Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates

Insu Jang
insujang@umich.edu
University of Michigan

Zhenning Yang
znyang@umich.edu
University of Michigan

Zhen Zhang
zhzhn@amazon.com
Amazon Web Services

Xin Jin
xinjinpku@pku.edu.cn
Peking University

Mosharaf Chowdhury
mosharaf@umich.edu
University of Michigan

Abstract

Oobleck enables resilient distributed training of large DNN models with guaranteed fault tolerance. It takes a planning-execution co-design approach, where it first generates a set of heterogeneous *pipeline templates* and instantiates at least $f + 1$ logically equivalent pipeline replicas to tolerate any f simultaneous failures. During execution, it relies on already-replicated model states across the replicas to provide fast recovery. Oobleck provably guarantees that some combination of the initially created pipeline templates can be used to cover all available resources after f or fewer simultaneous failures, thereby avoiding resource idling at all times. Evaluation on large DNN models with billions of parameters shows that Oobleck provides consistently high throughput, and it outperforms state-of-the-art fault tolerance solutions like Bamboo and Varuna by up to 13.9×.

CCS Concepts: • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; **Distributed architectures**; **Neural networks**.

Keywords: Fault tolerant training, distributed training, hybrid parallelism, pipeline template

ACM Reference Format:
Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3600006.3613152

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613152

1 Introduction

DNN models continue to become larger [49]. Many recent advances in deep learning have been attributed to significant increases in model size to hundreds of billions of parameters and training on ever-growing datasets [5, 31, 32, 45]. Recent studies suggest that a trillion-parameter model would require at least 2TB of memory simply to store model parameters, and tens or hundreds of TB for training [18, 24, 37, 38, 42]. Naturally, scaling large model training has received intense attention over the past few years [3, 11, 29, 45, 53]. Distributed *hybrid-parallel training* [30, 56] that combines model and data parallelism has emerged as the primary approach to training such large models.

Unfortunately, the likelihood of experiencing failures also increases with the scale and duration of training [12, 17, 50]. The effect is further amplified by the synchronous nature of DNN training, which causes all participating devices to idle until the failed one has recovered, causing massive underutilization. Indeed, teams from Meta, HuggingFace, and LAION report significant underutilization from failures when training large models [2, 3, 53]. Failure rates are even higher for training jobs that use spot instances in the cloud [1, 48].

Existing frameworks have little systematic support for fault tolerance during hybrid-parallel training. Ensuring continuous operation in the presence of failures fundamentally requires redundancy in one form or another. Model state redundancy in data-parallel training is the only form of "free" redundancy, because each worker already has a copy of the model states. Most solutions harness the inherent redundancy provided by data parallelism and utilize its embarrassingly parallel nature to elastically change the number of GPUs while dynamically changing the global batch size [16, 21, 34, 54]. However, they are unable to extend these benefits to hybrid parallelism and are limited only to data-parallel training.

In contrast, fault tolerance approaches tailored toward hybrid parallelism struggle to leverage any inherent redundancy. Instead, they introduce additional redundancy in various forms; e.g., having a pool of standby GPUs [3], using checkpoints to reconfigure and restart [1], and performing

redundant computations in anticipation of a possible failure [48]. Essentially, they consider overhead during training vs. overhead to recover from failure(s), and choose one of the two extremes (§2.2): if failures are infrequent, amortized overhead for reconfiguration would also be low [1]; if failures are more frequent, then incurring some overhead during training may be more preferable than spending significant time in recovery [48].

In this paper, we present Oobleck, a fault-tolerant hybrid-parallel training framework. It provides high training throughput, guaranteed fault tolerance, and fast recovery *without* introducing additional overhead. *Pipeline templates* are at the core of Oobleck's design. A pipeline template is a specification of pipeline execution for a given number of nodes. They are designed during the planning phase by Oobleck's template generator and reused during execution by Oobleck's execution engine. All pipeline templates are logically equivalent yet physically heterogeneous; each has a different number of nodes and associated configurations that can be used to instantiate a pipeline for a given model. Oobleck uses one or more pipeline templates to create pipeline replicas to exploit the inherent model states redundancy across the replicas. Pipelines affected by failures can reconstruct model states by copying missing layers from other replicas without having to restart from a checkpoint.

More specifically, given a training job starting with the number of maximum simultaneous failures to tolerate f, Oobleck's execution engine instantiates at least $f + 1$ heterogeneous pipelines from the generated templates. The fixed global batch is distributed proportionally to the computing capability of heterogeneous pipelines such that all pipeline replicas train roughly at the same rate. Upon failures, Oobleck avoids demanding analysis of finding a new optimal configuration by simply reinstantiating pipelines from the precomputed pipeline templates while achieving maximum node utilization. This is always possible for f or fewer failures because Oobleck provably guarantees that a combination of pipelines generated from those precomputed templates can fully utilize all the remaining nodes.

We have implemented Oobleck on top of PyTorch and HuggingFace Transformers [51] using components from DeepSpeed [40] and Merak [19]. We evaluate Oobleck and compare its performance against Bamboo and Varuna across large models like GPT-3 with billions of parameters. Oobleck outperforms the state-of-the-art solutions by up to 13.9× as we consider different frequencies of failures, spot instance traces, and models of different sizes and computation complexity.

Overall, we make the following contributions in this paper.

- We present Oobleck, a novel framework for resilient distributed training that provides guaranteed fault tolerance and maximizes throughput.
- Oobleck introduces pipeline templates to (re)instantiate pipelines. Pipeline templates allow quick failure recovery and utilization of all available GPUs.
- We implement and evaluate Oobleck with several large models, e.g., variants of GPT-3, to demonstrate large improvements in terms of throughput and failure recovery.

Oobleck is open-source and available on GitHub.[1]

2 Background and Motivation

In this section, we briefly introduce hybrid parallelism that is commonly used for large model training. We also discuss existing fault tolerance strategies for hybrid-parallel training and highlight their limitations.

