Formulating Parallel Supervised Machine Learning Designs For Anomaly-Based Network Intrusion Detection in Resource Constrained Use Cases

Varun Joshi
Data Science
Georgia State University
Atlanta, United States
vjoshi6@student.gsu.edu

John Korah
Computer Science
California State Polytechnic University, Pomona
Pomona, United States
jkorah@cpp.edu

***Abstract*—One of the main problems with using supervised machine learning for anomaly-based Network Intrusion Detection (NID) in large cyber networks is the massive and dynamic data sets used in training and the computational overhead they induce in the training phase. In resource-constrained situations, there is a lack of powerful machines that would be traditionally used to deal with a high computational load. We focus on parallel processing designs that work in such resource-constrained use cases by lowering the computational overhead of supervised machine learning for anomaly-based NID, specifically for Mini-batch gradient descent. We avoid the black-box approach of traditional parallelization frameworks and allow the user to maximize the utilization of their scarce computational resources by granting greater control over parallelization while leveraging key aspects of the optimization algorithms being implemented. To demonstrate this, we implemented initial data and model parallel based designs, using the Compute Unified Device Architecture (CUDA) and Message Passing Interface (MPI) libraries, aimed at maximizing the use of a limited number of CPUs and GPUs. We conducted an initial comparative performance study using a large real-world network intrusion dataset called the KDD cup 1999; our results demonstrate up to 8.5 times more epochs per second using just 1 GPU (4000 threads) and up to 37 times faster convergence using just 1 compute node (7 cores) when compared to a serial approach.**

Keywords—parallelization, gradient descent, anomaly detection, network intrusion detection, resource constraints

I. INTRODUCTION

Cybersecurity is one of the most important modern-day challenges due to the ever-growing role of technology in our lives. In 2020, global losses from cybercrime totaled over 1 trillion dollars, which was a substantial increase from 2018 (around 500 billion dollars) [13]. Detecting intrusions within network settings is an important aspect of cybersecurity, since much of organizational/personal data is stored online and could potentially be used by cyber attackers for malicious purposes. Cybersecurity issues are especially challenging in resource constrained scenarios, where resources to fight such threats are limited [21]. Network Intrusion Detection (NID) is an attempt to detect malicious behavior within network traffic to prevent malicious users from gaining access to confidential information and is based on the assumption that malicious users will act in a way that is different from non-malicious users [1].

There are many strategies to detect network intrusions, but one of the most popular is anomaly-based NID. Anomaly-based NID is a strategy that detects network intrusions based on deviations from normal network behavior [2]. Anomaly-based NID is a notoriously difficult problem, but Supervised Machine Learning (SML) has been shown to be an effective methodology for anomaly-based NID [3, 26]. This is due to the fast-testing phases of machine learning algorithms and how well SML has been shown to perform for an anomaly-based NID approach [3, 26]. However, there are other challenges related to using SML for NID.

With SML, one of the biggest computational challenge is the long training phases and the need to utilize potentially massive and/or real time/near-real time training data. NID, unlike most other SML problems, requires frequent retraining to keep up with new vulnerabilities or new methods of intrusion [3, 4]. Not only must SML models be frequently retrained, but must do so using datasets that can be multiple terabytes or petabytes in size [5]. Retraining models with this much data would require exceptionally long and/or require expensive computing resources. Not only can models get outdated quickly, but without this re-training, zero-day attacks, which can be some of the most detrimental types of attacks, are much harder to detect [6].

Researchers may face limited resources (small companies, independent business owners, etc.). Such resource constrained situations make NID using SML a difficult solution to implement using existing computational frameworks for machine learning. Current approaches that focus on speeding up training time (such as Keras [22] and TensorFlow [18]) are not ideal for users with limited computational resources due to two main factors: lack of user input in parallelization and lack of optimization for specific learning algorithms. These popular approaches often leave the parallelization as a black box that generalizes to several different SML algorithms. The lack of user input and generalizability of the parallelization approaches oftentimes can underutilize the limited resources in situations where computational power is scarce.

