Machine Learning with Applications 13 (2023) 100483

Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

Minimization of high computational cost in data preprocessing and modeling using MPI4Py

E. Oluwasakin [a], T. Torku [a], S. Tingting [a], A. Yinusa [a], S. Hamdan [b,*], S. Poudel [b], N. Hasan [b], J. Vargas [c], K. Poudel [b]

[a] Computation and Data Science Program, Middle Tennessee State University, TN, USA
[b] Computer Science Program, Middle Tennessee State University, TN, USA
[c] Engineering Technology, Middle Tennessee State University, TN, USA

ARTICLE INFO

Keywords:
High computational cost
Performance
Data preprocessing
MPI
MPI4Py
Machine learning
Deep learning
Data parallelism
Model parallelism

ABSTRACT

Data preprocessing is a fundamental stage in deep learning modeling and serves as the cornerstone of reliable data analytics. These deep learning models require significant amounts of training data to be effective, with small datasets often resulting in overfitting and poor performance on large datasets. One solution to this problem is parallelization in data modeling, which allows the model to fit the training data more effectively, leading to higher accuracy on large data sets and higher performance overall. In this research, we developed a novel approach that effectively deployed tools such as MPI and MPI4Py from parallel computing to handle data preprocessing and deep learning modeling processes. As a case study, the technique is applied to COVID-19 data from state of Tennessee, USA. Finally, the effectiveness of our approach is demonstrated by comparing it with existing methods without parallel computing concepts like MPI4Py. Our results demonstrate promising outcome for the deployment of parallel computing in modeling to minimize high computational cost.

1. Introduction

Data has become indispensable in scientific and economic research, particularly in the field of Internet of Things (IoT). The abundance of data poses challenges in analysis and processing, leading to the increasing importance of deep learning. Deep learning, a prominent trend in machine learning, can uncover complex patterns from labeled and unlabeled data. However, deep learning models face performance issues on devices with limited CPUs and require substantial computational power on clusters and supercomputers. To address this, optimization techniques involving software and hardware trade-offs are developed to minimize computing procedures in modeling. New frameworks are being developed to enable tasks like training and inference to run on clusters with different architectural nodes. MPI (Message Passing Interface) plays a crucial role in optimizing high computational processes by allowing multiple functions to work simultaneously on one task, improving efficiency for deep learning tasks that involve substantial information processing. By distributing tasks across nodes, latency is reduced, and memory buffers called verbs are allocated to each node. Effective data preprocessing and modeling procedures are vital for model performance. Traditional methods may be time-consuming, especially when dealing with large datasets and deep learning models that require labor-intensive preprocessing. MPI4Py parallelization is an effective technique to speed up these operations. MPI4Py, a Python module, integrates with the MPI library, enabling data scientists to accelerate data preprocessing and modeling using high-performance computing methods. Jobs can be assigned to multiple cores or computing clusters using MPI4Py, facilitating quick analysis of massive volumes of data and running complex models.

2. Background

2.1. The challenges of speed, performance, and computational cost in data preprocessing and modeling

Generally, in data science, there are steps in getting started with data preprocessing and modeling (Jassim & Abdulwahid, 2021). Data can be explored visually to identify patterns or build a prediction model. For these steps, speed, performance, and computational cost are the three essential factors of the data preprocessing and modeling journey. These factors are so crucial that when any of them faces challenges, it can cause the whole process to stop working or even fail to work at all. The concept of speed is relatively clear, but what

* Corresponding author.
E-mail addresses: eo3j@mtmail.mtsu.edu (E. Oluwasakin), Thomas.Torku@mtmail.mtsu.edu (T. Torku), ts7f@mtmail.mtsu.edu (S. Tingting), aay2g@mtmail.mtsu.edu (A. Yinusa), sah9j@mtmail.mtsu.edu (S. Hamdan), sp2ai@mtmail.mtsu.edu (S. Poudel), mh2ay@mtmail.mtsu.edu (N. Hasan), jvargas@mtsu.edu (J. Vargas), kpoudel@mtsu.edu (K. Poudel).

https://doi.org/10.1016/j.mlwa.2023.100483
Received 19 May 2023; Received in revised form 3 July 2023; Accepted 3 July 2023
Available online 17 July 2023

about performance and computational cost? Moreover, when gathering the data, its size and complexity will dictate how long it takes, but these factors can help determine what type of data to work with. The speed specifies how long it takes for this data to download or process. The performance suggests that the size or structure of the data makes any calculations too slow or cumbersome. The computational cost specifies the robustness of the computing resources, whether to choose computers with limited CPUs or clusters of computers with effective Graphics Processing Units (GPUs). Evaluating speed and performance is essential when deciding what data to work with. Furthermore, after knowing what type of data to work with, it is time to clean it. This step ensures that the dataset is free from missing or erroneous values by removing any outliers or other data points that are not needed or unnecessary (Ferrão et al., 2016).

2.1.1. The challenges of data preprocessing steps

When working with massive datasets, it is essential to carefully consider how to tackle tasks to optimize performance and computational cost (Famili et al., 1997). Data preprocessing is manipulating and preparing data to be analyzed and used in the next stage, whether in machine learning or another preprocessing step. This process might involve removing noise, identifying outliers, or aggregating data sets, as mentioned previously (Ferrão et al., 2016). However, if not done correctly, data preprocessing can result in high computational costs, reduced speed performance, and increased time to completion. Each data preprocessing step can be time and resource consuming, especially when the goal is high performance across different datasets. When choosing between processing time and accuracy tradeoffs, it is helpful to understand what code does and how much it affects runtime. The stochastic gradient descent algorithm typically outperforms the gradient descent technique for big datasets in a given problem, such as linear regression. This example portrays the role of a faster algorithm. An important factor in data preprocessing is the choice of proper hardware for a given problem, which enhances the minimization of computation time without sacrificing accuracy. Some tasks are better suited for CPUs, while others work better on GPUs. For example, most neural network architectures run best on GPUs because they have specialized circuitry to perform matrix multiplication and convolution operations. Matrix multiplication is computationally expensive due to its complexity, but GPU architectures are designed to handle these operations quickly. A good strategy for choosing which type of hardware (CPU vs. GPU) to use is to look at the kinds of calculations that need to be performed (matrix multiplication vs. convolution) (Pawliczek et al., 2014). If there are many multiplications, then a GPU should be used because the intensive calculations will take less time than the CPU. When choosing where the processing should occur (either locally or remotely), it must be remembered that the delay before receiving feedback from remote servers can significantly affect performance.

