Elastic deep learning through resilient collective operations

Jiali Li, George Bosilca, Aurelien Bouteiller, Bogdan Nicolae

Elastic deep learning through resilient collective communication

Jiali Li
University of Tennessee, Knoxville
Knoxville, Tennessee, United States
jli111@vols.utk.edu

George Bosilca
University of Tennessee, Knoxville
Knoxville, Tennessee, United States
bosilca@icl.utk.edu

Aurelien Bouteiller
University of Tennessee, Knoxville
Knoxville, Tennessee, United States
bouteill@icl.utk.edu

Bogdan Nicolae
Argonne National Laboratory
Lemont, Illinois, United States
bnicolae@anl.gov

ABSTRACT

A robust solution that incorporates fault tolerance and elastic scaling capabilities for distributed deep learning. Taking advantage of MPI resilient capabilities, aka. User-Level Failure Mitigation (ULFM), this novel approach promotes efficient and lightweight failure management and encourages smooth scaling in volatile computational settings. The proposed ULFM MPI-centered mechanism outperforms the only officially supported elastic learning framework, Elastic Horovod (using Gloo and NCCL), by a significant factor. These results reinforce the capability of MPI extension to deal with resiliency and promote ULFM as an effective technique for fault management, minimizing downtime, and thereby enhancing the overall performance of distributed applications, in particular elastic training in high-performance computing (HPC) environments and machine learning applications.

CCS CONCEPTS

• **Software Systems, Designing Software**;

KEYWORDS

Distributed deep learning, fault tolerance, elastic training, resilient collective communication

ACM Reference Format:
Jiali Li, George Bosilca, Aurelien Bouteiller, and Bogdan Nicolae. 2023. Elastic deep learning through resilient collective communication. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), 2023, Denver, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3624062.3626080

1 INTRODUCTION

As the workload and complexity of deep training continue to grow, there is a rising trend of scaling training processes across parallel resources. The use of accelerators, with their outstanding efficiency at performing variable-precision floating point operations, has unleashed the efficiency of deep learning training, and is a major part in the overall success that machine learning systems have enjoyed in recent years. When increasing the training scale on distributed systems, fault tolerance becomes a critical aspect to ensure a robust training progress. The larger the system, the more prone to failures, whether due to hardware malfunctions, network issues, or software errors. These failures can disrupt the training process, leading to data loss, wasted computation, and increased training time. In addition, deployments in the cloud may exhibit higher failure rates, even at modest scale, and may provide additional incentives for scaling up or down (e.g., expanding or shrinking) the training operation based on external factors, such as spot node pricing. To address these challenges, fault tolerance mechanisms are employed to ensure that deep learning training can withstand failures and continue running seamlessly. Fault tolerance techniques encompass various strategies, including checkpointing [20][33][39], replication [14], and error detection [15]. Checkpointing involves periodically saving the state of the training process, enabling it to resume from a recent checkpoint in the event of a failure. Replication involves creating multiple copies of the model or data, allowing the training to continue using an alternative copy in case of failure[32][7]. Error detection and recovery mechanisms focus on identifying and mitigating errors during the training process, ensuring that the model's integrity is preserved and training progresses uninterrupted[38].

In addition to fault tolerance, the elastic scaling of deep learning resources has become increasingly important. Deep learning models often require substantial computational resources, including processors, memory, and storage, to train effectively. Elastic scaling involves dynamically adjusting the available resources based on the workload and demand [28]. This flexibility enables efficient utilization of resources, ensuring that the deep learning system can scale up or down as needed, optimizing performance and cost efficiency. Elastic Horovod [3] has emerged as an advanced and widely embraced solution in enterprise clouds for effectively handling dynamic host changes during distributed data parallel training. This powerful framework leverages communication libraries such as Gloo [1] for CPU operations and NCCL [2] for GPU operations. With support for popular training engines like TensorFlow and PyTorch as backends, Elastic Horovod has established itself as a cutting-edge choice for managing training processes in a flexible and scalable manner within enterprise cloud environments. As the demand for large scale training on supercomputer-scale machines continues to rise, there is a need for alternative elastic training solutions that can effectively employ the type of high performance network capabilities and minimize overhead costs. In response to this demand, we present an innovative approach that harnesses the capabilities of User Level Fault Mitigation (ULFM) in Open MPI [24].

This paper is organized as follows. Section 2 provides some background about machine learning, elasticity in machine learning algorithms, and the ULFM communication library; Section 3 describes

SC-W 2023, November 12–17, 2023, Denver, CO, USA
2023. ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3626080

our design for fault tolerant machine learning; Section 4 presents an experimental evaluation of our approach while Section 5 presents prior and related works, before Section 6 concludes.

