PROCESSOR PARTITIONING FOR ANOMALY DETECTION USING MACHINE LEARNING

Student Name: Ruiyang Wang

Bronco ID: 016087383

Faculty advisor: John Korah

Course: CS 4620

Semester: Spring 2025

Date of Submission: May 21, 2025

Abstract

In the field of cybersecurity, detecting network anomalies is a critical yet computationally intensive task. Traditional intrusion detection systems often struggle to scale with the increasing volume and complexity of network data. This research presents a custom-built deep learning framework that applies processor-level partition to improve performance in anomaly detection tasks. The approach specifically focuses on vertical partition of input features across multiple processors using the Message Passing Interface (MPI), enable with parallel execution of forward pass in the neural network.

The architecture was developed from scratch in Python without high-level libraries, which allows full control over layer structure and weight updates. To validate correctness, the output of the custom model was compared against a PyTorch implementation, and the results showed a low mean absolute error. Experimental evaluations were conducted using the CIC-IDS 2017 dataset, a standard benchmark for intrusion detection systems.

Results show a significant reduction in computation time compared to serial execution. The most notable speedup was observed when running on 16-32 processors. This work lays the foundation for scalable, distributed anomaly detection models and offers insights into future implementations involving autoencoders and high-performance computing clusters.

Introduction and Problem Statement

In today's fast-paced digital landscape, timely anomaly detection is essential for identifying both known and emerging threats, thereby safeguarding the integrity and resilience of cyber systems. Traditional security systems are often insufficient for detecting sophisticated threats, especially in large-scale network environments. This challenge has motivated the rise of machine learning–based anomaly detection systems, which aim to identify unusual patterns that may indicate malicious activity.

Deep neural networks (DNNs) have shown strong performance in this domain, but they are computationally intensive—especially when applied to high-dimensional datasets. This computational burden presents a significant obstacle for real-time cybersecurity applications, where rapid and responsive detection is critical.

The core problem addressed by this project is the high computational cost of training and executing deep learning models for anomaly detection. To overcome this, the project explores processor partitioning strategies, specifically vertical partitioning using the Message Passing Interface (MPI), to accelerate the forward propagation process. By distributing input features across multiple processors, the model can compute outputs in parallel, significantly reducing runtime while preserving output fidelity.

This system is built entirely from the ground up without the use of high-level framework, which grants full control over the network architecture, data flow, and inter-processor communication.

Literature Review

I. Anomaly Detection

Anomaly detection, also known as outlier detection, refers to the process of identifying instances in a dataset that deviates significantly from the norm. These instances often signal critical issues such as security breaches, fraud, or system failures. Chandola et al. define anomaly detection as "determine which instances stand out as being dissimilar to all others" in a data-driven fashion [1]. As data volumes grow to gigabyte or terabyte scales, conventional approaches also become computationally infeasible for real-time detection. In contrast, deep anomaly detection (DAD) techniques leverage deep neural networks to automatically learn hierarchical, discriminative features directly from raw input data [2].

Intrusion Detection Systems (IDS) are a critical component of modern cybersecurity, tasked with monitoring network traffic, detecting malicious activities, and issuing alerts to prevent potential breaches. However, traditional IDS approaches often struggle with limitations in real-time responsiveness, adaptability to evolving threats, and scalability in high-volume environments. While this project does not yet implement autoencoders, their potential remains a key motivation for future work.

II. CIC – IDS 2017 Dataset

The CIC-IDS 2017 dataset, developed by the Canadian Institute for Cybersecurity, is a widely adopted benchmark for evaluating intrusion detection systems (IDS). It was created to address the limitations of outdated or synthetic datasets by providing realistic network traffic that mirrors modern enterprise environments. The dataset includes both benign and malicious traffic, encompassing a wide variety of up-to-date attack types such as brute force, botnets, DDoS, web attacks, and infiltration scenarios. According to the authors of CIC-IDS2017 [3], the dataset is distributed across eight separate files and contains five days of both normal and attack traffic data, collected by the Canadian Institute for Cybersecurity. Table 1 provides an overview of the dataset's structure and the specific attack scenarios captured across the five-day period.