2.1.2. Why parallel computing in data preprocessing and data modeling?

Parallel computing is an intelligent technique for accelerating computations in data preparation and modeling. Dividing the work across multiple processors or computers allows for concurrent processing and faster completion times. Deep learning with complex models and large datasets often benefits from parallel programming using graphics processing units (GPUs). GPUs excel at tasks such as image recognition and language translation, and can outperform CPUs in terms of memory bandwidth and numerical processing power. However, GPUs require asynchronous algorithms to coordinate their work. In the field of deep learning, MPI libraries such as MPI4Py are widely used for high performance computing (HPC) operations. HPC operations guarantee fast turnaround times without sacrificing accuracy. To effectively use HPC operations, Python's sequential algorithms must be rewritten as threaded algorithms. Python supports threading, which allows individual parts of a program to work simultaneously, while MPI allows direct communication over the network. MPI (Message Passing Interface) is a library of routines that facilitates the passing of messages between distributed systems. MPI4Py, an extension of MPI, simplifies the process by providing a readable pseudocode syntax similar to popular Python packages such as Scikit Learn. Overall, parallel computing and his MPI4Py play an important role in optimizing computing speed and managing large amounts of data in various industries.

2.2. High-performance computing (HPC)

Generally, high-performance computing (HPC) involves the application of computer clusters or parallel computer systems to resolve different challenges in the computing domain. These high-performance supercomputers and some specific software can handle large computations that require detailed procedures. It is standard in computing to segment massive datasets into different tasks or chunks. Thus, these chunks are called "threads" (What is high performance computing? - high-performance computing news analysis, 2015).

2.2.1. Why it is important to use high-performance computing?

The human brain can process images, judgments, and thoughts in milliseconds. This can be completed in a nanosecond by a powerful computer. Due to its capacity for parallel computing, it can process information at an astounding rate. To use parallel computing, two or more processors must work on the same issue simultaneously (i.e., all working together on a problem instead of one at a time). If one processor is unable to fix the issue, another one can. This aids in speedy calculations and makes it extremely challenging to overload an HPC system. Parallel computing enables the quick resolution of issues that would otherwise take humans years to solve (What is high-performance computing (HPC), 0000). It has even been used to model protein folding, evaluate oil reservoirs, and create new automobiles, aircraft, and household devices.

2.2.2. The benefits of using HPC for large datasets

Since high-performance computing has become more popular, there is a greater need for large datasets. Parallel processing with MPI can help solve this problem by splitting a data set into pieces that are then processed simultaneously. It is also much faster when there are multiple CPUs to work with, and the users do not have to wait for all the CPUs to finish before moving on to the next set of information (What is high-performance computing (HPC), 0000). With HPC, large amounts of data can be processed quickly without wasting too much time or space on hard drives. Not only that, but it saves time and resources because it does not require lengthy downloads and compiles (What is high-performance computing (HPC), 0000). Furthermore, data is better protected because instead of having one instance where it resides-like on a personal computer or server-many copies of the same information are spread over various computers worldwide. Not only does this protect against natural disasters like hurricanes, but also hacking attacks! Moreover, just as important as what HPC offers is what it requires from the user: time and patience. Depending on how much data needs to be analyzed and how powerful the CPU is, MPI may take longer than expected to complete its tasks.

2.3. The basics of data modeling

Message Passing Interface, or MPI, is a set of guidelines that allow software applications running on several machines in a cluster (or grid) to coordinate and communicate. This can be accomplished via sharing data access or passing messages (Gropp et al., 1996). MPIs are an alternative form of communication across clusters or grids that can be utilized when the machines running programs are not precisely in the same place. In parallel computing, this programming interface makes sending messages between two or more processes easier. Developers can access several processors on their machine or in a cluster of computers by using this interface with Python. Any application on a single processor can run concurrently on various other processors without requiring changes to the original code.

2.4. Message Passing Interface (MPI) in parallel computing

MPI stands for Message Passing Interface. It is a set of rules that allow programs running on different machines in a cluster (or grid) to communicate and synchronize with each other. This can be done by passing messages or sharing data access (Gropp et al., 1996). When the machines running programs are not in the exact location, MPIs may be used as an alternative communication between clusters or grids. This programming interface facilitates the transfer of messages between two or more processes in parallel computing. Using this interface with Python gives developers access to multiple processors on their computer or in a cluster of computers. Any program that can run on one processor can be executed simultaneously on many different processors without modifying the original code.

2.5. MPI4Py: High-performance computing for data preprocessing

MPI4Py is a Python package designed for high-performance computing and parallel computation. It acts as a bridge between Python and HPC systems, enabling the utilization of multiple CPU cores on compute clusters. It supports data preprocessing and modeling frameworks such as Pandas SQL, NumPy, Scikit-Learn, and Matplotlib. With MPI4Py in parallel mode, large datasets can be efficiently preprocessed. It also provides access to distributed clusters for data modeling and offers GPU support through OpenCL or CUDA, enabling fast model training on large datasets. An example of parallel computation involves dividing a dataset into subsets, performing operations on each subgroup (e.g., removing outliers), and combining the results. Overall, MPI4Py enhances the interoperability of Python programs and HPC systems, facilitating efficient data preprocessing and modeling tasks.

2.5.1. Comparing MPI4Py with other Python parallel computing libraries

MPI4Py has a few benefits over other Python parallel computing tools, such as Cython and Charm4Py (Fink et al., 2021).

- It is free.
- It can be used through Python scripts or a command-line interface.
- It is highly scalable because it can run on clusters of computers using the MPI message passing standard.
- It does not need additional software to function as Dask or Multiprocessing.