We showcase two parallel processing designs that represent an initial effort towards providing solutions for the general problem of formulating parallelization techniques for SML. We

2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS) | 978-1-6654-7180-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/MASS56207.2022.00117

2155-6814/22/$31.00 ©2022 IEEE
DOI 10.1109/MASS56207.2022.00117

evaluate these two initial designs, a distributed memory-based design and a hybrid distributed/shared memory design. We formulate a parallelization scheme that is specific to Mini-batch Gradient Descent (MBGD) and can be used for Stochastic Gradient Descent (SGD). This allows the design to take advantage of the unique properties of MBGD to maximize the usage of limited resources. Our approach provides greater control to users by allowing them to specify the number of CPU cores and number of GPU threads to be used, allowing for the utilization of heterogenous computing resources (distributed and shared memory architectures) and allowing for scaling over available computational resources.

Our distributed memory design uses an architecture based on a tree topology parallelization strategy. Unlike common approaches to parallelizing SML that typically induce heavy bottlenecks like MapReduce or a parameter server approach [18, 19, 20, 22], a tree topology design is able to scale much better due to highly parallelized communication. Our hybrid design utilizes both MPI and CUDA, using MPI to perform data parallelism on the CPU and using CUDA to perform model parallelism on the GPU.

This rest of the paper is divided up into 6 main sections: background, related works, methodology, performance analysis, and conclusion/future work. The background covers the mathematics behind gradient descent and some key parallelization concepts. The related works covers relevant parallelization methods that have attempted to parallelize gradient descent and how our approach differs. The methodology explains our models and our initial hypothesis. The performance analysis goes over our experimentation and discusses our results. The conclusion and future work addresses our contributions and our future plans to expand our initial models.

II. Background

The SML algorithm that is parallelized in this paper is Mini-batch Gradient Descent (MBGD), which is a variation of gradient descent. Gradient descent is a key algorithm used for optimizing parameters in the training phase of SML.

MBGD (1) is an iterative process that starts with a group of parameters, or weights, that are used to make predictions. These weights are then updated using the gradient of a loss function that measures how well the model is performing. During each iteration, the loss is calculated for a number of given values (a mini-batch) in the training dataset (x and y) and the gradient of the loss function with respect to a weight θ is taken. θ is then subtracted by a learning rate, φ, times the average gradient of the loss function. MBGD is a popular variation of batch gradient descent [8] which uses the average gradient calculated across the entire dataset rather than just the average gradient from a subset of values. Stochastic Gradient Descent (SGD), is simply MBGD except with a batch size of one.

$$\theta = \theta - \varphi \nabla_{\theta} J(\theta; x^{(i+n)}; y^{(i+n)}) \quad (1)$$

To test our parallelization methods, we used logistic regression on the KDD Cup 99 dataset to perform binary classification. We trained our models to determine whether certain network activity was malicious or benign.

In our work, we focused on algorithm designs that leveraged two key parallel processing approaches for the training phase of SML: data parallelism and model parallelism. With data parallelism, the training data is split up across nodes and each node gets a copy of the model parameters. Each node then trains their parameters on their subset of data. The parameters from each node are then combined into one model [7]. Whereas in model parallelism, the model parameters are distributed across nodes and trained on copies of the data. These parameters are combined at the end to form a unified model [7].

We experimented on both shared and distributed memory architectures. In shared memory architecture each worker has access to a global memory location, with communication between processors being done in the form of read and write operations [9]. On the other hand, distributed memory architecture gives each processor its own memory and has processors communicate by passing messages [9]. For the implementation on shared memory architecture, we used the Compute Unified Device Architecture (CUDA) libraries [10]. We specifically used the python implementation, Numba, for our experimentation [10]. For distributed memory architecture, we used the Python implementation of the Message Passing Interface (MPI) library [11].

III. Related Works

In previous work, authors have attempted to parallelize machine learning processes [7, 18, 19, 20, 22] and SGD in particular [14, 15, 16, 17, 23]. Papers typically focus on either CPUs [14, 15, 23, 25] or GPUs [16, 17, 24]. Most research using CPUs to parallelize SGD can typically be divided up into two basic sub-types: synchronous and asynchronous methods. With synchronous approaches to SGD, convergence is more deterministic, but communication costs more quickly add up at larger scales and this method scales poorly to heterogeneous environments. With asynchronous approaches, communication costs are minimized and more diverse environments can be used, but users can easily run the risk of stale gradients and poor convergence.

