UNIVERSITY OF MESSINA

ENGINEERING DEPARTMENT
MASTER DEGREE COURSE IN ENGINEERING AND COMPUTER SCIENCE

PARALLEL PROGRAMMING

Federated Learning With MPI

Supervisor: Prof. Salvatore **Distefano**

Student: Khanh Dung **Dao**

09 November 2021

Contents

Chapter 1

Introduction

With an increasing focus on privacy, Federated Learning(1) has become one of the essential concepts in modern machine learning. Federated learning is geared towards training a model without uploading personal information or identifiable data to a cloud server. As you might already know, a machine learning model needs a lot of data to train. But there are times when the training data is sensitive, and people are growing reluctant to share their personal data with a third party. With growing concerns for privacy, federated learning is now essential in most machine learning applications.

Therefore, I decided to build a simple Federated Learning project using Message Passing Interface (MPI) which is a standardized and portable message-passing standard designed to function on parallel computing architectures. Thanks to this project, we can acquire many new skills in using MPI4py, Tensorflow framework, as well as other tools.

Chapter 2

Federated Learning

Definiton

Traditional machine learning involves a data pipeline that uses a central server (on-prem or cloud) that hosts the trained model in order to make predictions. The downside of this architecture is that all the data collected by local devices and sensors are sent back to the central server for processing, and subsequently returned back to the devices. This round-trip limits a model's ability to learn in real-time.

Federated learning (FL) in contrast, is an approach that downloads the current model and computes an updated model at the device itself (ala edge computing) using local data. These locally trained models are then sent from the devices back to the central server where they are aggregated, i.e. averaging weights, and then a single consolidated and improved global model is sent back to the devices.

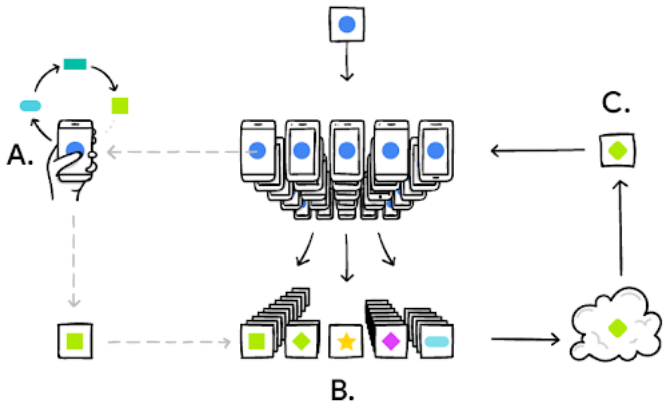

Figure 2.1: Your phone personalizes the model locally, based on your usage (A). Many users' updates are aggregated (B) to form a consensus change (C) to the shared model, after which the procedure is repeated.

In a more general sense, FL(2) allows for machine learning algorithms to gain experience from a broad range of data sets located at different locations. The approach enables multiple organizations to collaborate on the development of models, but without needing to directly share secure data with each other. Over the course of several training iterations, the shared models get exposed to a significantly wider range of data than what any single organization possesses in-house. In other words, FL decentralizes machine learning by removing the need to pool data

into a single location. Instead, the model is trained in multiple iterations at different locations.

Centralized federated learning

In the centralized federated learning setting, a central server is used to orchestrate the different steps of the algorithms and coordinate all the participating nodes during the learning process. The server is responsible for the nodes selection at the beginning of the training process and for the aggregation of the received model updates. Since all the selected nodes have to send updates to a single entity, the server may become a bottleneck of the system.