2.1 Hybrid Parallelism

As DNN models continue to grow in size and are trained on increasingly larger datasets [3, 45, 53], using just *data parallelism* or *model parallelism* is often not enough to efficiently train a DNN. Data parallelism splits and distributes input to multiple GPUs, but it requires each GPU to hold the entire model [16]. Model parallelism accommodates large model training by splitting the model across multiple GPUs – for example, *pipeline parallelism* splits the model into groups of layers called stages, and *tensor parallelism* slices each model layer into several tensor chunks. However, the former has pipeline bubble overheads that decrease compute utilization as the pipeline grows deeper [45]. The latter suffers from high communication cost that cannot be hidden in computation, because it requires several all-reduce operations in the critical path of both forward pass and back-propagation [37]. Consequently, a combination of data and model parallelism techniques – aka *hybrid parallelism* – is used in practice to train large DNN models [3, 11, 29, 30, 45].

2.2 Fault Tolerance in Distributed Training

Failures are the norm in distributed systems, and distributed DNN training is no exception. The probability of experiencing one or more failures increases with the increasing number of GPUs and the duration of training. For instance, a Meta AI team suffered approximately 100+ hardware failures and had to do 100+ major restarts during OPT-175B training [53]. Because of the synchronous nature of distributed training, the cost of even one failure is *multiplied*: all the GPUs must idle until the impact of failure has been mitigated. The impact of this phenomenon was recently highlighted in detail by a LAION team when training CLIP models [2] as well as a BigScience team during BLOOM training [3].

Several recent works have focused on fault-tolerant data-parallel training via dynamically changing the global batch size [16, 21, 34, 52]. Fault-tolerant hybrid-parallel training is more challenging because the model is distributed across multiple GPUs. There are two primary approaches.

[1] https://github.com/SymbioticLab/Oobleck

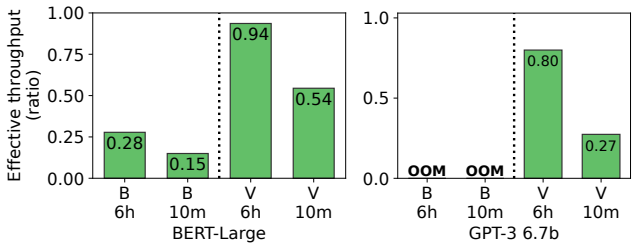


Figure 1. *Effective time spent in training for Bamboo (B) and Varuna (V) running BERT-large and GPT-3 6.7b models for different frequency of failures (6h and 10m). An optimistic upper-bound of the optimal is 1.00, when training remains unaffected by failure(s).*

1. *Checkpointing:* Checkpointing is a popular mechanism to persistently store training progress. For example, the BigScience team training the BLOOM model [3] and the Meta AI team training the OPT model [53] used this recently. However, manual reconfiguration after identifying and replacing the failed GPUs with spare ones is time-consuming. Varuna [1] introduced job morphing to dynamically reconfigure training jobs to achieve the best performance with the remaining resources after restarting from the most recent checkpoint. While this does not introduce significant fixed overhead, recovery time can be unsustainable when the failure rate is high [48].
2. *Redundant computation (RC):* To avoid reconfiguration and restart overheads, Bamboo [48] recently introduced redundant computation (RC) where each pipeline stage is redundantly computed in two subsequent nodes. When a node fails, the backup node computes the forward and backward passes of the failed node. RC introduces fixed computational overhead due to redundancy in computing and memory overhead of holding redundant states in each node. Note that reconfiguration and restart from a checkpoint is still necessary if two adjacent nodes fail.

2.3 Limitations of the State-of-the-Art

State-of-the-art approaches for fault-tolerant hybrid-parallel training do not provide any systematic fault tolerance guarantees, and they have large overheads, especially when models become larger.

Figure 1 shows the time effectively spent in training (i.e., the time that leads to training throughput) using Varuna and Bamboo when training BERT-Large and GPT3-6.7B – with 340 million and 6.7 billion parameters, respectively – when one failure happens every six hours and every 10 minutes on average. Detailed experimental setup is in Section 7.1. Varuna provides higher training throughput compared to Bamboo, but it has noticeable job restart overhead. Although some recent proposals improved checkpointing overhead [10, 28], loading checkpoints upon restarts is still in the critical path. When failures become more frequent, restarting overheads dominate. Additionally, performance degradation after a failure is not proportional to the number of failures, but worse. This is because Varuna's hybrid parallelism uses a grid topology; one GPU failure breaks the grid of GPUs, leaving some of them idle.

Bamboo, in contrast, reduces checkpointing and restart overheads, but RC in Bamboo introduces significant performance overhead, even when some portion of the overhead is hidden in pipeline bubbles. Specifically, its forward RC redundantly computes the next stage all the time, lowering throughput even in the absence of failures. Backward RC takes place only after failure(s), but it adds additional overhead to some pipelines' iteration times, making them stragglers and inflating the iteration time of synchronous training. Worse, Bamboo also needs to restart with a full reconfiguration from a checkpoint for as few as two failures when two adjacent nodes fail.

Finally, both approaches perform poorly for larger models, especially when failures are frequent. Bamboo runs out of memory and Varuna spends most of the time preparing to train. We aim to design a solution that works well regardless of the frequency of failures both in terms of the fault tolerance guarantee it provides and the throughput it achieves.

3 Oobleck Overview

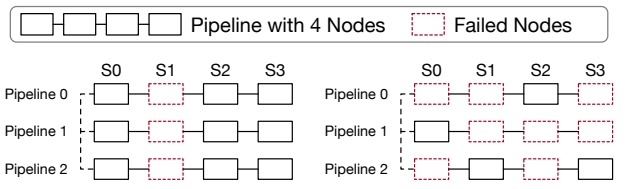
Oobleck is a resilient distributed training platform for large models with guaranteed fault tolerance. It presents the concept of *pipeline templates* to achieve high throughput and fast fault tolerance at the same time (§3.1). Its fault tolerance guarantees allow for reconfiguration without restarts for up to f simultaneous failures in the worst case (§3.2). We also discuss Oobleck's overall architecture (§3.3) and how it integrates with the training lifecycle (§3.4).