2 BACKGROUND

2.1 Fault tolerance in distributed training

In the context of deep learning, fault tolerance refers to the capability of continuing the training without significant disruption from unexpected events, errors or failures. Unexpected events include hardware issues, such as memory crashes, network disconnection, or software failures. The introduction of fault tolerance mechanism ensures the program can be recovered from such unexpected events without a substantial loss of the work already finished, as a consequence, prevents the need to re-execute the program from the scratch.

Typical fault tolerance technologies include checkpointing and shutdown-restart recovery[25][31][26]. In the checkpoint mechanism, the state of the training, including model parameters and state of optimizers, is regularly saved to stable storage. With this information, even if a fault occurs, the system can roll back to the latest saved checkpoint and resume the training from that state, aiming to reduce the overhead of repeating execution.

Recently, techniques like algorithm-based fault tolerance[30][38], which involves incorporating redundancy in the algorithms themselves to detect and correct errors[19]. As deep learning models and the infrastructures used to train them continue to scale, developing effective and efficient fault tolerance mechanisms becomes an increasingly important challenge. It's also important to balance the overhead of fault tolerance mechanisms with their benefits, as these mechanisms can add additional computational or storage costs. Current research is focused on devising fault tolerance techniques that are not only effective at handling faults but also efficient and scalable to keep up with the evolving scale of deep learning.

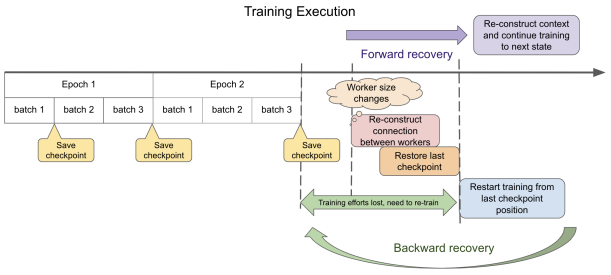

Figure 1: Backward recovery based on the checkpointed training state

2.2 Elastic training

Elastic training refers to the ability of a distributed deep learning training process to dynamically adjust to the changes in the availability of computational resources[13][36][35]. Essentially, it allows the system to add or remove nodes during the training process based on the resource availability and demand [27]. Elastic training has the potential to optimize resource utilization, minimize training time, and, in cloud-based pay-as-you-go scenarios, reduce the cost associated with computational resources. Key benefits aside, implementing elastic training in deep learning introduces new challenges. One of the main challenges is maintaining model consistency when nodes are added or removed. For instance, when new nodes are incorporated, the training process must ensure that the model's state is efficiently synchronized across all nodes, including the new ones. Similarly, when nodes are removed, whether due to failure or resource reallocation, the system needs to devise strategies to recover the lost computation or redistribute the workload amongst the remaining nodes. Furthermore, the training process must be resilient enough to handle any interruptions due to changes in the computing environment and resume from where it left off without any significant loss in training progress. Achieving such robustness often requires sophisticated checkpointing strategies and efficient communication protocols.

$$\begin{aligned} C_{\text{fault\_recovery}} &= C_{\text{checkpoint\_saving}} \times freq_{\text{saving}} \\ &+ Count_{\text{fault}} \times (C_{\text{checkpoint\_loading}} \\ &+ C_{\text{re-configuration}} + C_{\text{re-compute\_from\_checkpoint}} \\ &+ C_{\text{new\_worker\_init}}) \end{aligned} \tag{1}$$

The expense associated with recovery training from a checkpointed state, including the reconfiguration of the worker group, can be evaluated using the subsequent equation. In Eq. (1), the cost incurred while saving a checkpoint is contingent upon the technique employed and the frequency of saving. Similarly, the cost of loading is contingent on the location of the checkpoint file, and the recomputing cost of lost training progress since the saved state. Moreover, the expense of reconstructing the communication context is tied to the communication protocol and libraries used. If new workers join the training process following re-configuration, the cost associated with loading the training environment and re-establishing the training state across all workers needs to be taken into account. The cost of recomputation has an inverse relationship with the total cost of saving checkpoints. In other words, a shorter interval between checkpoints results in a reduced cost for recomputation, but an increase in the total cost of saving these checkpoints.

2.3 ULFM

The User-Level Failure Mitigation (ULFM) specification is an extension of the MPI standard that enables the continued operation of MPI programs across failures. Implementations of ULFM are available in both major open source implementations of the MPI standard (MPITCH [11] and Open MPI [24] from which most vendor-specific MPI libraries are derived.