3. Data and model parallelism in deep learning

Deep learning is integral to our data modeling, with large-scale neural networks such as Convolutional Neural Networks (CNN) or Recurrent Neural Networks (RNN) representing the forefront of artificial intelligence research. However, data parallelism and model parallelism are computational techniques in the deep learning toolkit that enable researchers to accelerate the training of neural networks to obtain greater performance and speed in modeling. However, these deep learning models require vast quantities of training data to be effective, with small datasets frequently leading to overfitting and poor performance on large datasets. One solution to this issue is parallelization in data modeling, which enables the model to fit the training data better, resulting in greater accuracy on refined data sets and enhanced performance overall.

3.0.1. Parallel and distributed methods in deep learning modeling

The computation for deep neural networks can be parallelized or distributed in various ways by using several computers or cores. Listed below are some of the possible ways (Verma, 2021):

- **Local Training**: This method requires saving the model and data on a single computer while concurrently employing the graphics processing unit (GPU) or the computer's multiple cores.
- **Multi-Core Processing**: One or more processing cores from the same machine can be used to fit the data and the model, and those cores can share memory (PRAM (Parallel Random Access Memory) model).

Distributed Training: Suppose we determine that saving the data and models on a single machine would compromise the performance of both the machine and the process. In that case, we can utilize numerous machines to attain a greater degree of performance. This is when storing data and models on a single machine is impractical for both machines. Increasing memory usage in the processing of data and models frequently causes challenges that can be resolved by implementing the following techniques (Verma, 2021):

- **Data Parallelism**: It is a method for distributing large, memory-intensive datasets across multiple processors, allowing us to train and analyze the data faster.
- **Model Parallelism**: Occasionally, neural network models might grow so large that they cannot be saved on a single device. In order to make use of the parallelism given by the model, we partition it across multiple processors. Forward and reverse propagation can be employed for machine-to-machine communication, with the result saved on a single machine at a single network layer. Both unidirectional and bidirectional communication is possible.

4. The purpose of the research

Data preprocessing and modeling in Python for data science and analytics, particularly in machine learning and deep learning, require handling large volumes of data for accurate modeling and predictions. To achieve high-speed processing, accurate results, efficient performance, and reduced computational costs, conventional methods of data preparation, such as statistical methods, are inadequate. Instead, it is crucial to optimize packages that can decrease computational costs and enhance performance in data preprocessing and modeling.

This research aims to minimize the computational cost required for high-speed processing of big data. To accomplish this, parallel computing tools like MPI and MPI4PY are deployed, leveraging system design to accelerate the data preprocessing stage. The proposed approach is scalable to large datasets containing millions of data records. While it is known that cloud computing platforms process large datasets faster, this research provides additional tools to expedite the computation and processing of big data in such platforms. Even users without access to cloud computing resources can benefit significantly from this work, as it enables faster processing of large datasets compared to traditional approaches.

Furthermore, this research focuses on reducing the high computational cost associated with training deep learning models. The speed, convergence rate, and accuracy of these models depend on the choice of hyperparameters, parameters, and the platforms they are executed on (CPU or GPU). The findings demonstrate that combining parallel computing tools like MPYI4PY with GPUs significantly improves the speed and convergence of deep learning models.

5. Related work

The implementation of parallel computing for the optimization of the deep learning model has been explored by many researchers using various parallel processing concepts such as OpenMPI, MPI, MP, and CUDA. This is necessary because the production and deployment times for most modeling processes done on single or multi-core machines (CPUs or GPUs) are usually more when not parallelized. Most deep learning modeling tasks are usually carried out on GPUs for faster modeling production, but these GPUs are often expensive in some cases. To tackle these challenges, modeling can be parallelized using

parallelization packages in C, C++, or Fortran. Because of the practical challenges experienced by most programmers when using these compiled programming languages, parallel computing libraries have been developed for Python and some other interpreted programming languages. This section discusses the related works by other researchers on parallelizing most deep learning models for fast model deployment and production. This section also describes other related works that have used parallel computing for other complex tasks. Furthermore, in their research, Barrachina et al. (2021) present a framework for training deep neural networks on computer clusters that has the following appealing features:

- It is written in Python, exposing an approachable interface that offers an accessible entry point for the novice.
- It is extensible, providing a customizable tool for the more experienced user in deep learning.
- It covers the key functionality found in convolutional neural networks.
- To create data parallelism, both of these components are necessary. NumPy is used for the efficient execution of numerical kernels, and MPI through MPI4Py is used for communication. This gives it decent inter-node parallel performance, achieved through data parallelism.

To emulate parallelism in deep learning, Hewett and I.I. (2020) offer a linear algebraic technique that enables the parallel distribution of any tensor within the DNN. Their research shows that linear operators such as broadcast, sum-reduce, and halo exchange can be used in parallel data transit operations. The adjoint, also known as backward, operators required for gradient-based training of DNNs are manually created by describing the relevant spaces and inner products. Moreover, Zhang et al. (2021) show a way for distributed object detection that utilizes the MPI4Py module and several CPUs and GPUs for concurrent distributed processing. In another research, Jiang et al. (2022) describe the implementation of the mpi4py function module for parallel algorithms and executing multi-process synchronization processing based on a trained DeepUnet model. Large and high-resolution image data received through remote sensing should be handled quickly and with more processors. MPI for Python (mpi4py) is currently the most widely adopted Python binding for the message-passing interface (MPI). Dalcin and Fang (2021) explain the gradual growth of mpi4py's additions and capabilities over the past decade, including support for CUDA-aware MPI implementations up to the MPI-3.1 specification and additional tools that bridge the gap between Python application development and MPI-based parallel distributed computing. Barajas et al. (2019) investigate the wall time difference between preaugmented data and live data augmentation methods for training a convolutional neural network to predict tornadoes. Moreover, they evaluate training on GPU and CPU systems using augmented data sets of various sizes and also examine how altering the number of GPUs utilized for training will affect a convolutional neural network. Furthermore, Rogowski et al. (2023) cover the concept, implementation, and feature set of mpi4py.futures and its performance compared to existing shared and distributed memory architectures. On a shared-memory system, they also show that mpi4py.futures typically outperform concurrent.futures in terms of throughput (tasks per second) and bandwidth. On a Cray XC40 platform, mpi4py.futures is compared to Dask, a well-known Python library for parallel computing. Conclusively, they demonstrate that, although it gives more inconsistent results, mpi4py.futures outperforms Dask in the majority of scenarios.

