scalability is usually referred to parallelization efficiency which is the ratio between real speedup obtained and ideal speedup when specific number of processors used. Ideally, software would have a linear speedup with respect to number of processors, which indicates that each processor would contribute 100% of its processing capability. Unfortunately, it is extremely difficult to achieve in real-world applications (6).

Table 1: Strong scaling for parallelized neural network

Data $(No.)$	Processor $(No.)$	Time (s)	Speedup
60000	1	43.67	1
60000	2	23.79	1.84
60000	4	14.81	2.95
60000	6	12.17	3.59
60000	8	11.01	3.97

6.1 Strong Scalability

To make data amount large enough for testing strong scalability, the original dataset is replicated 12 times with 60000 pieces of data in total. Table 1 records the results of strong scaling for parallelized neural network with fixed job scale. The red dotted line in Figure 5 represents the ideal speedup result that the speedup should linearly increase with respect to the number of processors, while the black line is the real speedup in our project. The gradual decline in the speedup with more processors can be attributed to following reasons:

1) Amdahl's law illustrates that the speedup is limited by the fraction of the serial part of the software that is not amenable to parallelization (7). That is to say, Amdahl's law gives the upper limit of speedup for a problem of fixed size which is decided by the serial portion of our application.

2) With more processors involved, the increasing data traffic between the nodes cannot be ignored. In real application, all processors have non-zero latency and limited bandwidth, therefore, the data communication time must be counted.

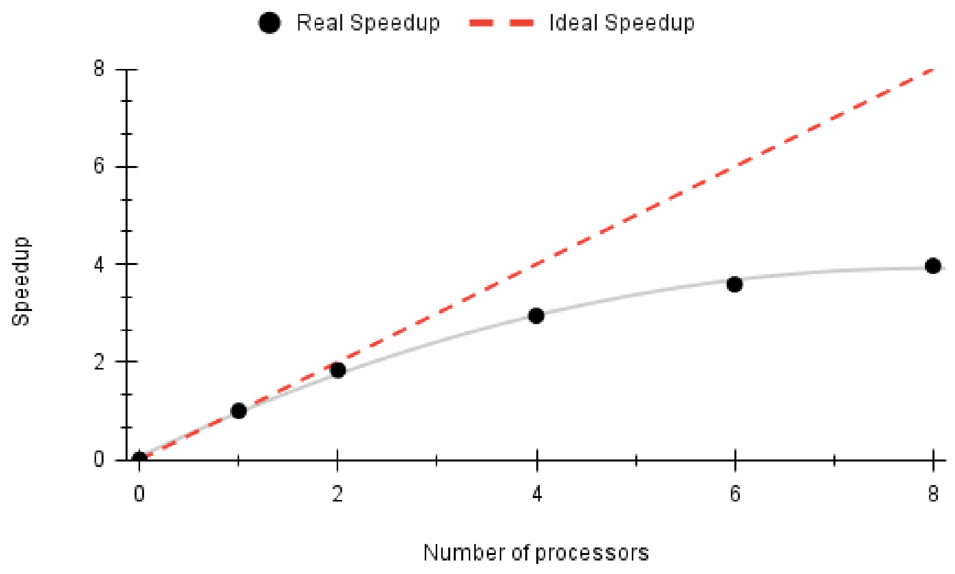


Figure 5: Strong scalability

6.2 Weak Scalability

Unlike Amdahl's assumption, Gustafson points out that the sizes of problems scale with the amount of available resources in practice. In this way, there is no upper limit for the scaled speedup (8). For weak scalability, the test is done by increasing the job size and the number of processing elements simultaneously. The problem size per processor keeps constant, so the expected running time should be the same, as the red dotted line in Figure

6 shown. However, our model reflects a decline in efficiency which is probably due to the load balancing problem. In addition, the communication expend will increase as the problem size increases.