Unlike legacy MPI which would abort the whole program on the first failure, ULFM programs can set up the MPI library so that it reports relevant errors when a process fails, call new MPI procedures to interrupt the ongoing flow of complex communication schemes, and call new procedures to create sane communicators, expunged from failed processes, to restore the full capability of performing high performance, collective communication. In ULFM errors are typically reported in a per-operation basis, and the meaning of an

error indicates that the operation did not achieve the desired semantic at the local rank. This relaxed semantic is key for both high performance fault-free communication, but also to enable a flexible recovery strategy that just keeps going with existing non-failed processes; a feature we will take advantage of when designing the recovery algorithm within Horovod.

More broadly, there is a rich literature documenting the successes of the HPC community in deploying fault tolerance using ULFM. It has been employed to support resilience features in programming languages [17, 18], resilient databases [21], checkpoint-restart frameworks [10], resilience frameworks [34], as well as purely algorithmic approaches [8, 12]. Some preliminary work has considered the adequacy of the ULFM constructs to support machine learning types of workloads [9]. This work will leverage on these established best practices for fault-tolerance patterns, but will also extend to new original patterns that enable the mixed use of different communication libraries in a resilient, yet highly efficient manner. Additional works using ULFM that are closely related to our effort are described further in Section 5.

3 DESIGN

3.1 Resilient collective communication

In conventional MPI, when a single process fails, it results in the termination of all remaining processes as MPI lacks fault-tolerant capabilities for handling unexpected events like process failures. Consequently, once an unexpected event occurs in a process, all participating processes need to conclude the current operation and restart the entire execution. However, ULFM MPI provides recovery capabilities that enable the development of resilient communication operations employed in distributed deep learning. These procedures involve handling unexpected events by obtaining acknowledgments of reported process failure errors and identifying the group of processes that have been acknowledged as containing the failure. This can be accomplished using the `MPIX_Comm_failure_ack` and `MPIX_Comm_failure_get_acked` routines. While the consensus regarding the failures is achieved through collective operation, `MPIX_Comm_agree`, which ensures agreement across all participating processes. After reaching an agreement, the application user can make decisions based on the corresponding error. By utilizing the `MPI_Comm_set_errhandler` function provided by MPI, the user has the ability to specify how error codes should be handled.

This can be done by either returning the error codes to the application or invoking a user-defined error handler procedure. Through this approach, we are able to handle events related to changes in worker size by the error handler function. In our design, we offer users a runtime command line flag that allows them to choose whether to drop a single process or the entire node in the event of changing worker size. This helps prevent node-level issues from causing additional failures. In the event of an unexpected occurrence, all processes associated with the same communicator will invoke `MPIX_Comm_revoke` to locally interrupt ongoing operations. Depending on the user-provided runtime flag, we either drop the failed process or eliminate the entire node, meanwhile, restoring the communicator with the routine `MPIX_Comm_shrink`. Given that, a large portion of communication operations during a distributed parallel training are collective operations, particularly allreduce and allgather, which are extensively used within a training epoch, we have developed a versatile error handler function to effectively manage unforeseen events that may occur during these collective operations. With this design approach, resilient collective operations serve as the primary method to handle any changes in worker size during training. To assess the effectiveness of our design, we have integrated the resilient collective operation into Horovod, enabling a comparison with the state-of-the-art elastic training mechanisms used in distributed systems.

3.2 Elastic training based on resilient collectives

During the training of a distributed deep neural network (DNN), workers frequently exchange data, primarily using the collective communication operation to combine worker contributions and synchronize gradients. Allreduce is one of the most commonly employed collective operations in DNN, which requires all workers to participate in reducing gradients regarding the same tensor. If a worker drops out during training, one of the gradient aggregation operations will fail, and the connection between workers must be rebuilt, forcing the training to stop to address the problem. With the help of ULFM MPI, however, failure information can be quickly and easily propagated to other workers, and the communicator can be reconstructed with the surviving workers.

Since all surviving workers contain all information about the failed Allreduce operation and retain the data from that operation, they can continue the training process by repeating the failed Allreduce operation. As a consequence, it eliminates the need to repeat the current mini-batch training to the contribution. This approach differs significantly from traditional checkpoint-based failure recovery, which involves backward re-computation to ensure accuracy. As shown in Figure 2, cutting-edge elastic training technologies traditionally require a minimum checkpoint interval of one mini-batch to save the training state. Within each mini-batch training, there could be multiple steps of computing and reducing gradients, with each aggregation denoted as *ARD* (Allreduce). In the event of a worker disconnecting, the conventional checkpoint-based approach necessitates rolling back to the last checkpoint. This checkpoint corresponds to the contributions before the current mini-batch. However, the resilient Allreduce method enables the surviving workers to redo the current Allreduce operation and compile the gradients based on the remaining contributions. In the approach we propose, the smallest granularity for recovery is each individual collective operation. This results in significantly lower costs compared to retraining the entire mini-batch.