An approach that is able to utilize both the speed of asynchronous approaches and maintain convergence is HogWild! [14]. It is one of the most popular asynchronous approaches, primarily utilizing data parallelism. HogWild! is able to achieve a near linear speed up using non-blocking communication and a parameter server topology, but it is limited to ML applications that have sparse matrix updates. A more recent paper has been able to utilize a lock-free design similar to the one in HogWild!, but with higher speed ups and less gradient staleness [23].

One of the most impressive results from synchronous communication largely comes from Das et al. [15], where the authors were able to use hundreds of nodes to achieve

impressive speed ups, maintain convergence, and not lose too much speed to communication. A more recent approach, EventGraD [25], utilizes a synchronous parallelization scheme that has nodes only communicating when a certain condition is met; this approach maintains accuracy while drastically reducing communication.

While parallelization on CPUs can lead to impressive speed gains, GPUs are popular within the SML community. GPUs can certainly achieve great speed ups in SML, but there are drawbacks to using GPUs for parallelizing SGD specifically. As noted by Shi et al. [16], the need for data communication in SGD can be extremely costly when using GPUs and severely limit their overall potential. However, many have found that proper utilization of communication can render GPUs a much more powerful alternative when compared to CPUs [17].

A recent paper by Elahi et al. [24] looked at parallelizing SGD by using fractional calculus. Their approach allowed them to have more fine-grained gradient updates, which allowed for greater parallelism on a GPU. They were able to see consistent speed ups and high accuracy. However, their approach was aimed at recommender systems that used sparse, big data.

The most popular approaches to parallelizing GPUs can be found in frameworks such as Keras [22] and TensorFlow [18]. These approaches are able to achieve high levels of parallelization on a limited number of GPUs. However, they do not allow the user to control the number of cores or threads used on GPUs or CPUs and the parallelization method is largely a black box. The parallelization methods are also not necessarily optimized for a particular SML algorithm like SGD. These approaches also utilize a parameter server or MapReduce based approach and bottlenecks can limit the scalability of these methods [15, 19, 20].

Our approach attempts to alleviate some of these issues through utilization of a tree topology to reduce the bottlenecks in parameter server and MapReduce approaches. We give the user direct control in the number of cores or threads used. We also make an algorithm specific parallelization scheme to maximize speed up. Finally, we present a hybrid design that utilizes both distributed and shared memory to take advantage of the strengths and weaknesses of both and allow for scaling over heterogeneous computational resources.

IV. Methodology

We have two designs: the first is a distributed memory design that focuses on maximizing CPU usage (which will be referred to as parallel design 1) and the other is a hybrid distributed/shared memory design that focuses on maximizing GPU usage (this will be referred to as parallel design 2).

Parallel design 1 focuses solely on CPUs since, in cases with limited resources, there may not be access to a GPU. Not only this, but CPUs generally consume less power than GPUs, which makes them a more valuable asset in situations with limited resources. For the CPU, a tree topology design was chosen to limit communication bottlenecks and increase scalability to a higher number of devices in cases without resource constraints.

Our hybrid design was chosen due to the fact that, if GPUs are available to be used, they are a powerful resource for parallelizing MBGD [17]. Here, we utilize CPUs and GPUs in tandem to take advantage of their respective strengths and weaknesses. We utilize GPUs to do most of the calculations involved in updating model parameters due to their high computing power, and utilize CPUs to combine the parameters efficiently due to the GPU's weaknesses in communication [16]. We also allow the user to control the number of threads and cores used in both designs, avoiding the black-box approach that is frequently found in other parallelization methods. User control is essential here since the program itself cannot make any assumptions about which resources are available (eg, the program cannot simply select the max available threads, since the scarce computational resources in a limited-resource scenario will likely not be just used for NID). However, a default number of threads/cores could be easily implemented to account for less experienced users.

A. *Parallel Design 1 (Distributed Memory)*

Parallel design 1 (Fig. 1) utilizes a tree-based topology to allow for overlapped communication during training and to reduce the bottleneck that comes from approaches like parameter server. The way this design works is that every node gets a subset of the training data and a copy of the model parameters (data parallelism). Each node then runs asynchronously for a user-specified number of epochs, at which point the nodes each send their parameters to the node above them. The nodes that receive these parameters must average them with their own parameters and send them up to the node above them. This process keeps repeating until the server node receives the parameters from the two nodes below it. The server node then averages its own parameters with the ones it received and broadcasts these averaged parameters back down the tree. This process keeps repeating until a node's weights reach convergence. Once a node converges, training is considered to be complete.