Table 1: Description of files for CIC-IDS 2017 dataset

Name of Files	**Day Activity**	**Attacks Found**
Monday-WorkingHours.pcap_ISCX.csv	Monday	Benign
Tuesday-WorkingHours.pcap_ISCX.csv	Tuesday	Benign FTP -Patator, SSH-Patator
Wednesday-workingHours.pcap_ISCX.csv	Wednesday	Benign, DoS GoldenEye, DoS Hulk DoS Slowhttptest, DoS slowloris, Heartbleed
Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv	Thursday	Benign, Web Attack – Brute Force, Web Attack – Sql Injection, Web Attack – XSS

Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv	Thursday	Benign, Infiltration
Friday-WorkingHours-Morning.pcap_ISCX.csv	Friday	Benign, Bot
Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv	Friday	Benign, PortScan
Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv	Friday	Benign, DDoS

It is important to note that an effective intrusion detection system (IDS) should be capable of detecting a wide range of attack types. To achieve this, the traffic data from all five days should be combined into a single, comprehensive dataset [4]. In this project, the individual CSV files listed in Table 1 were merged to create a unified dataset, ensuring that the simulation is based on diverse traffic patterns representative of real-world conditions. The resulting combined dataset contains 5,950,088 samples and 85 features, providing a comprehensive foundation for training and evaluation. Although this project has not yet implemented a complete anomaly detection model, the CIC-IDS2017 dataset was utilized to simulate data flow and evaluate the runtime performance of the proposed neural network architecture. Simulating data flow can aid in verifying model correctness and evaluating performance expectations before full implementation, making it a valuable step in system validation and design [5].

Pros:

- Provides modern, diverse attack types with labeled ground truth.
- Enables robust evaluation of machine learning models under realistic conditions.
- Highly cited and accepted in academic research

Cons:

- Scattered Presence
- Huge Volume of Data
- Missing Values
- High class imbalance [4]

Again, for now this project leverages the CIC-IDS 2017 dataset not for anomaly detection directly, but as a foundation for simulating realistic input during performance testing. Initially, smaller slices of the dataset were used to validate the basic functionality of the custom network architecture.

III. Autoencoder

An autoencoder is a type of feedforward neural network designed to reconstruct its input at the output layer by learning a compressed internal representation of the data. It consists of two

primary components: an encoder, which maps the input x to a lower-dimensional latent space $h = f(x)$ and a decoder, which attempts to reconstruct the original input from the latent code via $r = g(h)$ [6]. This architecture is presented in **figure 1**.

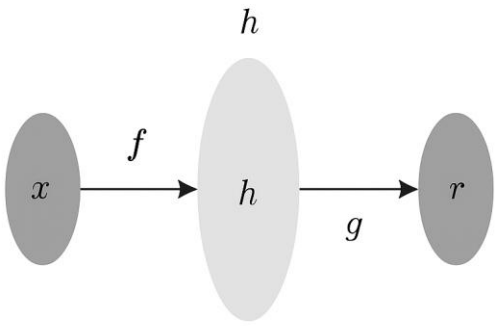

Figure 1

Basic structure of an autoencoder consisting of an encoder that maps input data to a latent representation, and a decoder that reconstructs the original input.

Unlike a simple identity mapping, the objective of an autoencoder is not to replicate the input exactly. Instead, constraints such as reduced dimensionality or regularization are imposed to prevent the network from learning a trivial identity function. These limitations compel the model to learn the most informative features of the input, effectively extracting latent representations that capture the underlying structure of the data. Such representations are particularly useful for anomaly detection, as deviations from the learned normal patterns result in higher reconstruction error, thereby flagging potential anomalies.

Pros:

- Unsupervised learning
  - Autoencoders don't require labeled anomaly data, which is often scarce in cybersecurity [7].
- Automatic feature extraction
  - Autoencoders learn latent representations directly from raw input. They reduce the need for manual feature engineering [2].
- Effective on high dimensional data
  - Autoencoders can model complex data distributions [8].
- Flexible architecture
  - Can be extended to different autoencoders, such as variational autoencoders (AVE), Conventional autoencoders, or sparse autoencoders [6, 10].

Cons:

- Sensitive to training data quality.