Fang et al. (2020) cover how the Python-based ptychography reconstruction software developed at Brookhaven National Laboratory became a high-performance toolset. This modification uses several programming techniques related to performance and productivity, such as mpi4py, PyCUDA, Scikit-Cuda, and CuPy. The transformation's in-depth coverage is also provided for various performance-critical parallelization and GPU offloading techniques. The original serial code is executed 10–100 times faster on CPUs with multiple cores and over 1000 times faster on GPUs when the resultant code is used than the original. To further implement the application of MPI for parallel processing, Anderson et al. (2017) provides a minimal Python framework for distributed neural network training on GPUs or CPUs. The framework is based on Keras. The system coordinates training using the Message Passing Interface (MPI) protocol and can submit tasks to supercomputers. They also cover the software's features, how to use it, and its performance on systems of various sizes for a high-energy physics benchmark problem.

Navarro et al. (2014) analyzes parallel computing and its applications in data-parallel problems using a GPU architecture. They give an overview of parallel computing and present a comprehensive examination of its applications in n-body, collision detection, Potts Model, and simulations using cellular automata. In the survey, they highlight that designing better GPU-based algorithms for computational physics problems and achieving speedups that can reach up to two orders of magnitude when compared to sequential implementations are all possible by comprehending the GPU architecture and its massive parallelism programming model.

Niu et al. (2018) focuses on the application of parallel computing techniques in concept-cognitive learning using the framework of granular computing. A parallel computing framework is specifically created for huge data to extract global granular concepts by mixing local granular concepts. To extract final concepts from multi-source data, an efficient information fusion approach is used to combine the concepts from each individual source. Niu et al. (2018) found that, under careful design and optimization, parallel computing could enhance the learning process for concept-cognitive learning by speeding up the process and making it more scalable.

Mehrabi et al. (2021) provides a method for the unobtrusive definition and integration of cloud-based and shared-memory parallel computing. It enables transparent and efficient utilization of diverse computing platforms. It accomplishes this by putting the suggested ideas into practice in @PT (Annotation Parallel Task), a parallel programming environment that makes use of native Java annotations as its linguistic building blocks. Through this, Mehrabi et al. (2021) was able to develop unified programming concepts.

Benalla et al. (2021) proposes a parallel computing method to address the computational complexity of Dempster's Rule of combination. While the rule is widely used, its computational complexity can be high, especially when dealing with large amounts of evidence. They were able to find some parts in Dempster's rule that could be optimized through parallelism. Through applying their method, the computational complexity was reduced and the process was sped up.

6. Methodology

There are various steps involved in data preprocessing. Each step requires data science libraries and packages to implement data collection, cleaning, transforming, and reduction for faster and more effective performance procedures. Moreover, for the modeling part, there are several categories of deep learning, such as supervised and unsupervised deep learning. And each category is further subdivided into different algorithms, such as Artificial Neural Networks (ANN), Convolution Neural Networks (CNN), and Recurrent Neural Networks (RNN) for various applications. In this research, we adopted the supervised deep learning approach, RNN, and Regression as a Machine Learning Model for analysis and prediction.

6.1. Datasets

The Covid-19 dataset was retrieved from the Tennessee Department of Health website. It contains data on the daily active cases of Covid-19 over the course of 766 days.

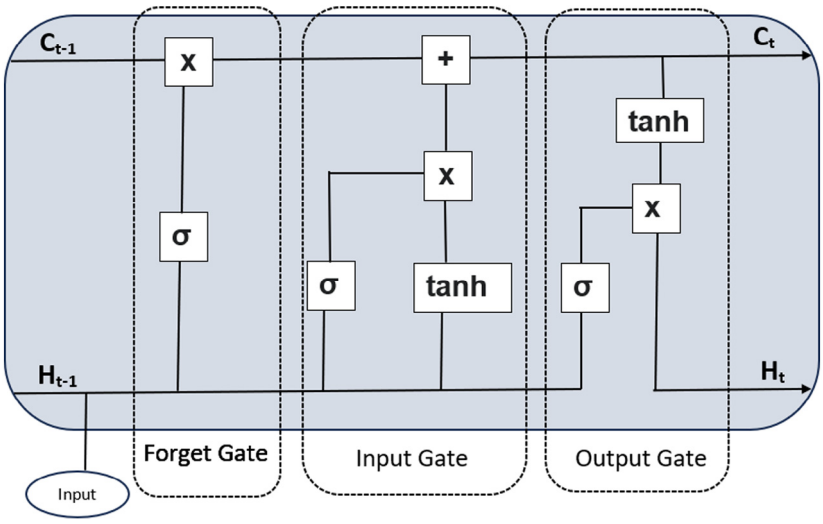

Fig. 1. This is what an LSTM module looks like.

6.2. Long Short-Term Memory (LSTM) models

RNNs are generally able to retain information throughout the model. The more complex the sequence, however, the more difficult it is for the RNN to retain the information. In order to bypass this issue, LSTM models were created, which are a type of RNN that specialize in retaining information over long periods of time. They are able to solve a variety of different problems, which has led to them being widely used. An LSTM model has four neural network layers that interact with each other (Zhao, 2023). They interact through what is known as the cell state, which transfers information through the whole model. LSTM models have the ability to modify the information that is sent through the cell state. They do this through three gates, which are named as follows: input, forget, and output. The input gate tells how much information should be added to the cell state, while the forget gate tells how much information should be removed. The output gate applies the changes from the previous two gates to the cell state.

Fig. 1 shows what one module of an LSTM consists of. An LSTM is made up of a series of modules, each of which processes information as it passes through them. Each gate consists of a sigmoid function, which is denoted by the sigma symbol, and a pointwise multiplication operation, denoted by X. A pointwise addition is denoted by the '+' sign. Information is taken from the hidden state, H_t, sent through the gates and operations, and then finally applied to the cell state, C_t.