3.1 Pipeline Templates

Oobleck introduces *pipeline templates*, each of which is a pipeline specification that defines how many nodes should be assigned to a pipeline, how many stages to create, and how to map model layers in stages to GPUs. All pipelines *instantiated* by Oobleck are from precomputed pipeline templates. In practice, Oobleck instantiates multiple (possibly heterogeneous) pipelines from a set of heterogeneous pipeline templates to fully utilize an arbitrary number of nodes even when they do not form a grid. Decoupling "planning" (pipeline template generation) from "execution" (pipeline instantiation) enables fast failure recovery; a pipeline with lost node(s) is replaced with a new pipeline instantiated from another pipeline template that requires a fewer number of nodes.

3.2 Fault Tolerance Guarantees

Oobleck guarantees fault tolerance without restart for up to f simultaneous *pipeline* failures, because in the worst case, f node failures are enough to cause f pipelines to fail. Consider Figure 2, where there are three pipeline replicas each



(a) *Failures happen to nodes with the same stage (S1 lost, not recoverable)* **(b)** *Failures in random places (all stages alive, recoverable)*

Figure 2. *An example of Oobleck's fault tolerance guarantees with $f = 2$. S refers to a pipeline stage. (a) In the worst case, we lose model states (a stage) if more than f nodes fail. (b) In the general case, however, more than f node failures can be tolerated.*

with four stages – i.e., each stage has three replicas. We can tolerate at most two simultaneous node failures in the worst case, because if three failures take out all three replicas of any stage (stage 1 in Figure 2a), the pipelines cannot be recovered. In general, however, Oobleck can tolerate in excess of f node failures, provided that a minimum of one copy of the entire model states is retained across the pipelines. For example, even after eight node failures in Figure 2b, one copy of each stage still remains alive; hence, it is recoverable.

3.3 System Components

Oobleck extends existing ML training frameworks in two primary aspects (Figure 3). First, it has *a pipeline template generator* to generate a set of heterogeneous pipeline templates that can be used by the execution engine for pipeline instantiation. Pipeline templates are created only once and never change during the entire training.

Second, Oobleck has a *distributed execution engine* that enables efficient heterogeneous pipeline execution. It instantiates pipelines from the given set of pipeline templates considering the user's fault tolerance threshold (f) and batch information (global batch and microbatch size). It creates at least $f + 1$ (possibly heterogeneous) pipeline replicas so that at least one copy of the model exists anytime during training for up to f simultaneous failures. The batch distributor calculates the number of microbatches for each pipeline that balances execution latency between heterogeneous pipelines. Pipeline instantiation and batch distribution happens whenever a node fails or is added. The node change monitor detects node failure(s) and node additions; then the execution engine dynamically reconfigures using precomputed pipeline templates.

3.4 Training Lifecycle

Oobleck users submit training jobs with a fault tolerance threshold f, a model and dataset to train on N homogeneous nodes (received from a GPU cluster manager [17, 50]), and batch size information ① (Figure 3). Oobleck's pipeline template generator first creates a set of pipeline templates ②. The distributed execution engine instantiates pipelines from the templates and deploys them on the cluster ③.

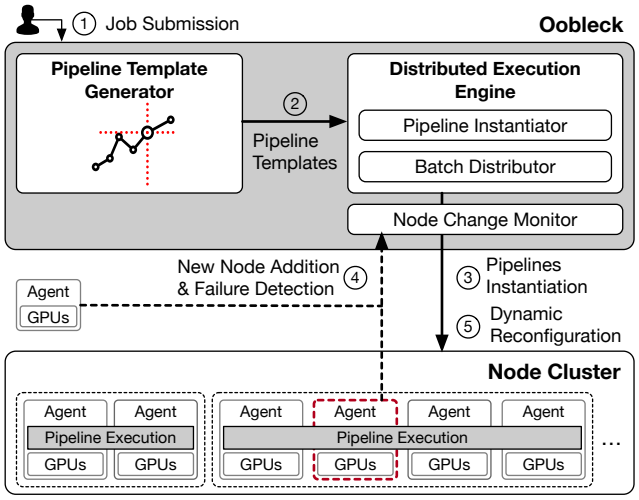


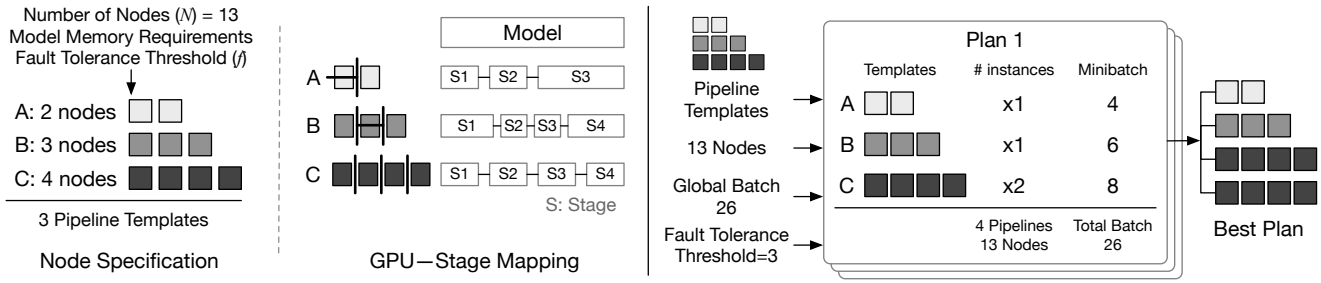
Figure 3. *Oobleck system overview.*

When node failure(s) happen ④, if we have a complete model replica, Oobleck does not restart but reconfigures the pipelines ⑤. The execution engine reinstantiates pipelines from the templates to make sure all nodes are used. During pipeline reinstantiation, nodes share information about the ownership of model states and copy missing model states from others. After reconfiguration and model states copying are done, nodes resume training. A job runs until it reaches the target accuracy, a user terminates it, or Oobleck cannot maintain $f + 1$ pipeline replicas. If the cluster cannot hold $f + 1$ replicas, Oobleck stores the progress, informs the user, and exits. Thereafter, the user can decide to restart training from a recent checkpoint once enough nodes have recovered to maintain $f + 1$ replicas.