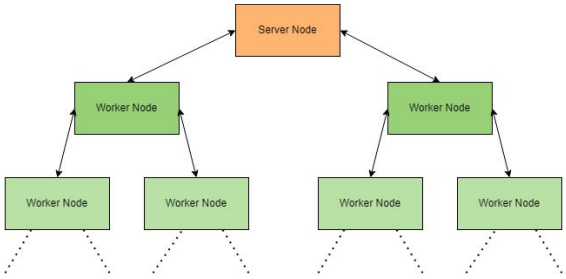

Fig. 1. Parallel Design 1

B. *Parallel Design 2 (Hybrid Distributed-Shared Memory)*

Our parallel design 2 (Fig. 2) focuses on utilizing the maximum number of threads possible in a GPU. The user sets the number of threads to utilize, and the design uses as many of those threads as possible to parallelize training.

The way that this design works is that each CPU core will get a partition of the data (data parallelism) and will send copies of the data to various partitions on the GPU (GPU blocks). These

partitions will each have one thread for each parameter there is to be trained. Each partition will then train each parameter in parallel (model parallelism) and send the trained parameters back to the CPU once they are done training. The CPU then combines all the parameters from the GPU, forming the final trained model.

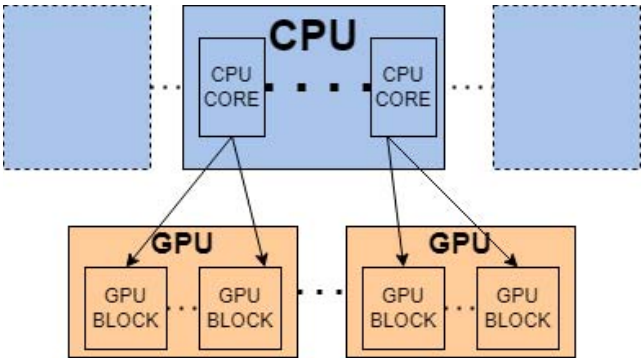

Fig. 2. Parallel Design 2

We hypothesize that parallel design 2 would have faster epochs per second due to the generally higher computational power of the GPUs. However, parallel design 2 not only has high levels of data parallelism and model parallelism, but also does so fully asynchronously. We suspected that while this high degree of parallelism and lack of synchronicity would allow for faster computation, it may lead to stale gradients and weight updates.

We predicted the opposite for parallel design 1; we assumed the accuracy would be much higher due to the frequent synchronization, but the design would not complete training much faster than the serial implementation. Parallel Design 1 allows for frequent synchronization, which should reduce the staleness of the gradients. However, with increased communication costs, we expected the runtime would be slower overall. Not just this, but the CPU's computing power being generally lower than the GPUs made us predict that parallel design 1 would perform generally slower overall due to slower calculations during gradient descent.

In the next section, we will perform experimentation to analyze if our hypotheses for the models hold.

V. Performance Analysis

In this section, we will describe the experimental setup and provide analysis of the experimental results. We will cover the computational resources used, the experimental results, and perform analysis of the results. We generally see that both the designs can achieve higher epochs per second than the serial design, but parallel design 2 lacks in accuracy and parallel design 1 appears to reach a communication bottleneck.

A. Experimental Setup

Our testing was all done on the KDD Cup 99 dataset [12]. While this may be an older dataset, the focus of this paper is the parallel processing designs and computational efficiency rather than over real-life efficacy. Moreover, KDD is still a widely used dataset with a wide range of attacks. We utilized an 80-20 split for the training/testing data. For preprocessing, we dropped duplicates, one-hot encoded categorical features, and standardized and normalized numerical features.

The compute node used for our experiments was an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and the GPU accelerator was a Tesla P100-PCIE-16GB. Our testing was only done on a maximum of 2 CPUs for parallel design 1 and a maximum of 2 GPUs for parallel design 2. The compute nodes were connected on a 10 Gbe network. This was done to mimic the conditions of extreme resource constraints.