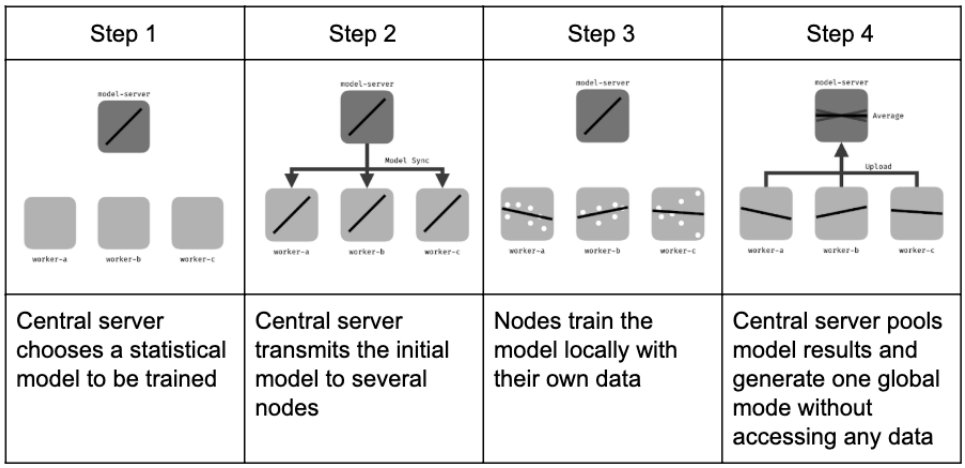

Figure 2.2: Centralized federated learning

1. The central server first sends out the current version of the global model to a subset of client devices.
2. Each device then produces updates to the model parameters based on their individual datasets, such as via stochastic gradient descent.
3. The devices then send their client-model updates back to the central server.
4. The server aggregates the updates to produce an updated version of the global model.
5. The server then sends out the new version to the client devices, and the entire process repeats itself.

This cyclical process can be effective — but relying on a single server can be risky. The central server is a single point of failure. It needs to be able to support large amounts of data transfer between itself and the clients. And it has to be trusted by all parties in the network.

There are still other major risks associated with FL — and with distributed machine learning in general. Bad actors may attempt to compromise the global model, such as through model poisoning or by trying to access or recreate clients' private datasets. These risks can be addressed using decentralized or blockchain-native approaches such as randomized verifier peer selection and by model-update validation in combination with traditional privacy safeguards.

Chapter 3

Message Passing Interface

3.1 Introduction

Parallel programming is using multiple resources such as processors, memories, data to solve a problem, which breaks it down into a series of smaller steps, delivers instructions, and processors execute the solutions at the same time. It is also a form of programming that offers the same results as concurrent programming but, spending in less time and achieving with more efficiency. Many computers, such as laptops and personal desktops, use parallel programming in their hardware to ensure that tasks are quickly completed in the background.

Message Passing Interface (MPI)(3) is a standardized and portable message-passing model designed to function on parallel computing architectures. The MPI standard defines the syntax and semantics of library routines, which are useful by being able to write portable messagepassing programs in some language such as: C, C++, Fortran, Python, and so on. In addition, MPI is specifically used to allow applications to run in parallel across a number of separate computers (nodes) connected by a network, which is distributed architectures working with different memory location as known as distributed memory, and different data. MPI helps cores or memories in distributed system communicate via special send and receive routines which are message passing.

MPI is suitable for single program multiple data model that is a primary programming for large-scale parallel machines. In the other words, MPI is a communication protocol for parallel programming, which specifically used to allow applications to run in parallel across a number of separate computers connected by a network. So, using mpi4py based on python language is the approaching of parallel programming purpose in this project. MPI for Python provides MPI bindings for the Python language, allowing programmers to exploit multiple processor computing systems. mpi4py is is constructed on top of the MPI-1/2 specifications and provides an object oriented interface which closely follows MPI-2 C++ bindings.

3.2 MPI syntax

3.2.1 Communicators and Ranks

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

Here we used the default communicator named **MPI.COMM_WORLD**, which consists of all the processors. For many MPI codes, this is the main communicator that you will need. However, you can create custom communicators using subsets of the processors in MPI.COMM_WORLD. We use **comm.Get_rank()** to get the processor Id and **comm.Get_size()** to get number of processors.

3.2.2 Point-to-Point Communication

Sending and receiving are the two foundational concepts of MPI, which the process of communicating data follows a standard pattern, they are called "point-to-point" communication that is able to send and receive data between 2 specific processors.

The data will be sent from sender rank to receiver rank by tag (rank id). After **MPI_Receive** function is called, the data is already sent to destination rank. In addition, these follow 2 concepts of communication that are blocking and non-blocking communications.

Blocking communications will make a blocking send operation terminating when the message is received by the destination, and a blocking receive operation finishing when a message is received by the caller. There is a waiting state for tasks done in here.

Blocking communication is done using MPI_Send() and MPI_Recv(). These functions do not return (i.e., they block) until the communication is finished. Simplifying somewhat, this means that the buffer passed to MPI_Send() can be reused, either because MPI saved it somewhere, or because it has been received by the destination. Similarly, MPI_Recv() returns when the receive buffer has been filled with valid data.

Otherwise, Nonblocking communication will create a non-blocking send operation completing when the message is sent by sender, even if it has not been received yet. And non-blocking receive operation always terminates immediately, it may not receive any message. There is no wait this communication between ranks and there is a status variable in both sending and receiving operation to query the status of message sent and received respectively.