BiLSTM models added onto what regular LSTM models could do. Unlike LSTM models, BiLSTM models can take inputs from both directions. They do this by adding an additional layer that reverses the direction of information flow. This kind of model is very useful for natural language processing tasks. A downside to using BiLSTM models over LSTM models are that they are slower and requires more time for training.

6.3. Gated Recurrent Unit (GRU)

GRU (Gated Recurrent Unit) models are another type of recurrent neural network that address the issue of retaining information over long sequences. Similar to LSTM models, GRU models are designed to capture long-term dependencies in data.

GRU models have a simpler architecture compared to LSTM models, consisting of two main components: an update gate and a reset gate. These gates control the flow of information within the model, allowing it to selectively update and reset its hidden state.

The update gate determines how much of the previous hidden state should be combined with the current input, while the reset gate determines how much of the previous hidden state should be ignored. These gates enable the GRU model to adaptively update its hidden state, selectively incorporating relevant information while discarding irrelevant information.

Fig. 2 depicts the layout of a GRU module. When H_{t-1} and x_t enter the module, they are each multiplied by their respective weights. They are then sent through the gates as shown before the vectors generated by them are applied to the new hidden state.

The simplified architecture of GRU models makes them computationally less expensive compared to LSTM models. They have fewer parameters and are easier to train. However, this simplicity also means that GRU models may have a slightly reduced capacity to capture complex long-term dependencies compared to LSTM models.

BiGRU models, similar to BiLSTM models, extend the capabilities of GRU models by allowing inputs from both directions. They achieve this by adding a backward layer that processes the input sequence in reverse order. This bidirectional processing enhances the model's ability to understand contextual information from both past and future contexts, making it beneficial for tasks such as natural language processing, where the context of a word can be influenced by both preceding and succeeding words.

6.4. System design

In this work, we adopted the compartmental Susceptible–Infected–Recovered (SIR) model worked on by Torku et al. (2021), where they obtained data from Tennessee State Health Department.

- **Research Procedure for Data Preprocessing**
  - We adopted python programming language as the programming software
  - Python packages like NumPy for numerical calculation, Matplotlib for data visualization, Scikit-Learn for modeling, and SciPy for scientific implementation
- **Research Procedure for the Deep Learning Modeling**
  - We further applied LSTM, BiLSTM, and Gated Recurrent Unit (GRU) for data modeling.
  - PyTorch Framework for high computational graph support at runtime.
- **Data and Model Parallelism**
  For high computational performance, speed, and reduced computational cost, we adopted MPI4Py [MPI replica for parallel

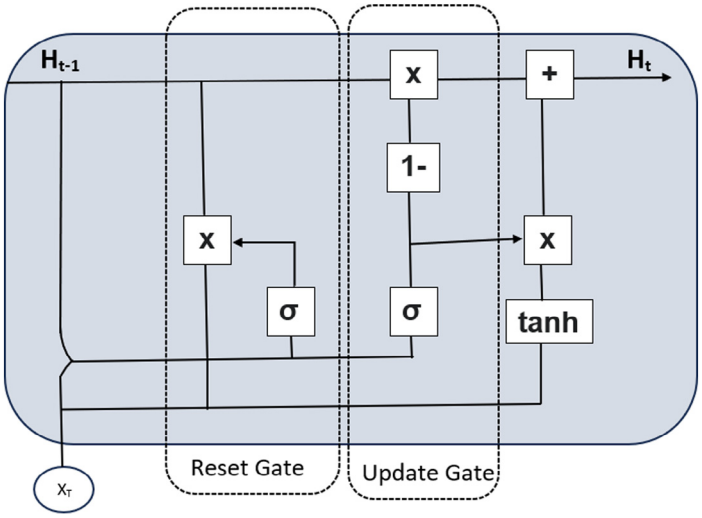

Fig. 2. This is what a GRU module looks like.

computing in C/C++ and Fortran], a Python parallel computing library, for the parallelization of both the data preprocessing stage and the modeling stage.

Data Preprocessing and Modeling Parallelization Procedures This research employed the MPI4Py to parallelize the data preprocessing and modeling tasks on CPU and GPU machines. The data preprocessing task was segmented into chunks, enabling the dataset to be parallelized using the MPI4Py module. In this case, we arranged the dataset in ranks, which were assigned to different processors using the $comm.Get-rank()$ function of the MPI4Py. The $MPI.COMM-WORLD$ function was initialized for communication amongst the ranks and processors. Synchronously, the size of every divided chunk was also initialized for the entire parallelization steps. Afterward, the data parameters were gathered from the CPUs or GPUs, depending on the machine used. Moreover, we also utilized the MPI4Py for the parallelization procedure for the modeling task. The necessary modules, such as MPI, PyTorch, LSTM, Adam, and torch, were imported into the code. After that, the MPI environment was initialized by creating an $MPI.COMM-WORLD$ communicator and determining the rank and size of the communicator. The rank, as was mentioned previously, is the unique identifier of each process within the communicator, and the size is the total number of processes in the communicator. After initializing the MPI environment, the model and optimizer used for training were defined. The model parameters were then distributed across the different GPUs or machines using the scatter and gather functions of the comm object. The root process (this procedure, rank 0, was adopted) distributed the model parameters to the other processes. Furthermore, we trained the model on the local data by looping over the number of epochs and using the optimizer to update the model parameters. After training the model, the parameters were gathered from all the (CPUs or GPUs) using the gather function of the comm object. The root process collected the parameters from all the other processes. Finally, we ran the code on the root process by averaging the model parameters across all the (CPUs or GPUs). Afterward, we updated the model with the averaged parameters. This procedure completed the process of using MPI4Py to parallelize the modeling task utilizing the necessary modules (see Fig. 3).

From the process model pipeline in Fig. 2, the Python library adopted is MPI4py, which allows MPI programs to run on multiple platforms and processors. It reduces the computational cost of computing tasks by minimizing the number of required operations, such as data transfer and communication. After data is remotely accessed via data sources such as databases, websites, and the cloud, the data preprocessing task is carried out by extracting, preprocessing, and transforming the data in the pipeline using python tools such as Pandas SQL, NumPy, Scikit-Learn, and Matplotlib in conjunction with the MPI4py library. This is done to efficiently increase the computational performance by distributing it across multiple cores or nodes. Next, the transformed data is modeled for regression tasks using deep learning frameworks like Pytorch, Long Short-Term Memory (LSTM), BiLSTM, and Gate Recurrent Unit (GRU). In particular, the LSTM and BiLSTM are used in the regression tasks, while the GRU is used to complement the LSTM and BiLSTM to solve the problem of exploding and vanishing gradients. MPI4py is deployed to optimize tasks' training and testing speeds. Finally, for performance comparison, the computational accuracy and speed are measured in the models with MPI4py and the models without MPI4py based on data analysis.