4 Oobleck Planning Algorithm

Oobleck tolerates f simultaneous failures by instantiating r ($\geq f + 1$) heterogeneous pipeline replicas of the same model. Each of these logically equivalent pipeline replicas performs hybrid-parallel training. Unlike existing solutions that force a single homogeneous hybrid-parallel configuration over a rigid grid (# GPUs per pipeline stage × # pipeline stages × # pipeline replicas) [1], Oobleck's *heterogeneous pipeline execution* can utilize all available GPUs.

Because the number of available nodes can vary over time due to failures, Oobleck requires an effective mechanism to derive *all* possible configurations of heterogeneous pipelines that can utilize all available GPUs at any point in time. Oobleck's pipeline template generator computes a fixed set of pipeline templates at the beginning of the training job for the entire training (§4.1). The pipeline execution engine instantiates zero or more copies of each of the templates (i.e., a collection of heterogeneous pipeline replicas) to utilize all currently available nodes (§4.2).



(a) *Generating pipeline templates (§4.1)* **(b)** *Pipeline instantiation (§4.2)*

Figure 4. *Oobleck's planning algorithm overview. First, it generates a set of pipeline templates, a combination of which can utilize all available nodes. A template is a specification of pipelines, how many nodes are assigned and how GPUs in the nodes should be mapped to pipeline stages. Then, pipelines are instantiated following the fastest (best) plan after checking all possible plans. A plan includes how many pipelines should be instantiated from each pipeline template given the number of nodes and how batch size should be distributed to the pipelines.*

4.1 Generating Pipeline Templates

Each pipeline template created by Oobleck is a set of specifications that defines how many nodes to use, and how the given GPUs and model layers are mapped to make pipeline stages use all those nodes. Figure 4a illustrates this process. In this example, we generate a set of pipeline templates. We first determine the number of heterogeneous pipeline templates and their node specifications (number of nodes) needed to utilize all available nodes, given the initial number of nodes N, the amount of memory required to train a model, and the fault tolerance threshold f (§4.1.1). In this case, three heterogeneous pipeline templates with 2, 3, and 4 nodes have been chosen. Then, for each template, we partition the model and map the available GPUs to them to create pipeline stages that minimize the iteration time (§4.1.2).

4.1.1 Node Specification.

Because pipeline templates never change during training and generating arbitrarily many of them is expensive, we must determine how many pipeline templates are needed, and then how many nodes each of the templates should use, so that some combination of them can always utilize any number of available nodes, even when we have fewer number of nodes than at the beginning after failures.

This can be formulated as the Frobenius problem [39], which finds the Frobenius number g, the largest number that cannot be represented as a linear combination of integers. Meaning, any number of available nodes after failures $N' > g$ can be expressed as a linear combination of the given pipeline templates, each with a specified integer number of nodes.

If we represent the number of pipeline templates as p and the number of nodes for the i-th pipeline template be ordered values $n_i (0 < n_i \leq N)$ where $n_i < n_{i+1}$, we can guarantee that any feasible $N' \geq (f+1)n_0$ is always larger than g when the following conditions are met [43].

1. $p > n_0 - 1$.
2. n_i are consecutive integers ($n_i + 1 = n_{i+1}$).

See Appendix A for a proof.

We set the lower bound of N' as $(f+1)n_0$: the smallest number of nodes required to maintain $f+1$ replicas of the model, because n_0 is the smallest number of nodes for a single pipeline. Any smaller N' cannot respect the fault tolerance threshold f.

Choice of n_0 and p. There are several choices for a set of pipeline templates that satisfy the conditions depending on the values of n_0 and p. We choose the smallest possible n_0 and the largest p. We select the minimum n_0 because shallow pipeline execution (smaller n_0) typically takes less time for the same amount of computation [45]. Although a large p does not directly benefit planning, it helps reduce reconfiguration overhead. The largest p can be calculated from the largest possible value for n_{p-1} (referred to as $n_{p-1}^{\max}$). When all but one of the $f+1$ replicas use n_0 nodes and the last one uses all the remaining nodes, $n_{p-1}^{\max} = N - fn_0$. We now have p to be the length of the range from n_0 to $n_{p-1}^{\max}$.

4.1.2 GPU–Stage Mapping.

Given the number of nodes in each pipeline template, we must determine how to best use them by finding the number of pipeline stages, partitioning the model layers to the stages, and mapping the nodes to those stages. We propose a divide and conquer algorithm to find the mapping that minimizes the iteration time. This algorithm divides the model into pipeline stages and the nodes into a set of GPUs at the same time, and then maps each of them so that we utilize all GPUs in the given nodes. It then iterates over all possible combinations of GPU–stage mapping and finds the one that minimizes the iteration time.

Let $T(S', u, v, d)$ be the minimum iteration time for layers $(l_u, l_{u+1}, \ldots, l_{v-1})$ partitioned into S' stages and running on d GPUs. A pipeline for the entire model using all GPUs in the pipeline template then has the minimum iteration time $T(S, 0, L, n \cdot M)$, where the model has L layers, and there are n number of nodes in the pipeline template, each of which has M GPUs.