The elasticity in training functionality was primarily designed for cloud systems, with Gloo and NCCL being the predominant communication libraries employed on these platforms. Nevertheless, as illustrated in Fig.3, both Gloo and NCCL lack the ability to tolerate failures and reconfigure workers during runtime. To address this issue, Horovod introduced the Elastic Horovod feature that manages the reconfiguration of workers based on exception codes caught from lower levels. In contrast, ULFM MPI can directly capture and manage exception signals, independent of both Horovod and the training engine. It can restore the training process by rapidly reconstructing communicators and repeating any failed operations.

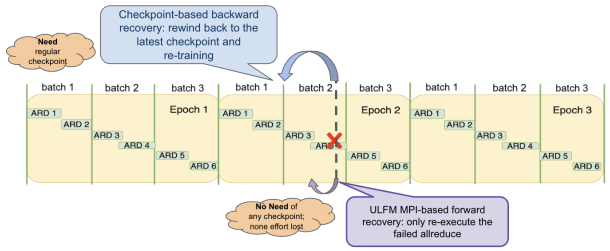

Figure 2: Backward recovery and proposed forward recovery based on ULFM MPI

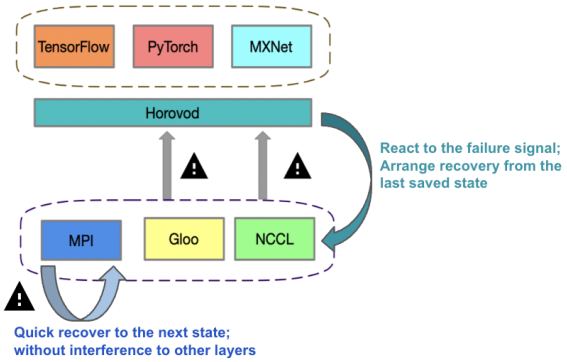

Figure 3: High-level abstract of the fault recovery and reconfiguration capability in ULFM MPI

3.3 Dynamic scaling depending on available resources

3.3.1 ***Scenario I: Downscaling Recovery.*** In this particular case, when a failure occurs, every worker is alerted about the shift in the current number of workers. The communication context is then rebuilt, excluding the malfunctioning workers. Elastic Horovod approaches this by blacklisting the entire node to avoid potential hardware issues at the node level, which prompts all remaining workers on that node to discontinue the training process. Nevertheless, the remaining nodes create a fresh communication context and resume training without having to restart the whole process. In our implementation, we provide runtime options to either eliminate only the dysfunctional workers or exclude the whole node, thereby offering flexibility in managing fault tolerance.

3.3.2 ***Scenario II: Replacement Recovery.*** In this situation, our objective is to keep the original number of workers even if a failure arises. This ensures training parameters associated with the worker size remain stable. As a result, we allocate the exact number of faulty workers or nodes to join the ongoing training process. Similar to Elastic Horovod, this method operates at the node level, guaranteeing the same adjustment unit. Moreover, ULFM MPI provides the flexibility to substitute individual processes or entire nodes with multiple processes, granting fine-tuned control over fault tolerance mechanisms. Both **Scenario I** and **Scenario II** can be considered as strategies related to failure recovery.

3.3.3 ***Scenario III: Automated Upscaling.*** In real-world scenarios, it is typical for distributed resources to be in inconsistent states at any point in time. Under such circumstances, while some computing resources might be ready to initiate training, others could be busy with other tasks. Instead of waiting for all resources to become available at once, a more effective strategy is to start training with the available workers and synchronize with the remaining resources as they become ready. This promotes a more adaptable training process that optimizes resource utilization.

4 EVALUATION

4.1 Experiment setup

The proposed design was assessed on Summit, a top-tier supercomputer located at the Oak Ridge National Laboratory. Every computing node of Summit consists of 2 IBM POWER9 CPUs and 6 NVIDIA V100 GPUs, each loaded with 16 GB of HBM memory. providing a node injection bandwidth of 23 GB/s. Three pre-trained Keras image recognition models, as cited in [6], were chosen considering their parameter size distribution, and were trained using ImageNet datasets [5]. The rationale behind choosing these applications was their trainable parameter size, directly influencing the count of Allreduce operations on both CPUs and GPUs. The scale of the training model significantly affects the expenses related to the creation and loading of checkpoints. We set up and deployed Horovod with optimal environmental variables such as tensor fusion and response caching sizes. For the sake of comparability in our experiments, we've limited our focus to memory checkpoints in the subsequent evaluation. This means that we do not delve into the costs associated with saving and loading checkpoints on parallel file system within this context. This section will solely focus on the costs associated with reconstructing the communication context, re-establishing rendezvous, and the re-computation costs necessary to resume training.