Non-blocking communication is done using MPI_Isend() and MPI_Irecv(). These function return immediately (i.e., they do not block) even if the communication is not finished yet. You must call MPI_Wait() or MPI_Test() to see whether the communication has finished.

Blocking communication is used when it is sufficient, since it is somewhat easier to use. Non-blocking communication is used when necessary, for example, you may call MPI_Isend(), do some computations, then do MPI_Wait(). This allows computations and communication to overlap, which generally leads to improved performance. Note that collective communication (e.g., all-reduce) is only available in its blocking version up to MPIv2. IIRC, MPIv3 introduces non-blocking collective communication.

3.2.3 Collective Communication

Broadcasting

A broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

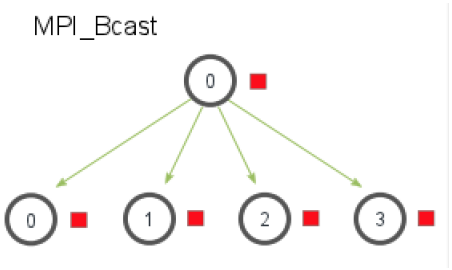

Figure 3.1: MPI_Bcast

Scattering

Scatter is a collective routine that is very similar to Broadcast, it involves a designated root process sending data to all processes in a communicator. The primary difference between Broadcast and Scatter is small but important. Broadcast sends the same piece of data to all processes while Scatter sends chunks of an array to different processes.

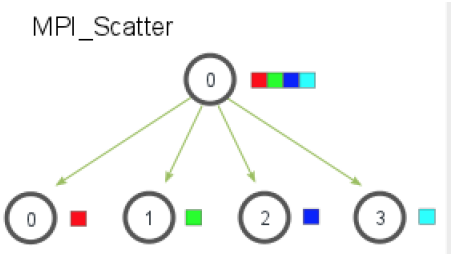

Figure 3.2: MPI_Scatter

Gathering

Gather is the inverse of Scatter. Instead of spreading elements from one process to many processes, Gather takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

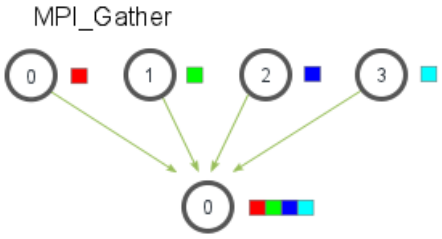

Figure 3.3: MPI_Gather

MPI_Allgather

So far, we have covered two MPI routines that perform many-to-one or one-to-many communication patterns, which simply means that many processes send/receive to one process. Oftentimes it is useful to be able to send many elements to many processes (i.e. a many-to-many communication pattern). MPI_Allgather has this characteristic.

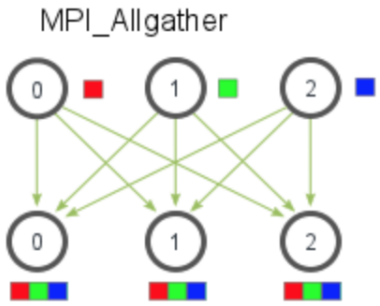

Figure 3.4: MPI_Allgather

Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes. In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast. The illustration below shows how data is distributed after a call to MPI_Allgather.

Just like MPI_Gather, the elements from each process are gathered in order of their rank, except this time the elements are gathered to all processes. Pretty easy, right? The function declaration for MPI_Allgather is almost identical to MPI_Gather with the difference that there is no root process in MPI_Allgather.