To test our hypothesis on how the models would perform, we used a few key performance metrics: epochs per second, time to convergence (parallel design 1 only; parallel design 2 ran for a pre-determined number of epochs), and cross entropy loss. We used the epochs per second as well as the time to convergence in lieu of the speedup of both models and we used cross entropy to measure the accuracy.

To determine convergence for Parallel Design 1, we had each core check if its weight updates had dropped below a certain threshold, 0.00003. Once any weight update drops below this threshold, we consider convergence to have occurred. We decided on this threshold through testing. The threshold for the serialized model was 0.000005. This was, again, decided on by testing focused on finding a threshold with the best cross entropy loss. In Parallel Design 1, the nodes would run asynchronously for 20 epochs and then synchronize during our experimentation. For parallel design 2, which involves GPU computing, we conducted experiments to appropriately tune the parameters in the classification algorithm, specifically, learning rate of the classification algorithm.

In the graphs (Fig. 4, 6), we did not calculate the epochs per second by looking at total epochs run across all devices, but rather epochs run in parallel. For example, if 2 nodes ran 800 epochs each in 10 seconds, we would calculate epochs per second as 800/10 rather than 1600/10.

B. Results and Analysis

We looked at the epochs per second to judge the performance of the design and the cross-entropy loss to make sure no accuracy was lost while decreasing training time. For parallel design 1, we also measured time to converge, since that design converged when weight updates were below a certain threshold. Parallel design 2 only runs for predetermined number of epochs. Therefore, the time it takes to run is proportional to the epochs per second it can achieve. For parallel design 1 the epochs per second is useful, but since the model also generally converges much faster, the parallel runtime displayed dramatically different results from the epochs per second.

For parallel design 2, we ran experiments with both one and two GPUs. Our design has each CPU core controlling one GPU, but in the future, we plan to allow one CPU core to control multiple GPUs. Here, as the number of threads increases, we can see a steady growth in the epochs per second of the design (Fig. 4). The maximum gain in epochs per second we see from both

one GPU and two GPUs is around 8.5 times using 4000 CUDA threads. Utilizing two GPUs also seems to work better at higher thread counts (we tested from 200-4000 threads), because more threads are available when utilizing 2 GPUs.

The biggest drawback to using this parallelization scheme is the low cross entropy loss. It hovers around 0.2 (Fig. 3), which is poor, especially for an NID application, where high precision is required. The serial design was able to obtain a significantly better accuracy.

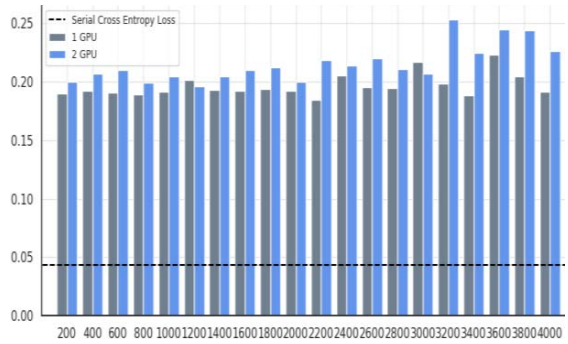

Fig. 3. Cross Entropy Loss Versus Number of CUDA Threads for Parallel Design 2

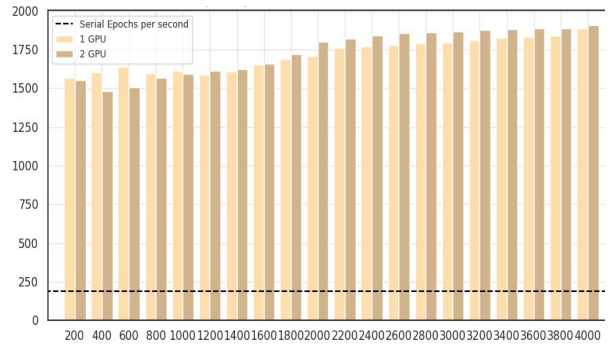

Fig. 4. Epochs per Second Versus Number of CUDA Threads for Parallel Design 2

Since CPUs are typically more accessible than GPUs due to their costs, parallel design 1's results indicate a possible solution for resource and cost constrained scenarios. Here, the seven and three core results are for one CPU, while the 15-core result is for two 2 CPUs. Even with just one CPU, this design (Fig. 6) rivals the speeds of parallel design 2 (Fig. 4) in terms of epochs per second and can achieve better accuracy (Figs. 3, 5), comparable to or better than the serialized design. Considering how much more accessible and cheaper CPUs are compared to GPUs, parallel design 1 is potentially a useful approach in scenarios where cheaper resources are preferred/available.