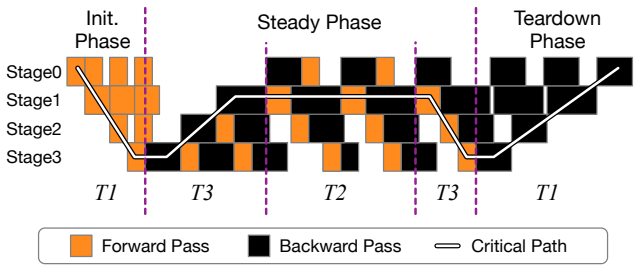


Figure 5. *1F1B pipeline execution breakdown ($T1, T2, T3$)*

To calculate the minimum iteration time of a pipeline, the algorithm considers its critical path and breaks T down into three terms $T1$, $T2$, and $T3$ (Figure 5). $T1$ represents 1F1B initialization and teardown phases of pipeline execution, which include one forward and one backward for all stages. The steady phase in the middle has one forward and one backward pass alternating. The critical path may still include forward and backward passes of other stages than the slowest one on each end, similar to $T1$. We thus split the steady phase into $T2$ and $T3$. $T2$ includes the slowest stage alternating forward and backward, and $T3$ is the remaining part.

Divide. In the division phase, we divide the model and the nodes at the same time. Division continues until we cannot partition either GPUs or model layers, or the number of partitions matches the desired number of stages. If multiple GPUs are assigned to a pipeline stage, tensor parallelism is used to accelerate it. Figure 6 illustrates such a division and mapping process. After both sub-problems are conquered, the algorithm combines their results to calculate the execution time of a multi-stage pipeline created by connecting two sub-problems. From the definitions of $T1$, $T2$, and $T3$, the division and combination process can be defined as recursive structures for each term:

$$\begin{aligned} &T1_{s,k,m}(S', u, v, d) \\ &= T1_{s,k,m}(s, u, k, m) + T1_{s,k,m}(S' - s, k+1, v, d-m) \end{aligned} \tag{1}$$

$$\begin{aligned} &T2_{s,k,m}(S', u, v, d \mid k^*) \\ &= (N_b - S' + k^* - 1)(F_{s_{k^*},m} + B_{s_{k^*},m}) \end{aligned} \tag{2}$$

$$\begin{aligned} &T3_{s,k,m}(S', u, v, d \mid k^*) \\ &= \begin{cases} T3_{s,k,m}(s, u, k, m \mid k_1^*) & \\ +T1_{s,k,m}(S' - s, k+1, v, d-m) & \text{if } k^* == k_1^* \\ T3_{s,k,m}(S' - s, k+1, v, d-m \mid k_2^*) & \text{else} \end{cases} \end{aligned} \tag{3}$$

We iterate over s, k, and m globally, and find a (s, k, m) that minimizes $T1_{s,k,m} + T2_{s,k,m} + T3_{s,k,m}$. Each $T1_{s,k,m}$, $T2_{s,k,m}$, and $T3_{s,k,m}$ is the solution of $T1$, $T2$, and $T3$, respectively.

k^* denotes the index of the slowest stage, derived from either k_1^* or k_2^*, the slowest stage indices of the two sub-problems. $T2$ depends on the number of microbatches (N_b) deployed to the pipeline, which is not yet determined. From

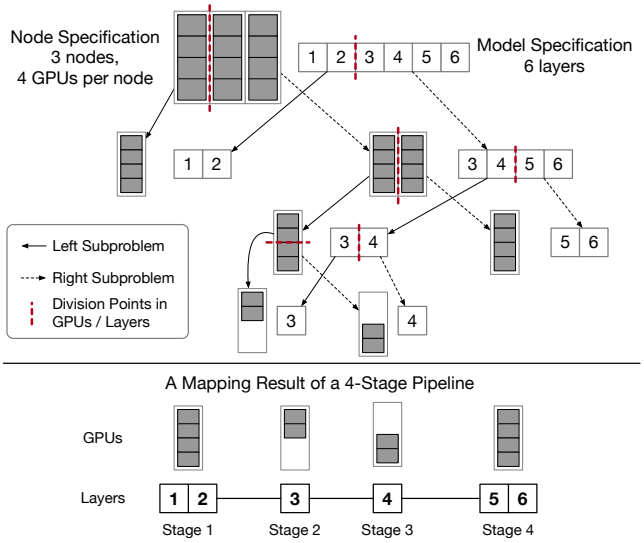


Figure 6. *A toy example of division process for a 4-stage pipeline template with 3 nodes (template B in Figure 4) and a model with 6 layers. The model and the GPUs in nodes are divided into two sub-problems together. When division is done, each group of partitioned GPUs and layers form a stage. The algorithm iterates all combinations of layer partitioning and GPU partitioning to find the minimum T.*

prior observations that the pipeline bubble overhead is negligible with $N_b \geq 4S'$ [15], we temporarily use $N_b = 4S'$ in planning. The structure of $T3$ is special as it includes $T1$ in Equation 3. T3 is an accumulation of forward and backward time for all the following stages after the slowest ($\sum_{k=k^*}^{S'-1}(F+B)$). If s_{k^*} is in the first half sub-problem (i.e. $s_{k^*} == s_{k_1^*}$), it can be broken down to $\sum_{k=k^*}^{s}(F+B) + \sum_{k=s+1}^{S'-1}(F+B)$, each of which represents $T3$ of the first half and $T1$ of the second half, respectively.

Conquer. When a problem has just one stage, we can easily calculate the execution time of running a stage s with $l_u, \ldots, l_{v-1}$ layers on d GPUs:

$$\begin{aligned} T1(1, u, v, d) &= F_{s,d} + B_{s,d} = \sum_{k=u}^{v-1}(F_{l_k,d} + B_{l_k,d}) \\ T2(1, u, v, d) &= 2(F_{s,d} + B_{s,d}) \\ T3(1, u, v, d) &= F_{s,d} + B_{s,d} \end{aligned} \tag{4}$$

There is one requirement for d GPUs running a single stage: *all d GPUs should be in the same node.* It is reasonable because if GPUs span several nodes, the cross-node network becomes a bottleneck and lowers the utilization of high-throughput intra-node network in collective communications done in intra-layer parallel execution. We simply mark all T values as ∞ if cross-node GPUs are given.