Chapter 4

Federated Learning using Tensorflow and MPI4PY

4.1 Image Classification with MNIST Dataset

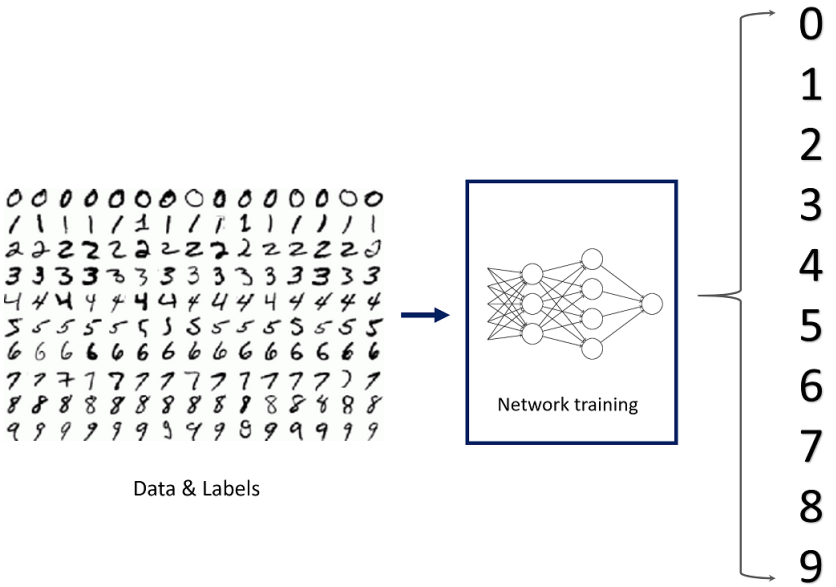

Figure 4.1: Image Classification with MNIST Dataset

When you start learning deep learning with different neural network architectures, you realize that one of the most powerful supervised deep learning techniques is the Convolutional Neural Networks (abbreviated as "CNN").

Additionally though, in CNNs, there are also Convolutional Layers, Pooling Layers, and Flatten Layers. CNNs are mainly used for image classification although you may find other application areas such as natural language processing.

4.1.1 Convolutional Layers

The convolutional layer(4) is the very first layer where we extract features from the images in our datasets. Due to the fact that pixels are only related to the adjacent and close pixels, convolution allows us to preserve the relationship between different parts of an image. Convolution is basically filtering the image with a smaller pixel filter to decrease the size of the

image without losing the relationship between pixels. When we apply convolution to 5x5 image by using a 3x3 filter with 1x1 stride (1-pixel shift at each step). We will end up having a 3x3 output (64% decrease in complexity).

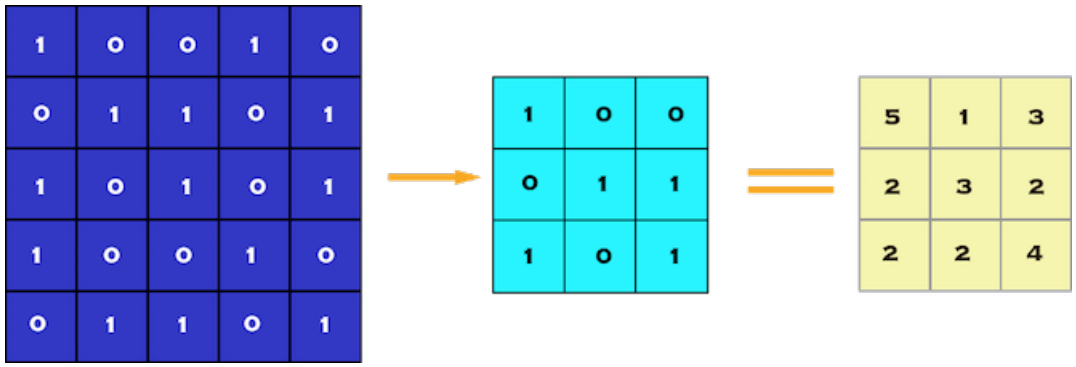

Figure 4.2: Convolution of 5 x 5 pixel image with 3 x 3 pixel filter (stride = 1 x 1 pixel)

4.1.2 Pooling Layer

When constructing CNNs, it is common to insert pooling layers after each convolution layer to reduce the spatial size of the representation to reduce the parameter counts which reduces the computational complexity. In addition, pooling layers also helps with the overfitting problem. Basically we select a pooling size to reduce the amount of the parameters by selecting the maximum, average, or sum values inside these pixels. Max Pooling, one of the most common pooling techniques, may be demonstrated as follows:

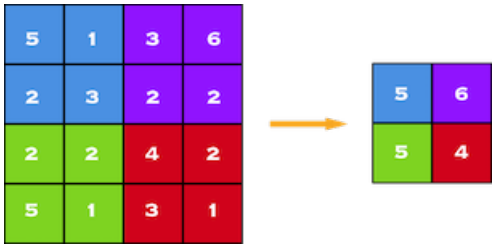

Figure 4.3: Max Pooling by 2 x 2

4.1.3 A Set of Fully Connected Layers

A fully connected network is our RegularNet where each parameter is linked to one another to determine the true relation and effect of each parameter on the labels. Since our time-space complexity is vastly reduced thanks to convolution and pooling layers, we can construct a fully connected network in the end to classify our images. A set of fully-connected layers looks like this:

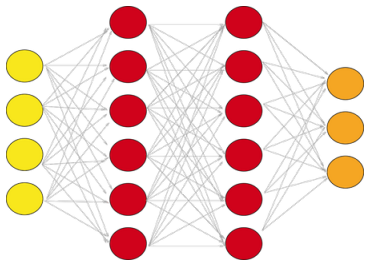

Figure 4.4: A fully connected layer with two hidden layers

Now that you have some idea about the individual layers that we will use, I think it is time to share an overview look of a complete convolutional neural network.