    - If trained on noisy or mislabeled data, the model may learn incorrect representations [2].
- Limited performance on mixed-type data.
    - Autoencoders often require numeric input and may struggle with categorical or hybrid features if not preprocessed carefully [9].
- Lack of explainability
    - The learned latent features are often difficult to interpret, may make the model a black box.

While this project does not yet implement an autoencoder, the custom neural network architecture was intentionally designed to support future integration of autoencoder-based anomaly detection. Many existing works focus solely on the detection accuracy of autoencoders without addressing the computational limitations in real-time or resource-constrained environments. This project complements such research by targeting the scalability and runtime performance of the model architecture itself.

IV. Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standardized and portable message-passing protocol used in parallel computing to enable communication between distributed processes. MPI originally designed for high-performance computing (HPC) applications, then it has since found use in scalable machine learning systems, particularly where neural network workloads are distributed across CPUs or nodes in a cluster. MPI enables multiple processes to coordinate by passing messages asynchronously or synchronously [11].

In deep learning, MPI has primarily been used for data-parallel training, where training samples are split across processes, and gradients are aggregated during backpropagation [12]. In contrast, model-parallel strategies such as vertical partitioning, horizontal partitioning, and block partitioning distribute different parts of the model or input data across processors.

Pros:

- Scalable
    - Enables parallel computation across many processors, it can improve runtime performance [11].
- Hardware-independent:
    - MPI can work efficiently on CPU clusters, without requiring GPUs.
- Fine-grained control
    - MPI can give developers precise control over data flow and communication.

Cons:

- Steep learning curve:
    - Requires manual management of communication, data distribution, and synchronization [13].
- Error-prone
    - Debugging MPI applications can be difficult.
- Limited adoption

- Most MPI applications focus on data parallelism, not feature-level splitting.

Most existing works that use MPI in deep learning focus on data-parallel training involving full backpropagation [12]. However, the project applies MPI to simulate vertical partitioning of input features in the forward propagation phase only. Although backpropagation has not yet been implemented in the MPI-based model, this project contributes an early-stage exploration of runtime behavior in vertically partitioned architectures. It addresses a gap in the literature by testing MPI's viability for high-dimensional input scenarios.

V. Data Partitioning ----Vertical Partitioning

Data partitioning is a key strategy in parallelizing neural network workloads to improve computational efficiency and scalability. The three most commonly used partitioning schemes in deep learning are sample parallelism, model parallelism, and hybrid partitioning [14]. Vertical partitioning is one of the model parallelism strategies that distributes input features (columns) across multiple processors, rather than distribute training samples (rows) as in data parallelism. Each processor is assigned a subset of features and is responsible for computing partial contributions to the output activations during forward propagation. This strategy contrasts with horizontal partitioning, which splits data samples, and block partitioning, which applies a two-dimensional split over both samples and features [15].

Pros:

- Scalability
  - Vertical Partitioning can reduce memory and compute overhead per processor when feature counts are large [16].
- Support early architectural simulation
  - Allows testing performance characteristics before full training or backpropagation is implemented.
- Simple partitioning
  - Vertical partitioning does not require complex clustering algorithms or dynamic partition restructuring [17].
- Easier to calculate runtime cost
  - Each processor handles a fixed subset of input features; it becomes easier to measure and predict the runtime cost of forward propagation across layers [18].

Cons:

- Synchronization overhead
  - Partial results must be gathered and combined across processes, introduce latency [19].
- Limited framework support
  - Popular deep learning frameworks do not natively support vertical partitioning.
- Sensitive to missing feature blocks
  - Missing or unavailable features can significantly degrade model performance and limit real-world applicability [20].

This project contributes to the gap by applying vertical partitioning at the feature level in a custom-built neural network architecture. Unlike property-table systems or data-parallel approaches, this implementation uses a fixed and deterministic partitioning strategy without requiring clustering algorithms or adaptive schema management [17]. By focusing on forward propagation only and simulating realistic input using the CIC-IDS 2017 dataset, this project demonstrates the runtime viability of vertical partitioning in CPU-based environments.