Choosing the number of stages S. We do not know which S provides the minimum iteration time. Therefore, we iterate over possible values of S in $(n, n+1, \ldots, L)$. Because we partition the model at layer granularity, the number of

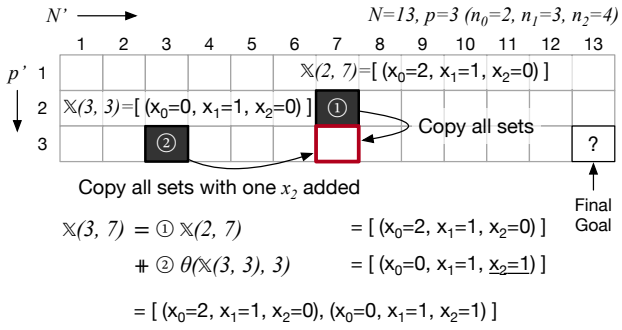


Figure 7. *The dynamic programming algorithm finding all list of feasible* $\mathbf{X}$*s. Underlined* $x_2 = 1$ *is added by* $\theta()$ *function.*

stages cannot exceed the number of layers L. The minimum is derived from the constraint that a single stage cannot be assigned to two or more nodes. If S becomes less than n, it breaks the constraint and some stages should have at least two nodes assigned according to the pigeonhole principle.

Time complexity of the naive implementation. The recursive stage division happens in $O(L)$. For every division, stages and layers are partitioned and they are assigned to two device sub-clusters. Stage and layer partitioning have $O(L)$ choices. Partitioning nodes is done in $O(n)$, but GPUs within a single node can further be partitioned, adding $O(M)$. Time complexity of layer partitioning and device assignment for a given number of stage is $O(LnM)$. Divide and conquer happens for each feasible S, and iterating over S values is $O(L-n)$. Therefore, the overall algorithm time complexity per pipeline template is $O\left((L-n)L^3nM\right)$.

Using memoization to reduce complexity. We cache all intermediate results to accelerate the divide and conquer algorithm. It boosts not only getting the mapping of one pipeline template but also helps in deriving the mapping of the other pipeline templates. In fact, running the algorithm for the largest pipeline template (with n_{p-1} nodes) is enough to calculate intermediate caches required for building the mapping of all the other pipeline templates. With all intermediate caches present, calculating the mapping of another smaller pipeline template can be done in $O(Ln)$.

4.2 Pipeline Instantiation

Given a set of pipeline templates, we know by construction (§4.1.1) that there exists a combination of them that Oobleck can instantiate to utilize all available nodes. However, such a combination of heterogeneous pipelines may not be unique. So we first find all such feasible combinations (§4.2.1). Oobleck chooses a plan with the highest estimated throughput among all the feasible combinations (§4.2.2).

4.2.1 Enumerating All Instantiation Options. Although we know that we can utilize all available nodes with the set of pipeline templates, the number of pipelines to be instantiated from each pipeline template is undetermined. Worse, there may be several pipeline configurations that use all nodes from the same pipeline template set. For example, 13 nodes can be represented as the plan 1 in Figure 4b ($1 \cdot n_0 + 1 \cdot n_1 + 2 \cdot n_2$), but also as ($0 \cdot n_0 + 3 \cdot n_1 + 1 \cdot n_2$), and more. We therefore enumerate all feasible pipeline sets for currently available nodes and pick one that maximizes training throughput.

Let $\mathbb{X}(p, N)$ be a list of all feasible pipeline sets $[\mathbf{X}_0, \mathbf{X}_1, ...]$. Each $\mathbf{X}_i$ is a set of the number of pipelines to be instantiated $(x_0, x_1, \ldots, x_{p-1})$, using p number of heterogeneous pipeline templates with the number of nodes specification $(n_0, n_1, \ldots, n_{p-1})$, so that all N nodes are used by pipelines. A feasible $\mathbf{X}_i$ satisfies the following requirements:

1. $N = x_0n_0 + x_1n_1 + \cdots + x_{p-1}n_{p-1}$ (All nodes are used).
2. $\sum_{j=0}^{p-1} x_j \geq f + 1$ (Number of pipelines is at least $f+1$).

We exploit dynamic programming for the coin change problem to find $\mathbb{X}$. The coin change problem finds a combination of coins that add up to the given amount of money [4]. It is an equivalent problem to Requirement 1 above if we replace denominations of each coin with n_i, and the given amount of money with N. We formulate the dynamic programming structure as:

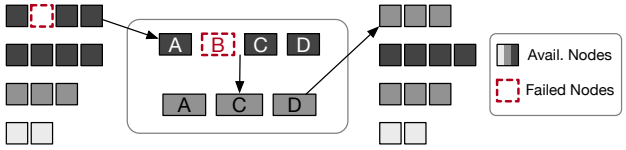
$$\mathbb{X}(p', N') = \mathbb{X}(p'-1, N') +\!\!+ \theta(\mathbb{X}(p', N' - n_{p'}), p') \quad (5)$$

where $+\!\!+$ means concatenating two lists, and $\theta(\mathbb{X}, p')$ is a function that increases $x_{p'}$ by 1 in every $\mathbf{X}_i$s in $\mathbb{X}$.