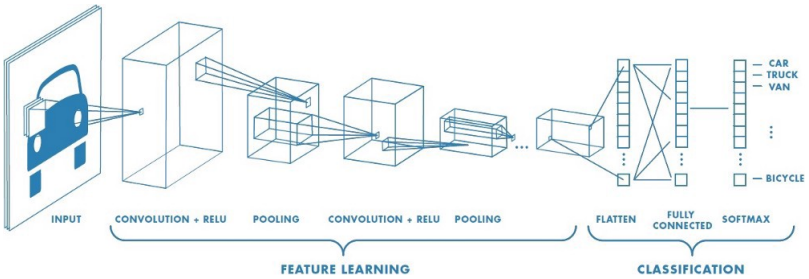

Figure 4.5: A Convolutional Neural Network Example

And now that you have an idea about how to build a convolutional neural network that you can build for image classification, we can get the most cliche dataset for classification: the MNIST dataset, which stands for Modified National Institute of Standards and Technology database. It is a large database of handwritten digits that is commonly used for training various image processing systems.

The MNIST database contains 60,000 training images and 10,000 testing images taken from American Census Bureau employees and American high school students [Wikipedia]. Therefore, in the second line, I have separated these two groups as train and test and also separated the labels and the images. x_train and x_test parts contain greyscale RGB codes (from 0 to 255) while y_train and y_test parts contain labels from 0 to 9 which represents which number they actually are.

In this project, I used Tensorflow(5) version 2 to build the Neural Network model to traning the Mnist dataset.

4.2 Centralized Federated Learning algorithms

4.2.1 Idea of Centralized Federated Leanrning In Sequential

Idea of Centralized Federated Learning in sequential following these steps:

1. Initialize Server and Clients.

2. Download and split N datasets from the Mnist dataset and temporarily treat it as the N local dataset of the our clients and Server.

3. Server will send the newest parameters to the Clients and then each Client produces updates to the model parameters based on their individual datasets, such as via stochastic gradient descent.

4. The Clients then send their client-model updates back to the central server.

5. The server aggregates the updates to produce an updated version of the global model.

6. The server then sends out the new version to the client devices.

7. Step 3,4,5 and 6 will be repeated until convergence.

4.2.2 Idea of Centralized Federated Learning In Parallel

Idea of Centralized Federated Learning in parallel following these steps:

1. The Master process read a data set.

2. The Master process will scatter all dataset divided by number of processes to each process.

3. Master process will broadcast the newest weight of model to each process.

4. Each process produces updates to the model parameters based on their individual datasets.

5. Each process then send their client-model updates back to the Master.

6. The Master process aggregates the updates to produce an updated version of the global model.

7. The Master process then broadcast the new version to all processes, and the entire process repeats itself until convergence.

4.3 Implementing with MPI

4.3.1 Decomposition and Partition

Data decomposition: Split input data by dividing all data by the number of processes (MPI) in each time running program. In detail, splitting the amount of data in MPI implementation, which help the program will access the data independently in each process during parallel executions. Then the output data will be composited to main process.

There are 6 tasks in whole program being executed to give out the final result, in which the number of concurrent tasks can be run and the maximum number of tasks running at an

executing point is 2 for MPI implementation. So, the number of these tasks is degree of concurrency of decomposition and maximum degree.

Partitioning based on data decomposition and task decomposition, the number of partitioning depends on the number of processes that created during executing program. In addition, the tasks in concurrency will be executed the same in every process, and the data is different from each of them, so the concurrent tasks will be replicated as the number of sub-tasks respectively to the number of processes.

Centralized Federated Learning

Tasks in Concurrency

+ 3rd Task: Each Process produces updates to the model parameters based on their individual datasets.

+ 4th Task: The processes then send their client-model updates back to the central server (Master process).

Tasks in Sequence:

+ 1st Task: Master process splits data based on division of the amount of data to the number of processes.

+ 2nd Task: Scatter data and broadcast data to each process before doing parallel.

+ 5th Task: Master process aggregates the updates to produce an updated version of the global model.

+ 6th Task: The Master process then broadcast the new version of weight to the client devices, and the entire steps repeat itself until convergence.