Table 1
The comparison of performance for CPU timing in data preprocessing with the application of parallel computing.

Data preprocessing performance comparison		
	Without MPI	With MPI
Scaled data	0.154 s	0.098 s
Unscaled data	0.122 s	0.064 s

7. Results and discussion

The machine learning models were each trained using the Covid-19 dataset over the course of 1500 epochs. The data was split with 80% of the data being used for training and 20% being used for testing. The batch size used differed between models. The LSTM and BiLSTM models used a batch size of 128, while the GRU model used a batch size of 100. The following graph depicts the performance of the models for their respective categories (see Fig. 4).

From the comparison, we can see that the data preprocessing with MPI4Py is faster, with a CPU Timing of 0.0413 s, while the data preprocessing without MPI4Py gives a range of 0.17 and 0.16 s with and without scaling for the CPU timing. Table 1 below depicts the actual CPU timing for the performance comparison with and without the MPI4Py application (see Table 2).

Table 3 depicts each model's performance in terms of total training time and each model's error. Using the CPU, the inclusion of MPI had no effect on the RMSE of the model, but did make training time shorter. using CUDA, the inclusion of MPI gave the same RMSE for each model as it did on the CPU. Using MPI on CUDA made the total training time for all of the models longer than without it, however, for LSTM and

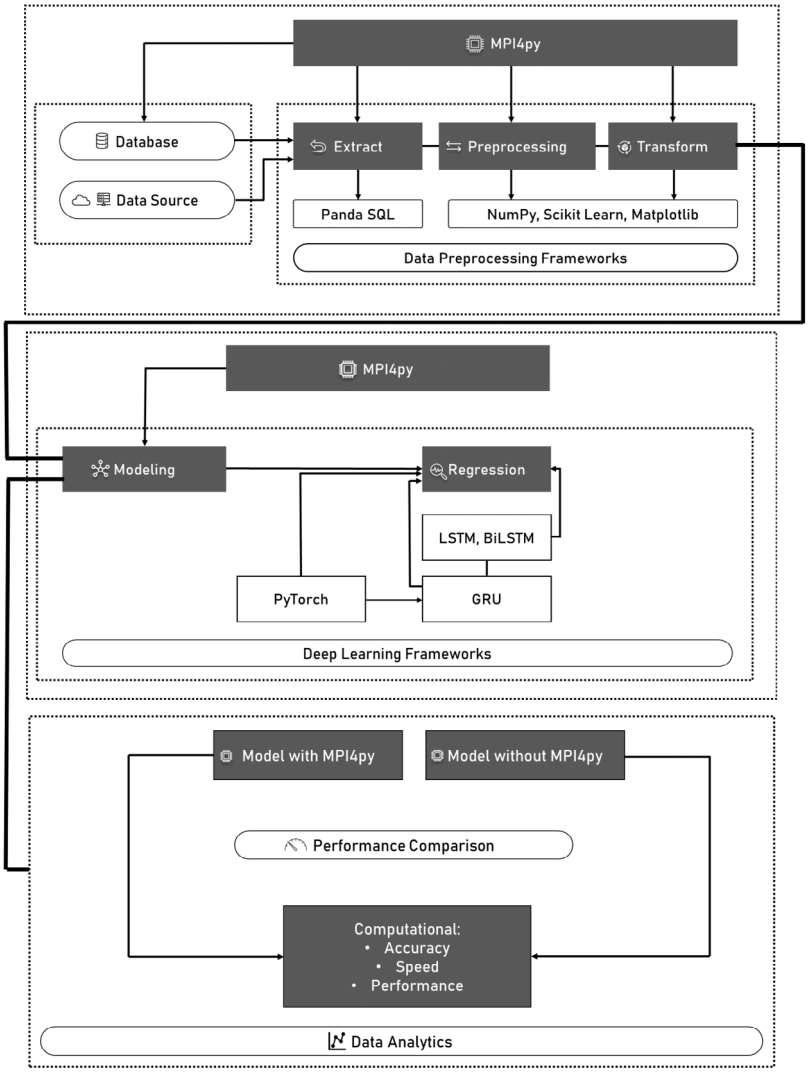

Fig. 3. Process model pipeline showing the data preprocessing section to data analytics section.

GRU models, MPI kept their RMSE lower than if they had been trained without it. Based on these results, if any of the three deep learning models are being trained on a CPU, then using MPI would be beneficial. On CUDA, using MPI would be dependent on which model is being trained.

8. Conclusion

In conclusion, our research focused on addressing the challenges of computationally expensive deep learning models by parallelizing them using MPI (Message Passing Interface) or multi-core CPUs. We recognized the importance of effective data preprocessing concepts in transitioning from the production stage to the deployment stage, including quality assessment procedures, cleaning processes, transformations, and reduction techniques. These preprocessing stages play a crucial role in developing accurate deep learning models and ensuring reliable data analysis.

However, data preprocessing can be challenging due to the relatively poor quality of the data and the complexity associated with building activities. Additionally, large-scale deep learning models, such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs), require substantial amounts of training data to achieve optimal performance. Insufficient data can lead to overfitting and poor generalization when applied to larger datasets. To overcome this limitation, parallelization in data modeling has emerged as a solution, enabling models to effectively learn from more extensive training data, resulting in improved accuracy and overall performance on large datasets.

Table 2
The comparison of performance for CPU timing in deep learning modeling with the application of parallel computing.