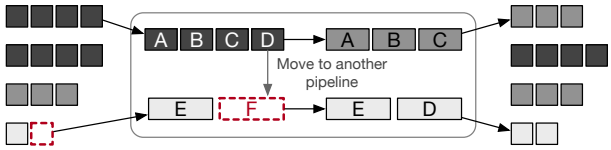
Figure 7 shows the execution of the dynamic programming algorithm. The two terms in Equation 5 are associated with each black boxes. $\mathbb{X}$ in the red box should include all $\mathbf{X}_i$s that use all seven nodes in instantiating pipelines using the three different pipeline templates ($n_0 = 2, n_1 = 3, n_2 = 4$). $\mathbb{X}$ in ①, all $\mathbf{X}_i$s use seven nodes and are already feasible for $\mathbb{X}(3, 7)$. We just copy them. $\mathbb{X}$ in ②, however, only uses three nodes in total. By adding one four-node pipeline (increasing x_2 by 1), all $\mathbf{X}_i$s use seven nodes and become feasible for $\mathbb{X}(3, 7)$.

The dynamic programming is done in $O(Np)$ filling all table elements. $\mathbb{X}$ in the bottom-right corner of the table contains all feasible $\mathbf{X}_i$s. To satisfy Requirement 2, we filter the list and obtain sets with $\sum_{j=0}^{p-1} x_j \geq f + 1$.

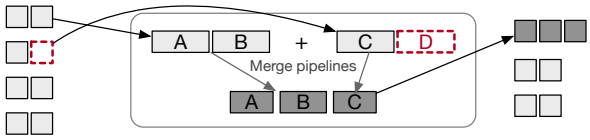
4.2.2 Calculating Throughput with Batch Distribution. Oobleck's execution engine needs to choose from several feasible $\mathbf{X}_i$s. We calculate the overall throughput for each $\mathbf{X}_i$ and choose the one that maximizes the throughput. To calculate throughput, we need to determine the batch size of each pipeline. While the global batch size is given by the user, it is Oobleck's responsibility to distribute them across heterogeneous pipelines to maximize overall throughput. It is crucial to assign work proportional to the amount of computing power of each pipeline; otherwise, the overall throughput will be decreased due to stragglers. We refer to this as *batch distribution*. Given the global batch size B and



(a) *A node failure in a 4-node pipeline. We have a 3-node pipeline template, thus a new pipeline with 3 nodes is instantiated, which replaces the existing one.*



(b) *A node failure in a 2-node pipeline. Since there is no template for one node, it gets another node from another pipeline to keep the 2-node pipeline. Two affected pipelines reinstantiate or reconfigure themselves.*



(c) *A node failure in a 2-node pipeline. Because it cannot borrow a node from any other pipeline, it is merged with another pipeline.*

Figure 8. *Three steps of pipeline reinstantiation. After reinstantiation is done, missing layers in a new pipeline are copied from other pipelines.*

microbatch size b, batch distribution calculates the number of microbatches for each pipeline that minimizes stragglers.

Let $N_{b,i}$ be the number of microbatches for i-th pipeline ($0 \le i < x,\ x = \sum_{j=0}^{p-1} x_j$) and T_i be the iteration time of the pipeline with a single microbatch of size b. Minibatch size for i-th pipeline can be calculated as $N_{b,i} \times b$. By adjusting $N_{b,i}$, we minimize variance between different pipelines' batch processing times. We formulate it as an integer optimization problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=0}^{x-1} (N_{b,i}T_i - \overline{N_bT})^2 \\ \text{subject to} \quad & \sum_{i=0}^{p-1} N_{b,i} b x_i = B, \\ & N_{b,i} \in \mathbb{N} \end{aligned} \tag{6}$$

where $\overline{N_bT}$ is the average iteration time of all ($0 \le i < x$) pipelines. Any integer nonlinear optimization solver can be used to get $N_{b,i}$ and thus minibatch size for each pipeline.

Note that the optimization may fail to redistribute batch properly, primarily when the global batch size is too small and cannot be split to integers. Oobleck does not change the global batch size arbitrarily in such cases. Instead, it recommends an adjusted global batch size close to the original one but distributable.

5 Dynamic Reconfiguration

Upon a node failure, the pipeline it was assigned to becomes incomplete and has missing model states; therefore, training halts in that pipeline. Pipelines affected by failures are replaced with new pipelines created via pipeline reinstantiation using precomputed pipeline templates (§5.1). After reinstantiating pipelines, the nodes copy missing layers from unaffected pipeline replicas. Oobleck also redistributes batch in response to the pipeline configuration change (§5.2). Provided that we have copies of the model states, Oobleck can recover from failures until we have fewer than $(f+1)n_0$ nodes.

5.1 Pipeline Reinstantiation

Oobleck instantiates a new pipeline from one of the pipeline templates, replacing the existing one affected by failures. Given our limited number of pipeline templates, there might not be a suitable pipeline template for the remaining number of nodes. Thus, pipeline reinstantiation is done in three steps: simple reinstantiation, borrowing nodes, and merging pipelines.

For each pipeline, Oobleck first checks if there is an instantiable pipeline template with remaining nodes; if so, Oobleck simply reinstantiates it and replaces the old one (Figure 8a). If there is no instantiable pipeline template with remaining nodes, Oobleck tries to *borrow nodes* from other pipelines until we have enough nodes to instantiate the smallest pipeline template (Figure 8b). Pipelines that yield their nodes should also be reinstantiated with fewer nodes.

After many reconfigurations and node borrowings, all pipelines may not be able to yield their nodes. When failures happen at this moment, the pipeline affected by failures cannot be reinstantiated due to a lack of nodes. In such a case, Oobleck *merges pipelines* to create a bigger pipeline (Figure 8c). It is guaranteed that we have an instantiable pipeline template for a merged pipeline if it has at least n_0 nodes (the minimum number of nodes to maintain one single pipeline). See Appendix B for a proof.

5.2 Batch Redistribution

After pipeline reinstantiation, execution configuration has been changed and distributed batches no longer ensure balanced execution. Oobleck runs Equation 6 again given the current set of the number of pipeline instances and continues training with a newly calculated batch size. Each pipeline may have more batches to compute, but the global batch size remains constant.