Table 2: Weak scaling for parallelized neural network

Data $(No.)$	Processor $(No.)$	Time (s)
5000	1	4.3
10000	2	4.97
20000	4	5.65
30000	6	6.51
40000	8	7.68

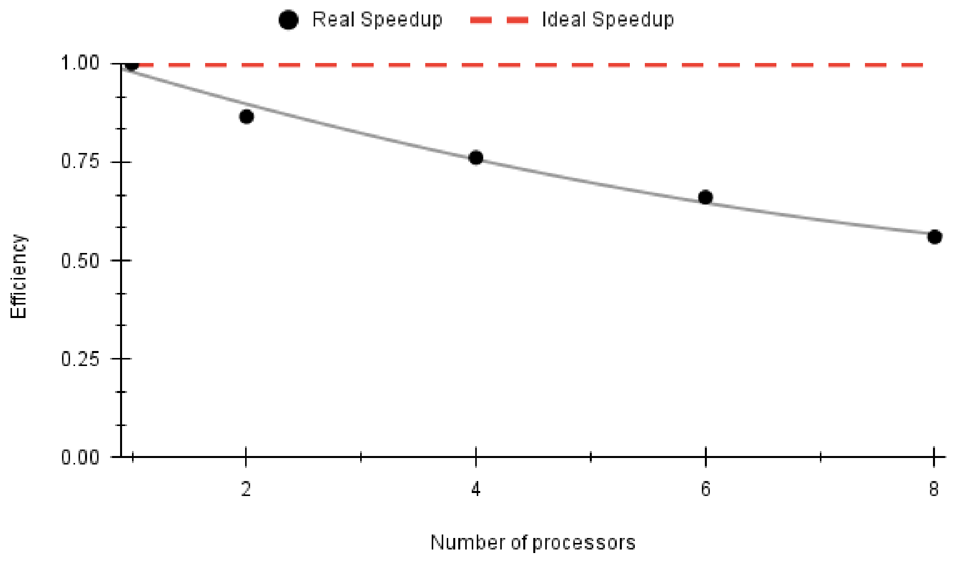


Figure 6: Weak scalability

7 Conclusion

This project successfully parallelized a two-layer neural network using MPI for python. The execution time of training process was largely shortened by data parallelism. Scalability is very important for parallel computing to be efficient, so we studied strong scalability and weak scalability in real application. Speedup and efficiency will decline when a relative large number of processors involved, because of the CPU performance limitation and increasing communication expense.

It is worth noting that our project is conducted on CPU based computing. However, GPU computing is one of the most effective technology for deep learning nowadays. Therefore, some future work might be proposed such as training the parallel neural network models on GPU.

References

[1] P. Chanthini and K. Shyamala, "A survey on parallelization of neural network using mpi and open mp," *Indian Journal of Science and Technology*, vol. 9, no. 19, 2016.

[2] S.-C. Wang, "Artificial neural network," in *Interdisciplinary computing in java programming*. Springer, 2003, pp. 81–100.

[3] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[4] L. Dalcín, R. Paz, and M. Storti, "Mpi for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