Research Objectives and Scope

The original objectives of this project:

- To implement an autoencoder neural network from scratch in Python and train it using the KDD Cup 1999 and CIC-IDS 2017 datasets.
- To parallelize the autoencoder using MPI and evaluate performance under different processor partitioning strategies: vertical, horizontal, and block.
- If time permitted, to explore CUDA-based parallelization as an alternative to MPI and compare performance outcomes.

Scope of Completed Work:

A simple feedforward neural network was developed from scratch in Python, and the forward propagation phase was parallelized using vertical partitioning. In this approach, the input features were divided across multiple MPI processes, with each process responsible for computing a partial contribution to the weighed sum in the network's forward pass. This setup was tested using a slice (10 features and 800+ rows) of the CIC-IDS 2017 dataset to simulate realistic data flow and evaluate runtime behavior under high-dimensional input conditions.

Changes from the original plan:

Several adjustments were made to the original objectives based on project complexity, time constraints, and resource availability:

- The autoencoder architecture and training components were not implemented in this phase. The project instead emphasized testing architectural scalability and runtime performance using forward pass simulation.
- The KDD Cup 1999 dataset was not used, as focus shifted entirely to the more modern and feature-rich CIC-IDS 2017 dataset.
- Only vertical partitioning was implemented as part of this individual contribution. Horizontal and block partitioning are being developed by teammates as part of the broader group project. At this stage, no direct performance comparisons between the different partitioning strategies have been conducted.
- CUDA parallelization was deprioritized to concentrate on MPI-based evaluation due to the complexity of GPU integration and the project's emphasis on CPU-based testing.

System Design and Methodology

This project implements a parallel deep neural network (DNN) from scratch using Python and MPI (Message Passing Interface). The goal is to evaluate the performance benefits of vertical partitioning, where input features are distributed across multiple processors to parallelize the forward propagation step. This approach targets scalability and runtime efficiency in high-dimensional input scenarios, such as those encountered in cybersecurity datasets.

The system is composed of modular Python files defining Activations, Layer, and Model structures.

Unlike common frameworks such as TensorFlow or PyTorch, this architecture was developed from scratch to provide full control over message-passing logic and network internals.

Tools:

Python: Used as the primary language for implementation.

Mpi4py: Python wrapper for MPI, used for process communication and synchronization.

Numpy: Used for numerical operations and matrix handling.

CIC – IDS 2017 dataset: Used to simulate real-world input traffic for forward propagation performance evaluation.

Vertical Partitioning Design:

The system uses vertical partitioning to distribute input features (columns of the input matrix) across multiple MPI processes. Each process receives the same set of input samples but only operates on a subset of the input features. Once each processor completes its portion of the computation (i.e., the partial weighted sums for each node), the results are gathered and combined to produce the full output of the forward pass.

Figure 2 illustrates the vertical partitioning architecture used in this project. Although this project has not yet implemented a trained autoencoder, the neural network architecture used in this simulation mirrors the structural layout of an autoencoder.