Deep learning modeling performance comparison		
	Without MPI	With MPI
CPU	9970 s	54 s
CUDA (GPU)	80 s	2.62 s

Considering the significant computational cost involved in data preprocessing and deep learning modeling, we developed a methodology that effectively utilized parallel computing tools such as MPI and MPI4Py. We applied this methodology to health and financial data, showcasing its potential applicability in diverse domains. Through experimental evaluation, we compared our parallel computing approach with other methods that did not incorporate parallelization, such as MPI4Py, and demonstrated its effectiveness in minimizing computational costs while maintaining high performance.

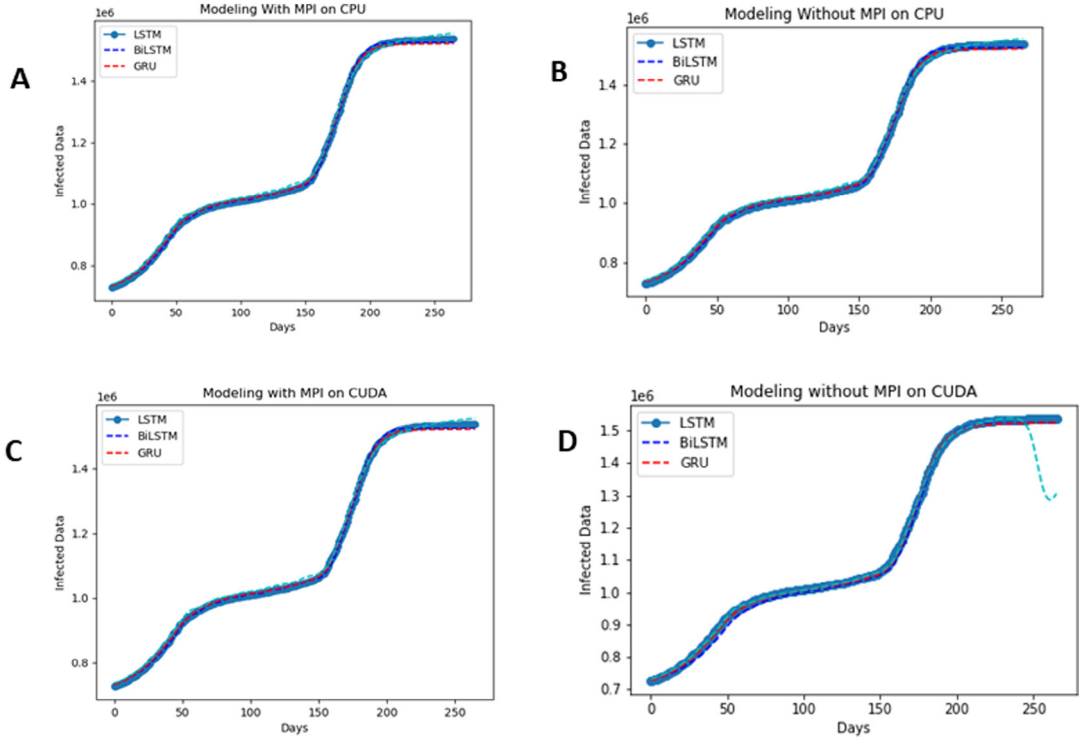

Fig. 4. Deep learning modeling graphical representation without and with the implementation of MPI4Py on CPU and CUDA(GPU).

Table 3
The comparison for the root mean square deviation (RMSE) and total training time (in seconds) for each model in their given environment.

	Without MPI (CPU)			With MPI (CPU)		
Models	LSTM	BiLSTM	GRU	LSTM	BiLSTM	GRU
Error (RMSE)	9115.19	14584.27	10764.89	9115.19	14584.27	10764.89
Performance (Time)	844.42	1635.38	1068.47	827.03	1633.98	1062.49
	Without MPI (CUDA)			**With MPI (CUDA)**		
Models	LSTM	BiLSTM	GRU	LSTM	BiLSTM	GRU
Error (RMSE)	11733.63	14213.05	15831.15	9115.19	14584.27	10764.89
Performance (Time)	865.08	1722.26	1119.98	892.54	1766.52	1132.13

In the future, there are several avenues for further research and improvement. First, exploring advanced techniques for data preprocessing, such as incorporating domain-specific knowledge or leveraging transfer learning approaches, could enhance the quality and relevance of the preprocessed data. Second, investigating different parallel computing strategies and architectures, such as GPU acceleration or distributed deep learning frameworks, could further optimize the computational efficiency of the modeling process. Additionally, the interpretability of deep learning models remains a crucial aspect, and developing techniques for understanding and explaining the decision-making processes of parallelized models would enhance their trustworthiness and practical applicability.

Furthermore, future studies can focus on scaling up the proposed methodology to handle even larger datasets, exploring techniques for distributed data preprocessing, and investigating the trade-offs between computational cost, accuracy, and scalability in parallelized deep learning modeling.

In conclusion, our research has demonstrated the benefits of parallelizing deep learning models using MPI and highlighted the importance of effective data preprocessing. By considering the recommendations and future studies outlined above, researchers can further enhance the efficiency, interpretability, and scalability of parallelized deep learning models, making them invaluable tools for reliable data analytics across various domains.

CRediT authorship contribution statement

E. Oluwasakin: Conceptualization, Methodology, Software. **T. Torku:** Conceptualization, Methodology, Software. **S. Tingting:** Conceptualization, Methodology, Software. **A. Yinusa:** Conceptualization, Methodology, Software. **S. Hamdan:** Writing – review & editing, Validation. **S. Poudel:** Writing – review & editing, Validation. **N. Hasan:** Writing – review & editing, Validation. **J. Vargas:** Conceptualization, Methodology, Software, Supervision, Writing – review & editing. **K. Poudel:** Conceptualization, Methodology, Software, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Anderson, D., Vlimant, J.-R., & Spiropulu, M. (2017). An MPI-based python framework for distributed training with keras.

Barajas, C. A., Gobbert, M. K., & Wang, J. (2019). Performance benchmarking of data augmentation and deep learning for tornado prediction. In *2019 IEEE international conference on big data (Big Data)* (pp. 3607–3615). http://dx.doi.org/10.1109/BigData47090.2019.9006531.

Barrachina, S., Castelló, A., Catalán, M., Dolz, M. F., & Mestre, J. I. (2021). PyDTNN: A user-friendly and extensible framework for distributed deep learning. *The Journal of Supercomputing*, *77*(9), 9971–9987. http://dx.doi.org/10.1007/s11227-021-03673-z.