Table 1: Keras benchmark applications

Model	Trainable	Depth	Total Parameters	Size (MB)
VGG-16	32	16	143.7M	549
ResNet50V2	272	307	25.6M	98
NasNetMobile	1126	389	5.3M	23

For scientific study, we selected Elastic Horovod as a representative technology that uses checkpoints for comparison purposes. However, it currently only provides support for failure recovery through GLOO and NCCL communication libraries. To avoid any potential overhead of GPU operations from different libraries, we made modifications in Horovod code to integrate ULFM MPI support for fault tolerance and communication among hosts, and we delegated all GPU computation and communication tasks to NCCL.

In our study, we evaluated three recovery scenarios using an equivalent number of GPUs. As Elastic Horovod only provides support for node-level failures, we conducted comparisons at both the process and node levels, providing the flexibility to choose from different levels. Consequently, we tested and validated two recovery levels in the scaling-in, replacement, and scaling-out cases. To emulate elastic computing resources during training on HPC

systems, we explore three scenarios that require reconfiguration of workers in varying sizes.

Table 2: Recovery capabilities of different communication libraries

Dynamic training scenarios	Elastic Horovod	ULFM MPI
Recovery by process	×	√
Recovery by node	√	√
Autoscaling by process	×	√
Autoscaling by node	√	√

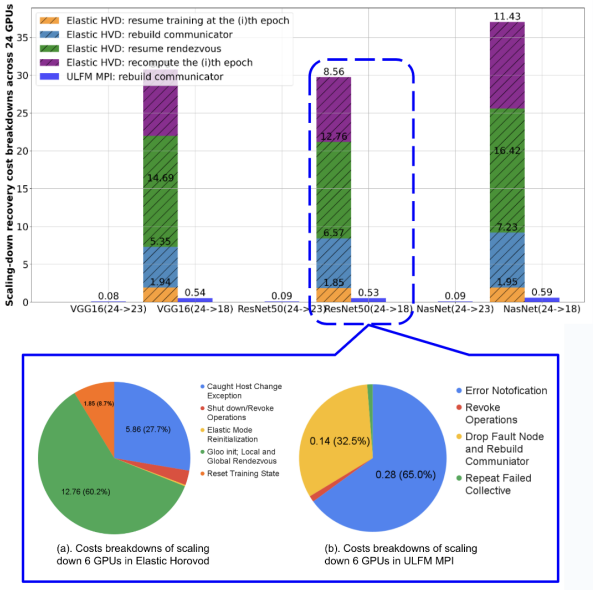

Figure 4: Detailed cost breakdowns in Scenario I when training ResNet 50 across 24 GPUs, with 18 GPUs left after resuming from failure.

We have segmented the costs associated with restoration in Elastic Horovod in Fig.4, illustrating the first scenarios mentioned in the previous section across 24 GPUs. To exemplify this, we conducted experiments using ResNet 50 training across 4 nodes, considering situations of dropping the failed process and dropping the entire node. In these experiments, we profiled the costs of each step, including catching exceptions, shutting down ongoing operations, re-initializing the elastic mode, reinitializing Gloo, and resuming both local and global rendezvous. In scenarios where dropping a node is required, the most time-consuming aspect is the reconstruction of the Gloo context and rendezvous. On the other hand, when new workers need to be added, there are additional costs associated with loading and initializing the necessary software libraries for new workers. Once new workers, with initialized training states, are prepared to participate in the training, the entire process is fully resumed. For the backward recovery method, the starting point is from the last checkpoint state, which, in our experiments, is the epoch at which failure occurred *(i)th* epoch. On the other hand, in the forward recovery method, the new workers receive the contribution from the failed *(i)th* epoch from the survivors, thus they commence from the *(i+1)th* epoch. However, it is important to note that this cost is only incurred once for every worker, until they exit the training execution.

In the following experiments, we scrutinize our methodology's performance in relation to Elastic Horovod across different models and scenarios. To draw a direct comparison between the two techniques, we conduct a detailed evaluation and analysis of their respective capabilities and performance. In this analysis, we categorize the costs into three principal segments: reconstructing the communicator and resuming rendezvous, reinitializing the training state for the new workers, and the cost associated with re-computation.