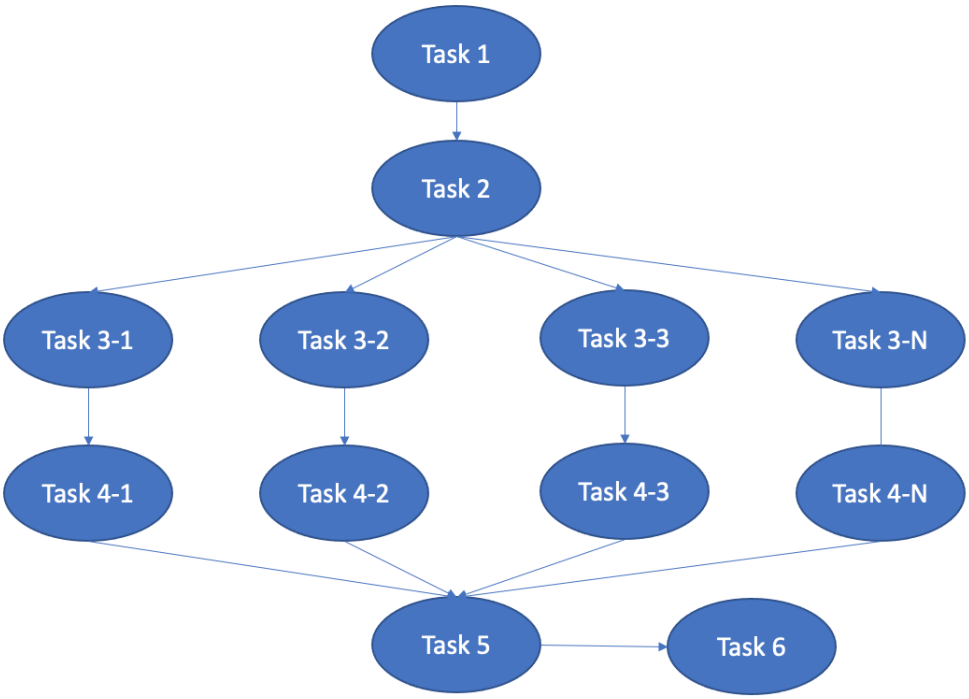

Figure 4.6: Task dependency graph

4.3.2 Communication

I used collective communication techniques which are scatter, broadcast, gather, allgather method to transfer, update or share data among processes for this project. Broadcast helps send the same data (the updated parameter) from root process to all processes. And scatter involves to send different chunks data from root process to all processes in a communicator. In contrast, gather is an inverse of scatter method that collects elements from many processes to one single process.

4.3.3 Aggregation

Aggregation is another common design pattern that's used in parallel applications. In parallel programs, the data is divided into units so that it can be processed across cores by a number of threads. At some point, there is a need to combine data from all the relevant sources before it can be presented to the user. This is where aggregation comes into the picture.

The resulting model is sent to clients to start local training from it. When on-device training is finished, the weights of the local models(**processes**) are sent to the server(**Master Process**) using the gather function in MPI. Here, aggregation is done in a weighted averaging (**Task 5**) manner where clients with more data influence more significantly the newly aggregated model.

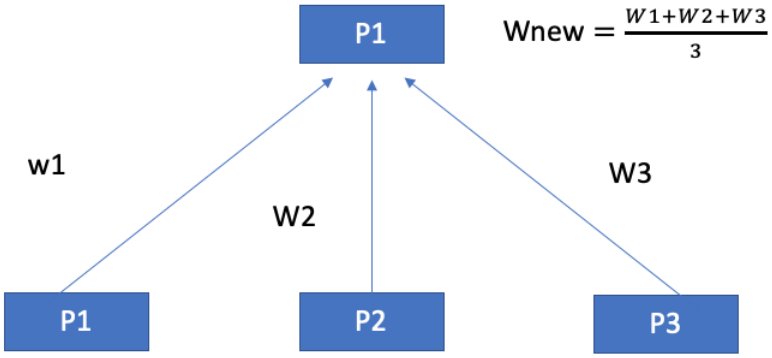

Figure 4.7: Federated Averaging

4.3.4 Mapping

In general, the number of tasks in a decomposition exceeds the number of processing elements available. For this reason, a parallel algorithm must also provide a mapping of tasks to processes.

Task dependency graphs can be used to ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance).