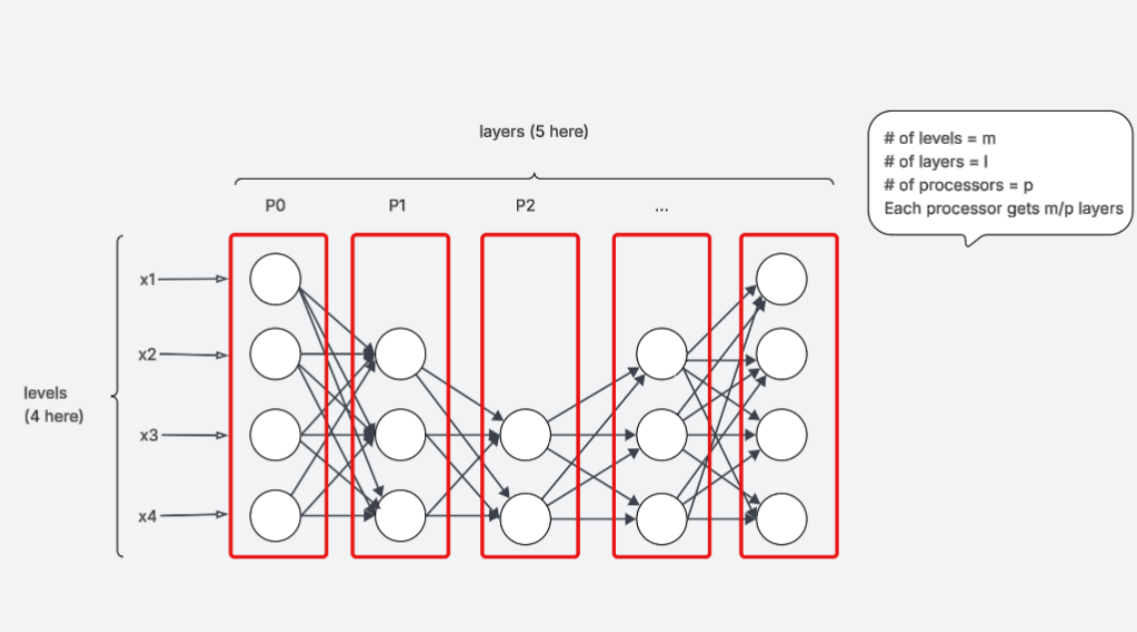

Figure 2. Vertical Partitioning Design

In the figure 2, each red box (P0, P1, P2…) represents a separate MPI process. All processors receive the same input samples (x1, x2, x3, x4), but each is assigned a distinct set of features (a vertical slice of the input matrix). These slices are processed independently, with each processor performing forward propagation on its subset. After local computation is completed, the partial outputs are aggregated to form the complete output.

This design matches the project's MPI implementation, where each process operates independently and contributes to the collective output in parallel.

Rationale for Design Decisions

- The high dimensionality of cybersecurity data (CIC-IDS 2017)
- The goal of enabling scalable, interpretable simulations before full model training is implemented.
- Building the network from scratch, rather than using a prebuilt framework. Enabled direct control over partitioning logic and communication strategy.

Implementation

Code structure and component:

- Models.py: Implements the core Model class, which constructs a multi-layer feedforward network using a configurable number of layers (depth) and neurons per layer (width).
- Layers.py: Defines the Layer. Each Layer includes logic for forward propagation, weight updates, and integration with activation functions.

- Activation.py: Contains activation function classes including ReLU and Sigmoid, both supporting forward activation and derivative computation, essential for future training extensions.
- verticalPartition.py: The primary driver script that launches the MPI-enabled parallel forward propagation. It performs feature-wise vertical partitioning, builds a local sub model on each MPI process, and gathers partial outputs for reconstruction.
- sharedParameter.py: Generates shared weights and biases using fixed random seeds to ensure consistency across MPI processes and PyTorch simulations.
- data_preprocessing.py: Preprocesses the CIC-IDS 2017 dataset by removing NaNs/infinities, normalizing features, and padding the feature vector to a fixed size (100 features).
- Compare_results.py and pyTorchTest_100x200.py: These are used to verify output correctness between the custom MPI implementation and a matching PyTorch simulation of the vertically partitioned model. The comparison includes mean absolute error measurement.

Technologies and platforms used:

- Programming Language: Python 3.10
- Parallel Framework: mpi4py
- Numerical Libraries: Numpy, matplotlib
- Deep Learning Framework for benchmarking: PyTorch
- Dataset: CIC-IDS 2017 (normalized and padded input)
- Execution Environment: CPU-based testing on local development machine

Major development challenges and solutions:

Building a Neural Network from Scratch

- Challenge: Implementing a functional multi-layer perceptron model without using high-level frameworks like PyTorch or TensorFlow.
- Solution: Create modular classes that support forward propagation and future extensibility for training. Reference the works done by the previous group.

MPI-Based Vertical Partitioning

- Challenge: Implementing vertical partitioning such that input features are split across processors while maintaining synchronization and correctness.
- Solution: Designed a balanced slicing algorithm to synchronize input features and gather partial outputs across processes. Each processor builds its own subnetwork to match its local feature dimensions.

Synchronizing Parameters Across MPI and PyTorch

- Challenge: Needed a way to fairly compare results between the custom MPI model and PyTorch.

- Solution: Used a shared seed (such as seed=55) and a utility script (sharedParameter.py) to generate identical weights and biases for both implementations.