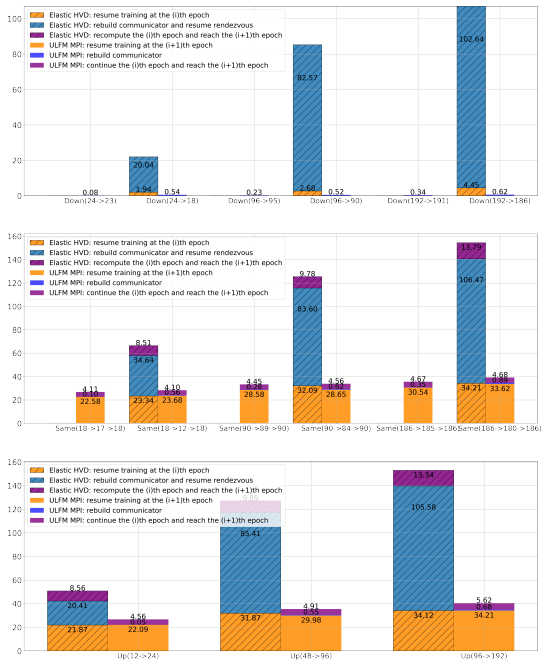

Figure 5: Costs (seconds) of recovering/reconfiguring workers when training the VGG-16 model in three scenarios: Scenario I: Dropping the failed process/node ("Down"); Scenario II: Remaining the original worker size by replacing the failed process/node with new process/node ("Same"); Scenario III: Automated doubling the worker size during the training ("Up"); scaling from 12 GPUs to utmost 192 GPUs

As our approach focuses on forward recovery, any new workers are introduced after the current epoch is completed by the existing processes. Therefore, we have analyzed the execution duration of the remaining epoch that includes worker changes. It is important to note that this duration represents continued training and does not incur additional costs. During this period, the surviving processes make valid contributions to the training process. Based on Figure 8 to Figure 10, it allows us to conduct a clear analysis of the costs associated with each of the abovementioned steps, and evaluation of the overall effectiveness of our approach compared to Elastic Horovod.

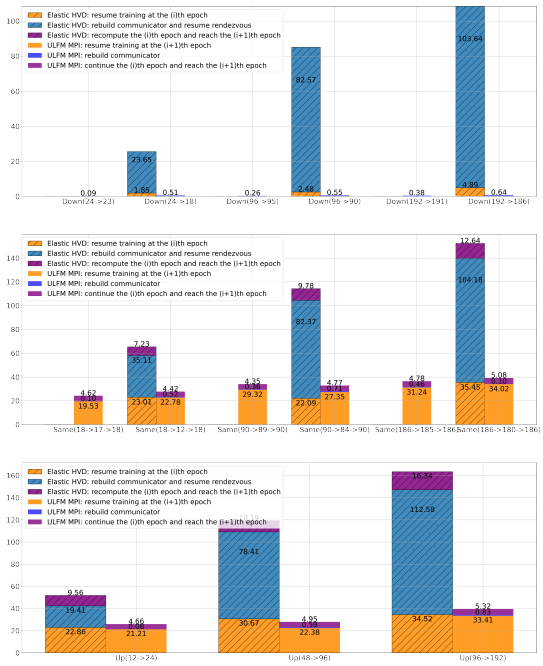

Figure 6: Costs (seconds) of recovering/reconfiguring workers when training the ResNet-50 model in three scenarios

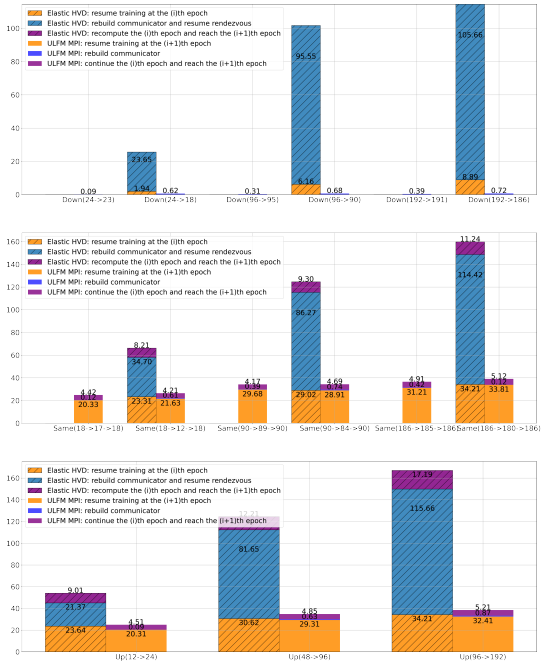

Figure 7: Costs (seconds) of recovering/reconfiguring workers when training the NasNet model in three scenarios

The results derived from these experiments clearly demonstrate that ULFM MPI consistently produces less overhead when reconstructing the communication context compared to Elastic Horovod via Gloo. This holds true irrespective of whether workers are added or removed during training. Furthermore, ULFM MPI provides the added benefit of flexibility in managing each individual process with minimal cost. This advantage becomes increasingly significant at larger scales.