Benalla, M., Achchab, B., & Hrimech, H. (2021). On the computational complexity of Dempster's rule of combination, a parallel computing approach. *Journal of Computer Science*, *50*, Article 101283. http://dx.doi.org/10.1016/j.jocs.2020.101283.

Dalcin, L., & Fang, Y.-L. L. (2021). Mpi4py: Status update after 12 years of development. *Computing in Science & Engineering*, *23*(4), 47–54. http://dx.doi.org/10.1109/MCSE.2021.3083216.

Famili, A., Shen, W.-M., Weber, R., & Simoudis, E. (1997). Data preprocessing and intelligent data analysis. *Intelligent Data Analysis*, *1*(1), 3–23. http://dx.doi.org/10.1016/S1088-467X(98)00007-9, Retrieved from https://www.sciencedirect.com/science/article/pii/S1088467X98000079.

Fang, Y.-L. L., Ha, S., Huang, X., Yan, H., Dong, Z., Chu, Y. S., Campbell, S. I., Xu, W., & Lin, M. (2020). Accelerated Computing for X-ray Ptychography at NSLS-II. (pp. 141–157). http://dx.doi.org/10.1142/9789811204579_0008, Retrieved 2023-04-12, from https://www.worldscientific.com/doi/abs/10.1142/9789811204579_0008.

Ferrão, J., Oliveira, M., Janela, F., & Martins, H. (2016). Preprocessing structured clinical data for predictive modeling and decision support: A roadmap to tackle the challenges. *Applied Clinical Informatics*, *07*(04), 1135–1153. http://dx.doi.org/10.4338/ACI-2016-03-SOA-0035, Retrieved 2023-04-12, from http://www.thieme-connect.de/DOI/DOI?10.4338/ACI-2016-03-SOA-0035.

Fink, Z., Liu, S., Choi, J., Diener, M., & Kale, L. V. (2021). Performance Evaluation of Python Parallel Programming Models: Charm4py and mpi4py. arXiv. Retrieved 2023-04-12, from http://arxiv.org/abs/2111.04872, arXiv:2111.04872 [cs].

Gropp, W., Lusk, E., Doss, N., & Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, *22*(6), 789–828. http://dx.doi.org/10.1016/0167-8191(96)00024-5, Retrieved 2023-04-12, from https://linkinghub.elsevier.com/retrieve/pii/0167819196000245.

Hewett, R. J., & I.I., T. J. G. (2020). A linear algebraic approach to model parallelism in deep learning. CoRR, abs/2006.03108 Retrieved from https://arxiv.org/abs/2006.03108.

Jassim, M. A., & Abdulwahid, S. N. (2021). Data Mining preparation: Process, Techniques and Major Issues in Data Analysis. *IOP Conference Series: Materials Science and Engineering*, *1090*(1), Article 012053. http://dx.doi.org/10.1088/1757-899X/1090/1/012053, Retrieved 2023-04-12, from https://iopscience.iop.org/article/10.1088/1757-899X/1090/1/012053.

Jiang, M., Zhang, X., & Zhang, C. (2022). Research on parallel technology of sea and land segmentation based on deep learning. In L. Xiao, & D. Xu (Eds.), *Society of photo-optical instrumentation engineers (SPIE) conference series, Vol. 12083* (p. 120831N). http://dx.doi.org/10.1117/12.2623443.

Mehrabi, M., Giacaman, N., & Sinnen, O. (2021). Unified programming concepts for unobtrusive integration of cloud-based and local parallel computing. *Future Generation Computer Systems*, *115*, 700–719. http://dx.doi.org/10.1016/j.future.2020.09.024.

Navarro, C. A., Hitschfeld-Kahler, N., & Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, *15*(2), 285–329. http://dx.doi.org/10.4208/cicp.110113.010813a.

Niu, J., Huang, C., Li, J., & Fan, M. (2018). Parallel computing techniques for concept-cognitive learning based on granular computing. *International Journal of Machine Learning and Cybernetics*, *9*(11), 1785–1805. http://dx.doi.org/10.1007/s13042-018-0783-z.

Pawliczek, P., Dzwinel, W., & Yuen, D. A. (2014). Visual exploration of data by using multidimensional scaling on multicore CPU, GPU, and MPI cluster. *Concurrency Computations: Practice and Experience*, *26*(3), 662–682. http://dx.doi.org/10.1002/cpe.3027, Retrieved from https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3027.

Rogowski, M., Aseeri, S., Keyes, D., & Dalcin, L. (2023). Mpi4py.futures: MPI-based asynchronous task execution for Python. *IEEE Transactions on Parallel and Distributed Systems*, *34*(2), 611–622. http://dx.doi.org/10.1109/TPDS.2022.3225481.

Torku, T. K., Khaliq, A. Q. M., & Furati, K. M. (2021). Deep-data-driven neural networks for COVID-19 vaccine efficacy. *Epidemiologia*, *2*(4), 564–586. http://dx.doi.org/10.3390/epidemiologia2040039, Retrieved from https://www.mdpi.com/2673-3986/2/4/39.

Verma, Y. (2021). A guide to parallel and distributed deep learning for beginners. Retrieved from https://analyticsindiamag.com/a-guide-to-parallel-and-distributed-deep-learning-for-beginners/.

What is high performance computing? - high-performance computing news analysis. (2015). Retrieved from insidehpc.com/hpc-basic-training/what-is-hpc/.

What is high-performance computing (HPC), Retrieved from https://www.oracle.com/middleeast/cloud/hpc/what-is-hpc/.

Zhang, C., Zhang, X., & Jiang, M. (2021). Research on parallel detection technology of remote sensing object based on deep learning. In *2021 4th international conference on intelligent autonomous systems (ICoIAS)* (pp. 29–32). http://dx.doi.org/10.1109/ICoIAS53694.2021.00013.

Zhao, Y. (2023). Complete guide to RNN, LSTM, and bidirectional LSTM. Retrieved from https://dagshub.com/blog/rnn-lstm-bidirectional-lstm/.