Experimental Validation

Comparison with Existing Frameworks

To evaluate the correctness of the MPI-based vertically partitioned model, its output was compared against a functionally equivalent PyTorch implementation using the same dataset (CIC-IDS 2017), shared parameters, and partitioning scheme. The comparison used mean absolute error (MAE) between the final outputs of both models as the metric.

While the overall shape and structure of the outputs were consistent, a notable discrepancy was observed: the mean absolute error was approximately 0.29. This indicates a divergence in numerical output despite identical weight initialization and input data. The likely causes include subtle implementation differences, such as mismatches in weight alignment, floating-point precision issues during MPI communication, or missing activation layers in either model.

Runtime Performance

To evaluate the performance of the MPI-based vertical partitioning approach, forward propagation runtime was measured across varying numbers of processors (p = 4, 8, 16, 32, 64) and input sample sizes (n = 5, 10, 20, 40, 50, 100). As a baseline, the serial (non-parallelized) version of the model completed a forward pass in **0.102218** seconds.

As shown in Table.2 vertical partitioning with MPI resulted in substantial speedups. For small sample sizes (e.g., n = 5), runtimes dropped to 0.00047 seconds with 32 processors—a speedup of over 217× compared to the serial baseline. At larger batch sizes (n = 100), the best performance was observed at p = 32, with a runtime of 0.004299 seconds, representing a speedup of approximately 23.77×.

Table. 2 Runtime Performance

n	P = 4	P = 8	P = 16	P = 32	P = 64
5	0.001544	0.000726	0.000571	0.00047	0.00089
10	0.002719	0.001214	0.000794	0.000569	0.001577
20	0.003863	0.00196	0.001648	0.000968	0.004186
40	0.007387	0.005268	0.002282	0.001799	0.008772
50	0.009178	0.004339	0.002927	0.002723	0.005952
100	0.01533	0.007617	0.007182	0.004293	0.114629

Notably, performance degraded slightly at p = 64, where communication overhead began to offset computation time, particularly for larger inputs. For example, runtime at n = 100 rose to 0.114629 seconds, exceeding the serial baseline due to communication costs. This behavior

reflects a typical tradeoff in parallel computing: beyond a certain threshold, increasing processor count yields diminishing or negative returns due to increased synchronization and message-passing overhead.

These results confirm that vertical partitioning using MPI is highly effective for moderate to large processor counts (up to 32 in this test), especially when balancing the computational load across high-dimensional input features.

Speedup Evaluation Across Processor Counts and Network Depths

To better understand the scalability of the vertical partitioning approach, a speedup analysis was conducted by measuring forward propagation times across different processor counts (p = 4, 8, 16, 32, 64) and varying neural network depths (n = 5, 10, 20, 40, 50, 100 layers). As shown in **Figure 3.**

To evaluate the scalability of the MPI-based vertical partitioning approach, speedup was used as the primary metric. Speedup was calculated using the standard formula:

$$S(p) = \frac{T_{p=4}}{T_p}$$

Where $T_{p=4}$ is the runtime at 4 processors (used as the baseline) and $T_p$is the runtime at a given number of processors p. This metric quantifies how much faster the system runs as more processors are added.