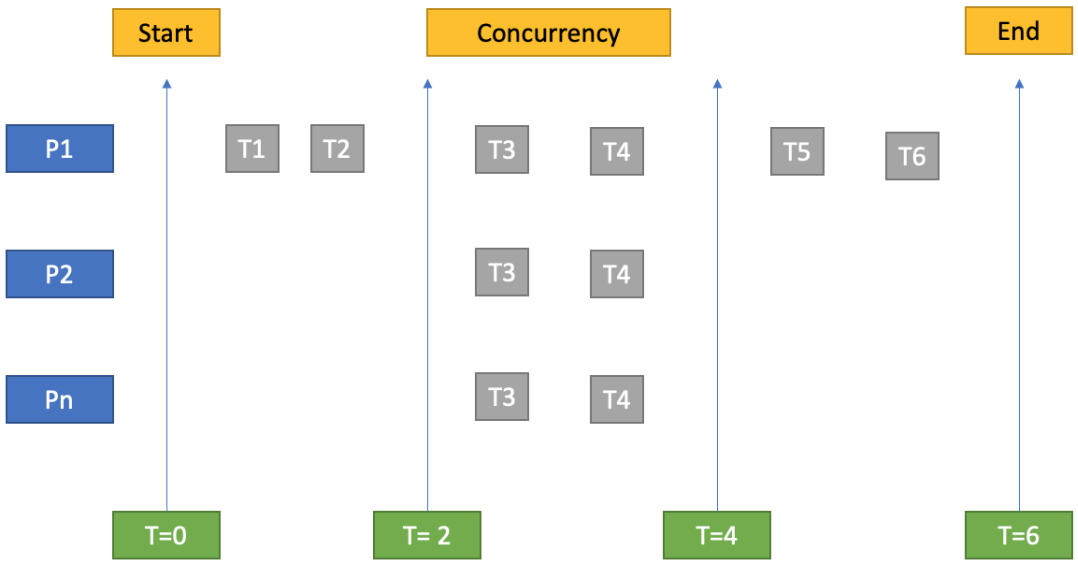

Figure 4.8: MPI Mapping the task to process for Minimum Idling

4.4 Results

In academy purpose, I used 2 processors in MPI for running Centralized Federated Leanrning . The device I used the Macbook pro 2015 - 13 inch with con figuration:
CPU: 2.7GHz dual-core Intel Core i5 processor
RAM: 8GB of 1866MHz LPDDR3 onboard memory
Operation System: MacOS

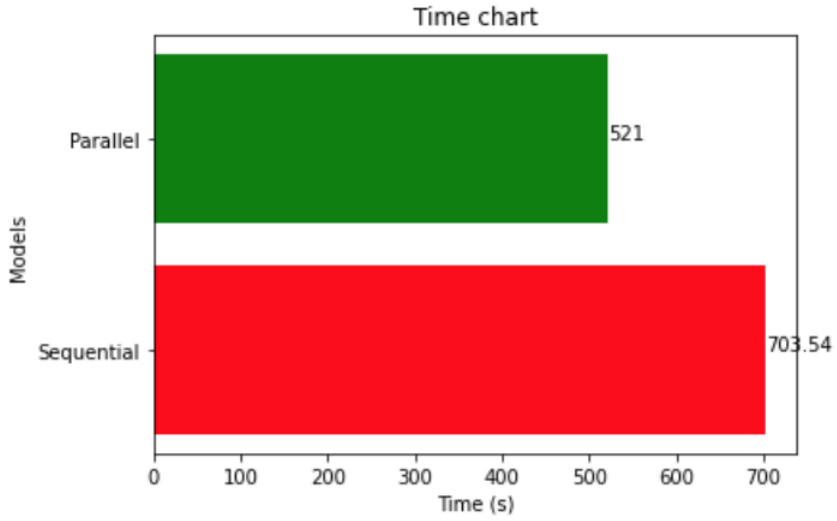

Figure 4.9: Time chart

It is clearly seen that the Centralized Federated learning in Sequential model consumed the highest amount of time (704.54 seconds). By contrast, the Centralized Federated Learning in Parallel consumed just around 521 second.

Applying Amdahl Law in parallelism, we can calculate the speed-up to measure the performance increase when increasing the processors or cores, and can find out efficiency that express the ability of exploiting the resources, which measures the work rate per processor or core. Following the result of program, we can use some Amdahl Law's formulas to find out speed-up and efficiency value:

$$fs = Ts/(Ts + Tp)(Serialfraction) \tag{4.1}$$

$$Sn = T1/Tn = n/(n * fs + (1–fs))(Speedup) \tag{4.2}$$

$$En = Sn/n = 1/(n * fs + (1–fs))(Efficiency) \tag{4.3}$$

Following equations above, we can calculate the Speed up and Efficency of the model:

- Centralized Federated Learning: $\mathbf{S} = 1.352$, $\mathbf{E} = 67.6\%$

Besides the time-consuming variables, we can check two other parameters are accuracy and loss value of the deep learning model in Centralized Federated Learning.