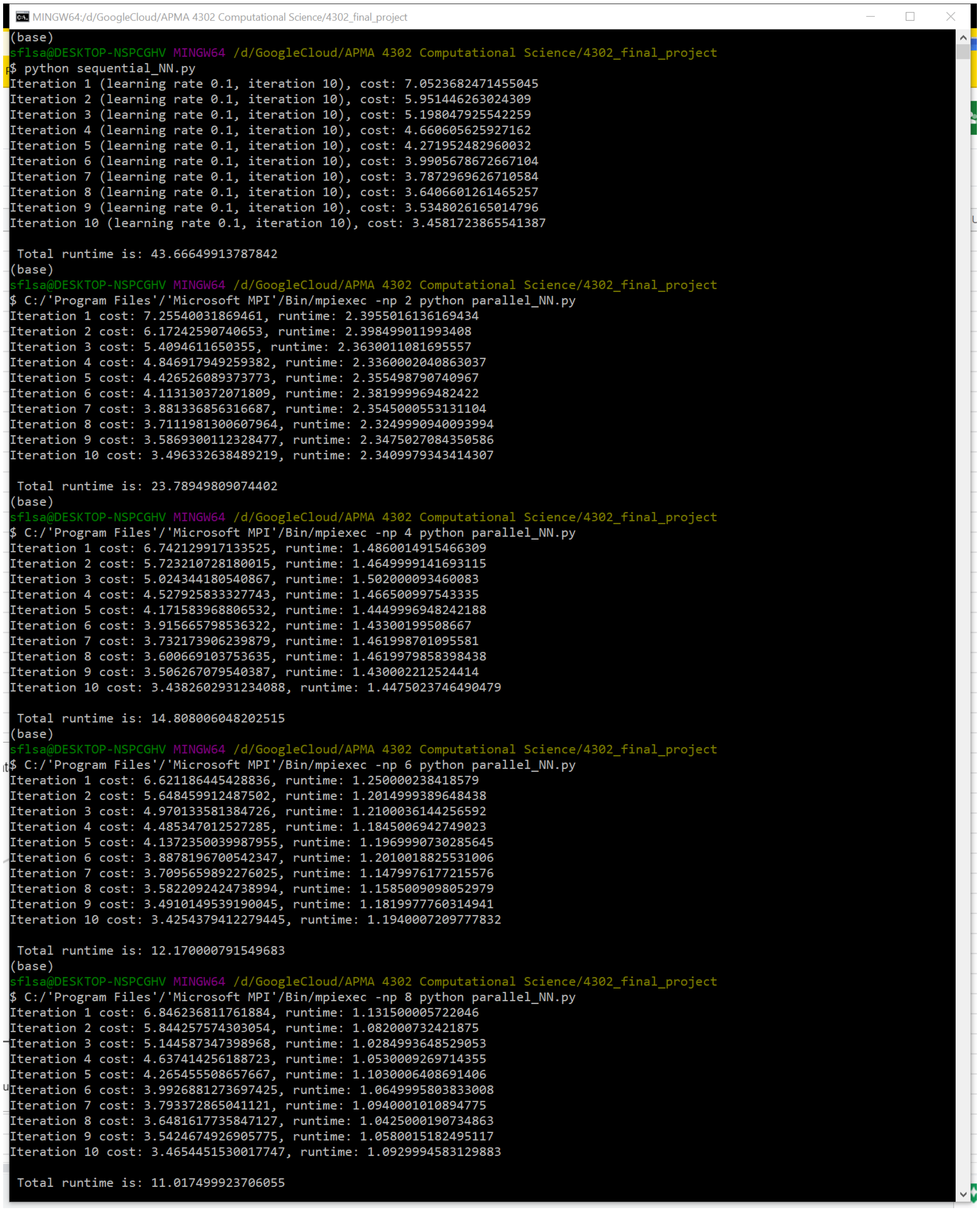
[5] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status update after 12 years of development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021.

[6] X. Li, "Scalability: strong and weak scaling," 2018. [Online]. Available: https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

[7] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[8] L. John, "Gustafson. reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