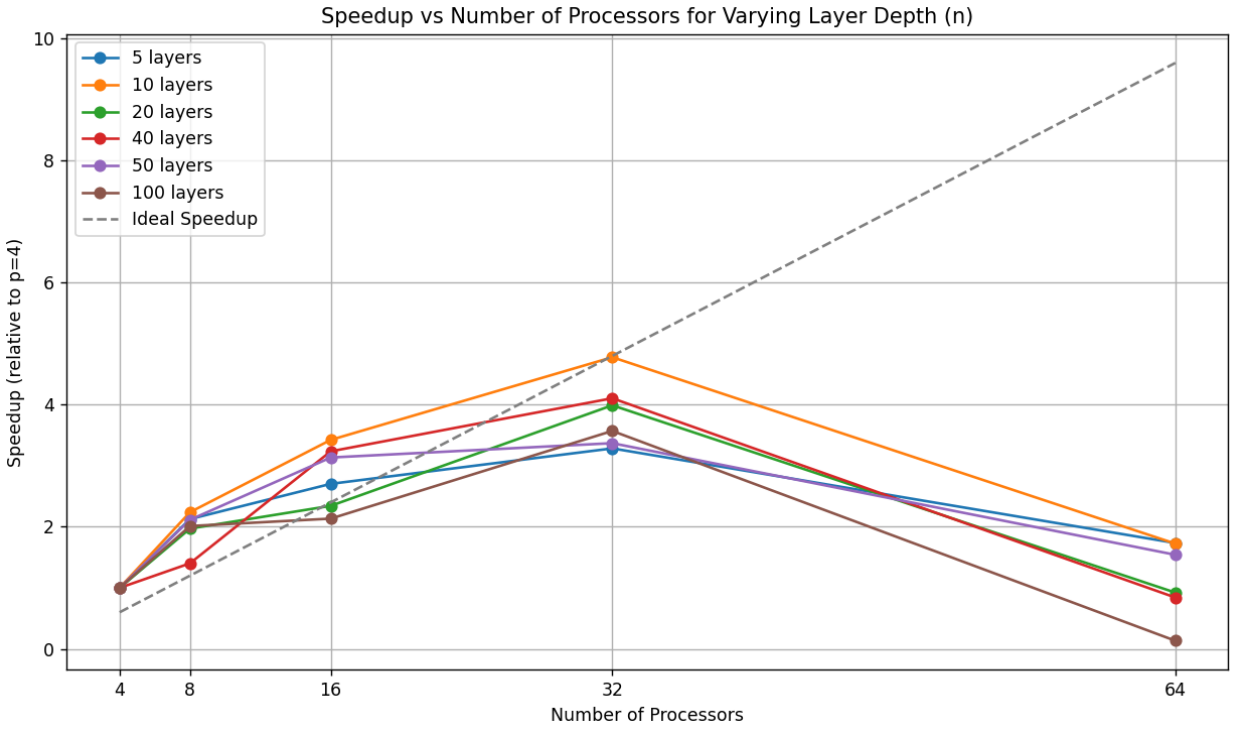

Figure 3. Speedup vs Number of Processors for Varying Layer Depth

Across all depths, the speedup curves exhibit a consistent pattern: performance increases as the number of processors grows, reaching a peak at p = 32, after which it begins to decline. This is expected behavior in parallel systems and aligns with Amdahl's Law [21], which states that the achievable speedup from parallelization is limited by the non-parallelizable portions of the program.

The initial gains in performance from p = 8 and p = 16 are significant because computation dominates communication. However, beyond p = 32, the parallel overhead increases, causing performance to deteriorate. Key factors contributing to this behavior include:

- Synchronization overhead between processes
- Increased communication cost during data gathering and broadcasting
- Fixed serial costs, such as data loading and MPI setup, which do not benefit from parallelism

At p = 32, the input features are already divided into small enough slices that additional processors add more communication than computation, resulting in diminishing returns. This inflection point represents a practical upper bound on processor scalability for this model and dataset configuration.

Timeline Review

Over the course of the project, I consistently dedicated approximately 3–4 hours per week to development, debugging, and testing. This steady weekly progress allowed me to contribute updates during each scheduled group meeting and ensure that the vertical partitioning implementation remained on track.

The project remained generally aligned with our planned timeline during the early phases of system design, preprocessing, and modular implementation. However, more time should have been allocated to debugging and validating model correctness, particularly in comparing MPI outputs to PyTorch baselines. This became a critical step later in the project and revealed discrepancies that required further investigation.

Despite these deviations, the vertical partitioning component reached a functional and testable state. Runtime benchmarking and speedup evaluation were completed, contributing meaningful insights into the parallel performance of the system.

Conclusions and Future Work

This project demonstrates the effectiveness of vertical partitioning in accelerating deep neural networks using MPI. By splitting input features across processors and parallelizing the forward pass, the system significantly reduced runtime compared to baseline serial execution. Runtime evaluations showed that speedup peaked with 32 processors before diminishing returns were observed due to parallel overhead, consistent with Amdahl's Law.

While the project did not focus on training or classification accuracy, the results highlight the practical benefits of vertical partitioning for forward-pass optimization. The implementation

successfully simulates realistic data flow using the CIC-IDS 2017 dataset and serves as a scalable foundation for future distributed neural network systems.