Additionally, the results in Fig. 7 indicate that parallel design 1 runs several magnitudes faster than the serial implementation when just measuring time to convergence. With just three cores, we see improvements by 8 times; with 7 cores, a 37 times improvement; with 15 cores, nearly a 40 times improvement. Parallel design 1 can get a similar, if not better, accuracy than the serial design in a much lower number of epochs, while being able to achieve higher epochs per second (Fig. 6).

However, a main limitation of parallel design 1 begins to emerge in Fig. 6 and 7: the rising communication costs limiting how much faster the model can converge. The decrease in runtime from three to seven cores is much higher than the decrease in runtime from seven to fifteen cores. This may introduce a problem at higher core numbers if this design is to be used at a greater scale. However, moving from 1 device to 2 devices may also be a limiting factor; communication within a single device has lower communication costs than communication between devices. Further testing would yield a better understanding of the scalability of this design.

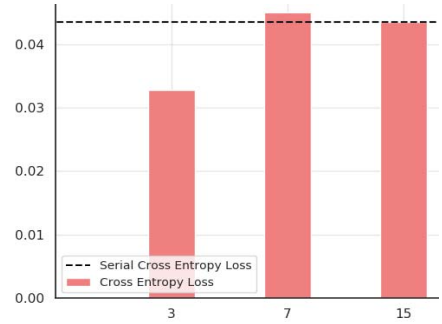

Fig. 5. Cross Entropy Loss Versus Number of Cores for Parallel Design 1

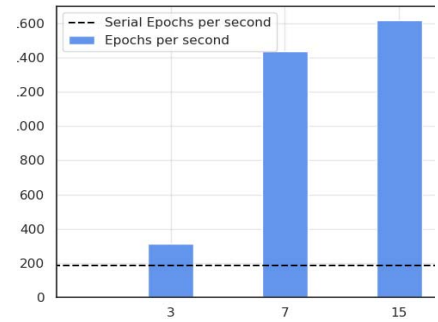

Fig. 6. Epochs per Second Versus Number of Cores for Parallel Design 1

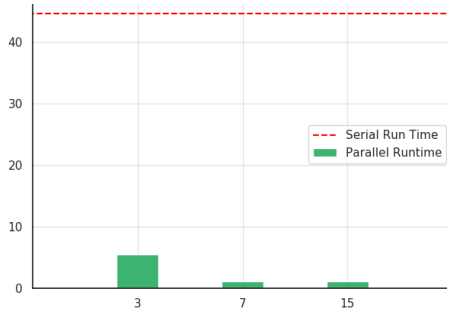

Fig. 7. Parallel Runtime (s) Versus Number of Cores for Parallel Design 1

C. Discussion

Regarding Parallel Design 2, our hypotheses largely hold. It was unable to maintain high accuracy likely due to the highly asynchronous and parallel approach but was able to achieve

much faster epochs per second. To address the lack of accuracy in Parallel Design 2, utilizing different optimization approaches and doing further hyperparameter optimization may prove fruitful. However, while the maximum epochs per second was greater for parallel design 2 when compared to parallel design 1, it was not greater by much.

Parallel design 1 had a much faster convergence and epochs per second than expected. Not only this, but the accuracy was also able to remain quite high. The main surprise about parallel design 1 was the appearance of a potential bottleneck at 15 cores. While the lack of performance improvement gained from 7 to 15 processors may be because the 15 processors are distributed over 2 devices while the 7 processors are only on 1 device, it is still a noteworthy result.

VI. Conclusion and Future Work

Our future work revolves around improving parallel design 2 to have a lower loss and improved epochs per second. We also want to see if we can allow for one CPU core to utilize the computing power of multiple GPUs rather than each CPU core controlling one GPU. We will also expand our performance study of both the initial designs for higher numbers of compute nodes and GPUs and see if these approaches are scalable.