Appendix



```
MINGW64:/d/GoogleCloud/APMA 4302 Computational Science/4302_final_project
(base)
sflsa@DESKTOP-NSPCGHV MINGW64 /d/GoogleCloud/APMA 4302 Computational Science/4302_final_project
$ python sequential_NN.py
Iteration 1 (learning rate 0.1, iteration 10), cost: 7.0523682471455045
Iteration 2 (learning rate 0.1, iteration 10), cost: 5.951446263024309
Iteration 3 (learning rate 0.1, iteration 10), cost: 5.198047925542259
Iteration 4 (learning rate 0.1, iteration 10), cost: 4.660605625927162
Iteration 5 (learning rate 0.1, iteration 10), cost: 4.271952482960032
Iteration 6 (learning rate 0.1, iteration 10), cost: 3.9905678672667104
Iteration 7 (learning rate 0.1, iteration 10), cost: 3.7872969626710584
Iteration 8 (learning rate 0.1, iteration 10), cost: 3.6406601261465257
Iteration 9 (learning rate 0.1, iteration 10), cost: 3.5348026165014796
Iteration 10 (learning rate 0.1, iteration 10), cost: 3.4581723865541387

 Total runtime is: 43.66649913787842
(base)
sflsa@DESKTOP-NSPCGHV MINGW64 /d/GoogleCloud/APMA 4302 Computational Science/4302_final_project
$ C:/'Program Files'/'Microsoft MPI'/Bin/mpiexec -np 2 python parallel_NN.py
Iteration 1 cost: 7.25540031869461, runtime: 2.3955016136169434
Iteration 2 cost: 6.17242590740653, runtime: 2.398499011993408
Iteration 3 cost: 5.4094611650355, runtime: 2.3630011081695557
Iteration 4 cost: 4.846917949259382, runtime: 2.3360002040863037
Iteration 5 cost: 4.426526089373773, runtime: 2.355498790740967
Iteration 6 cost: 4.113130372071809, runtime: 2.381999969482422
Iteration 7 cost: 3.881336856316687, runtime: 2.3545000553131104
Iteration 8 cost: 3.7111981300607964, runtime: 2.3249990940093994
Iteration 9 cost: 3.5869300112328477, runtime: 2.3475027084350586
Iteration 10 cost: 3.496332638489219, runtime: 2.3409979343414307

 Total runtime is: 23.78949809074402
(base)
sflsa@DESKTOP-NSPCGHV MINGW64 /d/GoogleCloud/APMA 4302 Computational Science/4302_final_project
$ C:/'Program Files'/'Microsoft MPI'/Bin/mpiexec -np 4 python parallel_NN.py
Iteration 1 cost: 6.742129917133525, runtime: 1.4860014915466309
Iteration 2 cost: 5.723210728180015, runtime: 1.4649999141693115
Iteration 3 cost: 5.024344180540867, runtime: 1.502000093460083
Iteration 4 cost: 4.527925833327743, runtime: 1.466500997543335
Iteration 5 cost: 4.171583968806532, runtime: 1.4449996948242188
Iteration 6 cost: 3.915665798536322, runtime: 1.43300199508667
Iteration 7 cost: 3.732173906239879, runtime: 1.461998701095581
Iteration 8 cost: 3.600669103753635, runtime: 1.4619979858398438
Iteration 9 cost: 3.506267079540387, runtime: 1.430002212524414
Iteration 10 cost: 3.4382602931234088, runtime: 1.4475023746490479

 Total runtime is: 14.808006048202515
(base)
sflsa@DESKTOP-NSPCGHV MINGW64 /d/GoogleCloud/APMA 4302 Computational Science/4302_final_project
$ C:/'Program Files'/'Microsoft MPI'/Bin/mpiexec -np 6 python parallel_NN.py
Iteration 1 cost: 6.621186445428836, runtime: 1.250000238418579
Iteration 2 cost: 5.648459912487502, runtime: 1.2014999389648438
Iteration 3 cost: 4.970133581384726, runtime: 1.2100036144256592
Iteration 4 cost: 4.485347012527285, runtime: 1.1845006942749023
Iteration 5 cost: 4.1372350039987955, runtime: 1.1969990730285645
Iteration 6 cost: 3.8878196700542347, runtime: 1.2010018825531006
Iteration 7 cost: 3.7095659892276025, runtime: 1.1479976177215576
Iteration 8 cost: 3.5822092424738994, runtime: 1.1585009098052979
Iteration 9 cost: 3.4910149539190045, runtime: 1.1819977760314941
Iteration 10 cost: 3.4254379412279445, runtime: 1.1940007209777832

 Total runtime is: 12.170000791549683
(base)
sflsa@DESKTOP-NSPCGHV MINGW64 /d/GoogleCloud/APMA 4302 Computational Science/4302_final_project
$ C:/'Program Files'/'Microsoft MPI'/Bin/mpiexec -np 8 python parallel_NN.py
Iteration 1 cost: 6.846236811761884, runtime: 1.131500005722046
Iteration 2 cost: 5.844257574303054, runtime: 1.082000732421875
Iteration 3 cost: 5.144587347398968, runtime: 1.0284993648529053
Iteration 4 cost: 4.637414256188723, runtime: 1.0530009269714355
Iteration 5 cost: 4.265455508657667, runtime: 1.1030006408691406
Iteration 6 cost: 3.9926881273697425, runtime: 1.0649995803833008
Iteration 7 cost: 3.793372865041121, runtime: 1.0940001010894775
Iteration 8 cost: 3.6481617735847127, runtime: 1.0425000190734863
Iteration 9 cost: 3.5424674926905775, runtime: 1.0580015182495117
Iteration 10 cost: 3.4654451530017747, runtime: 1.0929994583129883

 Total runtime is: 11.017499923706055
```

Figure 7: Program running results