5 RELATED WORK

To accommodate the dynamic addition or removal of servers and workers, Litz[29] is designed with capabilities of updating forwarding and executor migration. Cruise[23], on the other hand, uses a performance model to dynamically tweak the settings of parameter servers and workers to achieve optimal performance. These technologies, as representative of the field, rely on the parameter server, which has limited scalability on high-performance computing systems on a large scale. Similar with Elastic Horovod, recent proposed studies ElasticDL[40] and Pytorch-Elastic[4] use the checkpoint to replicate the training state. Our method is constructed using the highly scalable and portable communication library, MPI, which offers a scalable, decentralized, and lightweight solution for elastic resource management during training. It eliminates the need for shutdown-restart and checkpoint procedures by reducing the recovery tasks to just a single collective communication. Elan[37] also leverages collective communication to incorporate elasticity in deep learning, much like our approach. However, their focus primarily lies on the concurrent and efficient replication of state. When scaling DL, techniques to overcome the convergence issues include learning rate adjustment[22] and warmup scheme[16] are introduced to overcome the convergence problem at scale. Dynamic workload scheduling based on parallelism was proposed by[35] has been proposed to ensure that all data samples are used for training within a single epoch, even in case of failure.

6 CONCLUSION

Our approach contrasts with existing malleable deep learning systems with its rapid reaction to fault events and malleability requests: we employ a novel run-through recovery strategy that lets the training continue the failed epoch in degraded mode, before a thorough restructuring of the computation lets replacement or new resources join, and high performance networking, as available on HPC systems, be restored to its ideal capacity. Overall, our solution offers an efficient and flexible means of managing training processes, enabling researchers to leverage the full potential of high-performance computing environments for large-scale deep learning tasks.

ACKNOWLEDGEMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is based upon work supported by the National Science Foundation under Grant No. OAC-1664142 and by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under contract DEAC02-06CH11357.

REFERENCES

[1] [n. d.]. Gloo. https://github.com/facebookincubator/gloo, Last accessed on 2023-7-4.
[2] [n. d.]. NCCL. https://github.com/NVIDIA/nccl, Last accessed on 2023-7-4.