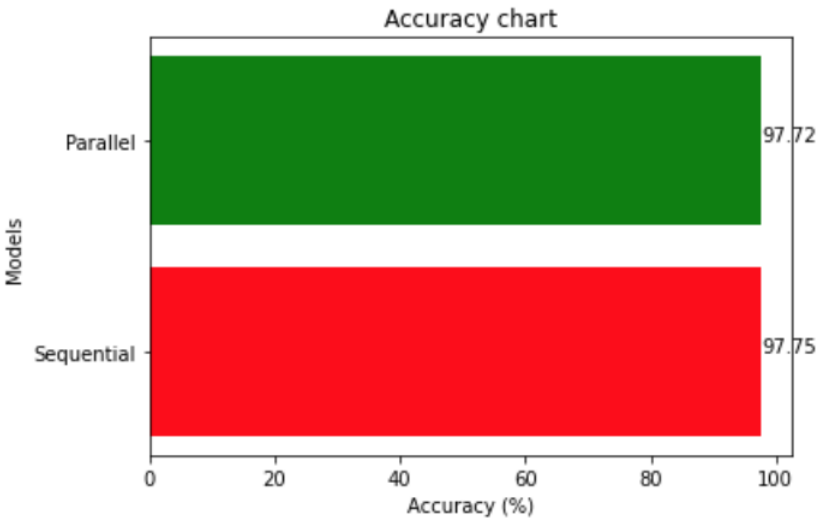

Figure 4.10: Accuracy chart

The above bar chart presents the accuracy of Centralized Federated Learning algorithms in sequential and parallel. It is easy to see that the Sequential and Parallel model get the similar accuracy value around 97.75% and 97.72% respectively.

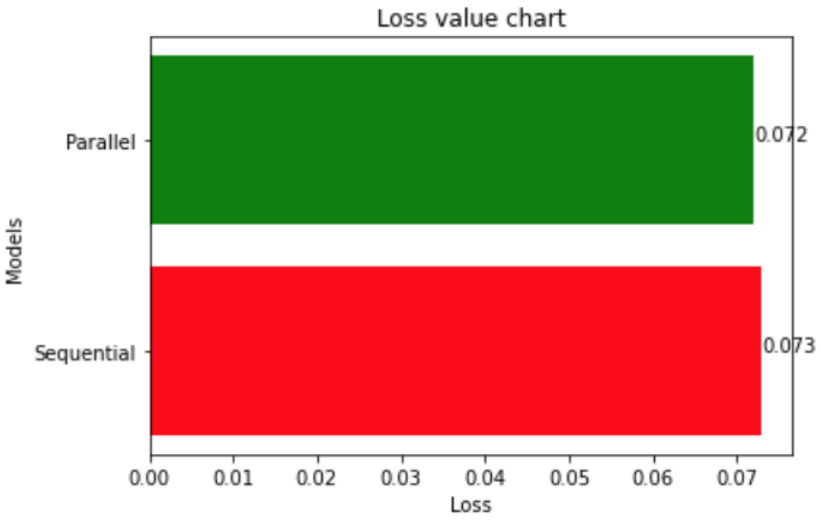

Figure 4.11: Loss value chart

Similarly, the loss value of Centralized Federated Learning in Parallel is as low as (around 0.072) the Sequential(at 0.073).

Chapter 5

Conclusions

The goal of this project was to be familiar with general points of parallel programming and implementation of Federated Learning to reach the boost of performance. The proposed aim was successful achieved and results of boosting were presented in MPI implementation.
However, these results are not optimal because of testing in environment by only a multicore laptop with a 2 cores whereas MPI works best on node clusters of a big computer network such as super computer or cloud system (distributed system of hardware).

Since this is a project within the scope of academic research, shortcomings and limitations are inevitable. However, applying parallelism to algorithms instead of doing sequential implementation, which saves execution time, optimizes the use of available hardware, and gives out many solutions to problems topic in life.

Bibliography

[1] Q. YANG, *Federated Machine Learning: Concept and Applications*. No Starch Press, 2019.

[2] *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. [Online]. Available: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html

[3] L. Dalcin, *MPI for Python*. No Starch Press, 2021.

[4] *Convolutional Neural Networks*. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[5] *https://www.tensorflow.org/*. [Online]. Available: https://www.tensorflow.org/