Limitations:

- The model only supports forward propagation; training is not yet implemented.
- Only vertical partitioning was completed; horizontal and block partitioning (assigned to teammates) remain untested in direct comparison.
- There is a measurable output discrepancy (0.29 mean absolute error) between the MPI implementation and the PyTorch baseline.

Future Work:

- Align custom and PyTorch models more precisely to reduce output differences and ensure numerical correctness.
- Compare vertical, horizontal, and block partitioning implementations to determine which strategy offers the best runtime performance and scalability.
- Extend the model to support training, enabling the system to perform actual anomaly detection using autoencoders and the CIC-IDS dataset.
- Deploy the system on an HPC cluster to test its large-scale performance across more processors and more complex datasets.

Bibliography

[1] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” ACM Computing Surveys, vol. 41, no. 3, pp. 1–58, 2009.

[2] R. Chalapathy and S. Chawla, “Deep learning for anomaly detection: A survey,” *arXiv preprint arXiv:1901.03407*, 2019.

[3] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorba-ni, “Toward Generating a New Intrusion Detection Dataset and Intru-sion Traffic Characterization”, 4th International Confer-ence on In-formation Systems Security and Privacy (ICISSP), Purtogal, Janu-ary 2018

[4] Panigrahi, R., & Borah, S. (2018). A detailed analysis of CICIDS2017 dataset for designing Intrusion Detection Systems. International Journal of Engineering & Technology.

[5] M. B. Blake and H. R. Choi, “Data-flow based model analysis of software systems,” NASA Technical Report, NASA/CR-2010-216709, 2010.

[6] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press

[7] M. Sakurada and T. Yairi, “Anomaly detection using autoencoders with nonlinear dimensionality reduction,” in Proceedings of the MLSDA 2014 Workshop on Machine Learning for Sensory Data Analysis, 2014.

[8] J. An and S. Cho, “Variational autoencoder based anomaly detection using reconstruction probability,” Special Lecture on IE, vol. 2, no. 1, 2015.

[9] B. Zong, Q. Song, M. R. Min, et al., “Deep autoencoding gaussian mixture model for unsupervised anomaly detection,” in ICLR, 2018.

[10] Z. Chen, C. K. Yeo, B. S. Lee and C. T. Lau, "Autoencoder-based network anomaly detection," 2018 Wireless Telecommunications Symposium (WTS), Phoenix, AZ, USA, 2018, pp. 1-5.

[11] W. Gropp, E. Lusk, and R. Thakur, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd ed. MIT Press, 1999.

[12] A. Sergeev and M. Del Balso, “Horovod: fast and easy distributed deep learning in TensorFlow,” arXiv preprint arXiv:1802.05799, 2018.

[13] Foster, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Reading, MA: Addison-Wesley, 1995.

[14] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” ACM Computing Surveys, vol. 52, no. 4, pp. 1–43, 2019.

[15] I. -K. Kim, J. Min, T. Lee, W. -J. Han and J. Park, "Block Partitioning Structure in the HEVC Standard," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, no. 12, pp. 1697-1706, Dec. 2012

[16] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou, “Vertical partitioning algorithms for database design,” ACM Transactions on Database Systems, vol. 9, no. 4, pp. 680–710, 1984.

[17] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, “Scalable Semantic Web Data Management Using Vertical Partitioning,” in Proc. of the 33rd International Conference on Very Large Data Bases (VLDB), Vienna, Austria, Sept. 2007, pp. 411–422.

[18] W. W. Chu and I. T. Ieong, "A transaction-based approach to vertical partitioning for relational database systems," in IEEE Transactions on Software Engineering, vol. 19, no. 8, pp. 804-812, Aug. 1993

[19] Ceballos, I., Sharma, V., Mugica, E., Singh, A., Roman, A., Vepakomma, P., & Raskar, R. (2020).

[20] Valdeira, S. Wang, and Y. Chi, “Vertical Federated Learning with Missing Features During Training and Inference,” arXiv preprint arXiv:2410.22564, 2024.

[21] John L. Gustafson. 1988. Reevaluating Amdahl's law. Commun. ACM 31, 5 (May 1988), 532–533.