We also plan to complete a hybrid MPI and CUDA design that uses a tree topology-based strategy. This approach aims to take advantage of both CPUs and GPUs, but also aims to leverage the tree topology's parallelized communication. We have done limited testing but have been able to see around 3x faster epochs per second using 2 GPUs thus far for the design.

Our work showcases initial designs that demonstrate faster convergence and epochs per second on limited resources, utilizing both CPUs and GPUs at their maximum potential. With this work, those with limited access to computational resources can utilize SML-based anomaly detection to avoid the immense destruction that can come from NID.

Acknowledgment

This work is supported in part by an NSF REU grant CNS 2050826.

References

[1] B. Mukherjee, L. T. Heberlein, and K. N. Levitt, "Network intrusion detection," *IEEE Network*, vol. 8, no. 3, pp. 26–41, 1994.

[2] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers & Security*, vol. 28, no. 1-2, pp. 18–28, 2009.

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

[4] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[5] D. M. Best, R. P. Hafen, B. K. Olsen, and W. A. Pike, "Atypical behavior identification in large-scale network traffic," *2011 IEEE Symposium on Large Data Analysis and Visualization*, 2011.

[6] Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, and F. Ahmad, "Network intrusion detection system: A systematic study of machine learning and Deep Learning Approaches," *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 1, 2020.

[7] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A Survey on Distributed Machine Learning," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 1–33, Mar. 2021, doi: 10.1145/3377454.

[8] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, doi: 10.48550/ARXIV.1609.04747.

[9] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 4, no. 2, pp. 63–71, 1996.

[10] L. Oden, "Lessons learned from comparing C-CUDA and Python-Numba for GPU-computing," *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020.

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

[12] "KDD Cup 1999 Data." http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html (accessed Jun. 16, 2022).

[13] "The Hidden Costs of Cybercrime." https://www.csis.org/analysis/hidden-costs-cybercrime (accessed Jun. 16, 2022).

[14] F. Niu, B. Recht, C. Re, and S. J. Wright, "HOGWILD! a lock-free approach to parallelizing stochastic gradient descent," *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*, pp. 693–701, 2011.

[15] D. Das et al., "Distributed Deep Learning Using Synchronous Stochastic Gradient Descent," 2016, doi: 10.48550/ARXIV.1602.06709

[16] S. Shi, Q. Wang, X. Chu, and B. Li, "A DAG model of synchronous stochastic gradient descent in distributed deep learning," *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018.

[17] R. Kaleem, S. Pai, and K. Pingali, "Stochastic gradient descent on gpus," *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, 2015.

[18] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016, doi: 10.48550/ARXIV.1603.04467.

[19] T. Chen et al., "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," 2015, doi: 10.48550/ARXIV.1512.01274.

[20] Y. Jia et al., "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, Orlando Florida USA, Nov. 2014, pp. 675–678. doi: 10.1145/2647868.2654889.

[21] D. Otieno, "Cyber security challenges: The Case of Developing Countries," presented at the Promoting Creativity, Innovation and Productivity for Sustainable Development, Nov. 2020. [Online].

[22] "Keras," *GitHub*. https://github.com/keras-team/keras

[23] K. Bäckström, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Consistent Lock-free Parallel Stochastic Gradient Descent for Fast and Stable Convergence," in 2021 *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2021, pp. 423–432. doi: 10.1109/IPDPS49936.2021.00051.

[24] F. Elahi, M. Fazlali, H. T. Malazi, and M. Elahi, "Parallel Fractional Stochastic Gradient Descent with Adaptive Learning for Recommender Systems," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–14, 2022, doi: 10.1109/TPDS.2022.3185212.

[25] S. Ghosh and V. Gupta, "EventGraD: Event-Triggered Communication in Parallel Stochastic Gradient Descent," in *2020 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*, Nov. 2020, pp. 1–8. doi: 10.1109/MLHPCAI4S51975.2020.00008.

[26] H. Tyagi and R. Kumar, "Attack and Anomaly Detection in IoT Networks Using Supervised Machine Learning Approaches," *RIA*, vol. 35, no. 1, pp. 11–21, Feb. 2021, doi: 10.18280/ria.350102.