[3] 2019. Horoovd. https://github.com/horovod/horovod, Last accessed on 2023-7-4.
[4] 2019. Pytorch elastic. https://github.com/pytorch/elastic, Last accessed on 2023-7-4.
[5] 2022. *Kaggle Fruits 360 datasets.* https://www.kaggle.com/datasets/moltean/fruits
[6] 2022. *Keras Applications.* https://keras.io/api/applications/
[7] Kamal K Agarwal and Haribabu Kotakula. 2022. Replication Based Fault Tolerance Approach for Cloud. In *International Conference on Distributed Computing and Internet Technology*. Springer, 163–169.
[8] Mirco Altenbernd, Nils-Arne Dreier, Christian Engwer, and Dominik Göddeke. 2021. Towards Local-Failure Local-Recovery in PDE Frameworks: The Case of Linear Solvers. In *High Performance Computing in Science and Engineering: 4th International Conference, HPCSE 2019, Karolinka, Czech Republic, May 20–23, 2019, Revised Selected Papers 4*. Springer, 17–38.
[9] Vinay Amatya, Abhinav Vishnu, Charles Siegel, and Jeff Daily. 2017. What does fault tolerant deep learning need from mpi?. In *Proceedings of the 24th European MPI Users' Group Meeting*. 1–11.
[10] Aurelien Bouteiller and George Bosilca. 2022. Implicit Actions and Non-blocking Failure Recovery with MPI. In *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 36–46.
[11] Aurelien Bouteiller, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. 2006. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *The International Journal of High Performance Computing Applications* 20, 3 (2006), 319–333.
[12] Edson T Camargo and Elias P Duarte. 2021. An Algorithm-Based Fault Tolerance Strategy for the Bitonic Sort Parallel Algorithm. In *2021 10th Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 1–10.
[13] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. 2020. Elastic parameter server load distribution in deep learning clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 507–521.
[14] Tobias Distler. 2021. Byzantine fault-tolerant state-machine replication from a systems perspective. *ACM Computing Surveys (CSUR)* 54, 1 (2021), 1–38.
[15] Laird Egan, Dripto M Debroy, Crystal Noel, Andrew Risinger, Daiwei Zhu, Debopriyo Biswas, Michael Newman, Muyuan Li, Kenneth R Brown, Marko Cetina, et al. 2021. Fault-tolerant control of an error-corrected qubit. *Nature* 598, 7880 (2021), 281–286.
[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).
[17] David Grove, Sara S Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Olivier Tardieu. 2019. Failure recovery in resilient X10. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 3 (2019), 1–30.
[18] Sara S Hamouda, Benjamin Herta, Josh Milthorpe, David Grove, and Olivier Tardieu. 2016. Resilient X10 over MPI user level failure mitigation. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. 18–23.
[19] Siva Kumar Sastry Hari, Michael B Sullivan, Timothy Tsai, and Stephen W Keckler. 2021. Making convolutions resilient via algorithm-based error detection techniques. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2546–2558.
[20] Sachini Jayasekara, Shanika Karunasekera, and Aaron Harwood. 2022. Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: Theory to practice. *Software: Practice and Experience* 52, 1 (2022), 296–315.
[21] J.Stengler. 2017. Fault tolerant Collective communication Algorithms For distributed database systems. (2017).
[22] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).
[23] Woo-Yeon Lee, Yunseong Lee, Joo Seong Jeong, Gyeong-In Yu, Joo Yeon Kim, Ho Jin Park, Beomyeol Jeon, Wonwook Song, Gunhee Kim, Markus Weimer, et al. 2019. Automating system configuration of distributed machine learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2057–2067.
[24] Nuria Losada, Patricia González, María J Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. 2020. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Generation Computer Systems* 106 (2020), 467–481.
[25] Bogdan Nicolae. 2013. Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 19–28.
[26] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 172–181.
[27] Bogdan Nicolae, Justin M Wozniak, Matthieu Dorier, and Franck Cappello. 2020. DeepClone: Lightweight state replication of deep learning models for data parallel training. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 226–236.
[28] Andrew Or, Haoyu Zhang, and Michael Freedman. 2020. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems* 2 (2020), 400–411.
[29] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A Gibson, and Eric P Xing. 2018. Litz: Elastic framework for {High-Performance} distributed machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 631–644.
[30] Seth Roffe and Alan D George. 2020. Evaluation of algorithm-based fault tolerance for machine learning and computer vision under neutron radiation. In *2020 IEEE Aerospace Conference*. IEEE, 1–9.
[31] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista Gomez, and Rosa M Badia. 2020. A study of checkpointing in large scale training of deep neural networks. *arXiv preprint arXiv:2012.00825* (2020).
[32] Amrith Rajagopal Setlur, S Jaya Nirmala, Har Simrat Singh, and Sudhanshu Khoriya. 2020. An efficient fault tolerant workflow scheduling approach using replication heuristics and checkpointing in the cloud. *J. Parallel and Distrib. Comput.* 136 (2020), 14–28.
[33] Sharifah Hafizah Sy Ahmad Ubaidillah, Basem Alkazemi, and A Noraziah. 2021. An Efficient Data Replication Technique with Fault Tolerance Approach using BVAG with Checkpoint and Rollback-Recovery. *International Journal of Advanced Computer Science and Applications* 12, 1 (2021), 475–480.
[34] Matthew Whitlock, Nicolas Morales, George Bosilca, Aurelien Bouteiller, Bogdan Nicolae, Keita Teranishi, Elisabeth Giem, and Vivek Sarkar. 2022. Integrating process, control-flow, and data resiliency layers using a hybrid Fenix/Kokkos approach. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 418–428.
[35] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2021. Elastic deep learning in multi-tenant GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 144–158.
[36] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.
[37] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. 2020. Elan: Towards generic and efficient elastic training for deep learning. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 78–88.
[38] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. 2020. FT-CNN: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1677–1689.
[39] Diyu Zhou and Yuval Tamir. 2021. Hycor: Fault-tolerant replicated containers based on checkpoint and replay. *arXiv preprint arXiv:2101.09584* (2021).
[40] Jun Zhou, Ke Zhang, Feng Zhu, Qitao Shi, Wenjing Fang, Lin Wang, and Yi Wang. 2023. ElasticDL: A Kubernetes-native Deep Learning Framework with Fault-tolerance and Elastic Scheduling. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*. 1148–1151.