6 Implementation

We implement Oobleck in Python using PyTorch [23] and HuggingFace Transformers [51] using components from Merak [19] in using PyTorch fx symbolic tracer [41] to create pipelines and from DeepSpeed [40] to run them. For the

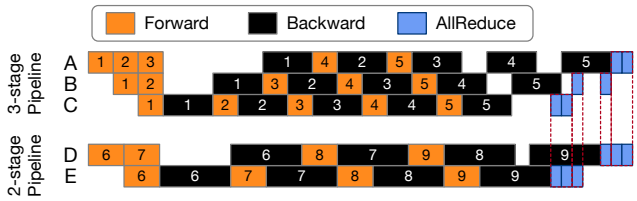


Figure 9. *Heterogeneous pipeline execution with one 3-stage pipeline and one 2-stage pipeline. Allreduce synchronization happens at the end of iteration and done in layer granularity to communicate between multiple GPUs.*

implementation of 3D parallelism, Oobleck has integrated PyTorch Fully Sharded Data Parallel (FSDP) [55] as a replacement for tensor parallelism in each stage [37], pipeline parallelism, and data parallelism. We have used Pyomo library and its Mixed-Integer Nonlinear Decomposition Toolbox (MindtPy) solver for non-linear integer optimization [6, 13]. Here, we elaborate on the key challenges addressed in implementing Oobleck.

6.1 Model Synchronization Between Heterogeneous Pipelines

Model gradient synchronization between pipelines in hybrid parallelism is typically done at pipeline stage granularity. However, because heterogeneous pipelines in Oobleck have different stage configurations, stage-wise synchronization does not work. Oobleck instead breaks down stages into layers and synchronizes them individually, similar to PyTorch bucketing [23]. Figure 9 illustrates an example of a 6-layer model execution with two heterogeneous pipelines. Stage E in the 2-stage pipeline has 3 layers to synchronize which are stored in stage B and C in the 3-stage pipeline. Oobleck performs synchronization for each individual layer with potentially different peer nodes.

Data synchronization in smaller data might have performance issue because it might not fully saturate the network. We overlap communication with computation to offset increased communication latency [23].

6.2 Detecting Node Failures

Oobleck uses NCCL for communication between GPUs. However, NCCL cannot detect unexpected communication channel disconnection and hangs when tries to communicate with the failed node until a timer expires. To detect a node failure immediately, we launch a CPU process on each node and establish a TCP connection to a centralized CPU process. When a node dies, a socket disconnection event is triggered and broadcasted for reconfiguration.

7 Evaluation

We evaluate the effectiveness of Oobleck on large DNN models with 340M to 6.7B parameters and compare it against both Bamboo and Varuna. We summarize the results as follows:

Table 1. *Model and batch configurations. * in Bamboo indicates the largest possible microbatch runnable in our evaluation environment. X means not runnable even with 1 microbatch size.*

	# Params	Global Batch	Microbatch Size		
			Bamboo	Varuna	Oobleck
BERT-Large [9]	340M	8192	4*	32	32
GPT-2 [36]	345M	8192	1*	32	32
GPT-3 Medium [5]	350M	8192	X	16	16
GPT-3 2.7b [5]	2.7B	1024	X	2	2
GPT-3 6.7b [5]	6.7B	1024	X	2	2

- Oobleck outperforms the state-of-the-art solutions by up to 13.9× when nodes fail more frequently and matches them as failures become less frequent (§7.2).
- Oobleck's benefits extend to real-world settings where nodes are out and join back following spot instance traces. It outperforms the rest by up to 9.1× on average (§7.3).
- Ablation studies show that Oobleck's one-time planning overhead is low and it has high GPU utilization (§7.4).

7.1 Experimental Setup

Cluster setup. We evaluate Oobleck using 30 NVIDIA A40 GPUs with 40GB GPU memory each. The GPUs are connected to each other via a 200Gbps Mellanox ConnectX-6 InfiniBand adaptor for communication.

Varuna requires a remote object storage to store checkpoints for fault tolerance. We deploy a distributed object storage that consists of 6 nodes with two Intel Xeon Gold 6330 CPUs with 28 cores each, 512GB CPU memory, a 4TB PCIe 4.0 NVMe drive, and a 200Gbps Mellanox ConnectX-6 InfiniBand adaptor, respectively. We use MinIO for distributed object storage software [27].

Baselines. We compare Oobleck to the following baselines:

- *Varuna* [1]: A resilient training framework based on automated parallel configuration and checkpoints [26].
- *Bamboo* [48]: A resilient training framework based on redundant computation without full restart [46].

Neither of them supports 3D parallelism; hence, Oobleck uses one GPU per node configuration to avoid its planner generating plans that cannot be implemented in our baselines.

Both works focus on utilizing spot instances, while Oobleck supports general fault tolerance including preemptions in spot instance environments. Spot instance environments have a unique mechanism of *preemption notification*; the system is notified prior to actual preemption happening. In our spot instance-based evaluation, all three frameworks leverage early warning. In general, however, there is no such notification.

For Varuna, we periodically perform synchronous checkpointing for every 10 iterations following their continuous checkpointing policy [1].

Table 2. *Throughput (samples/s) with different frequency of failures. Bamboo was not able to run any GPT-3 model due to lack of memory (OOM).*

Models	BERT-Large			GPT-2			GPT-3 Medium			GPT-3 2.7b			GPT-3 6.7b		
Failure Frequency	6h	1h	10m	6h	1h	10m	6h	1h	10m	6h	1h	10m	6h	1h	10m
Bamboo	77.04	75.60	69.84	17.47	17.13	16.01									
Varuna	260.11	246.03	173.52	86.51	85.17	76.39	29.50	28.49	20.19	7.27	6.53	1.76	4.14	2.69	0.26
Oobleck	287.10	286.28	282.11	85.59	85.42	84.80	29.30	29.21	28.70	7.29	7.23	6.89	4.33	4.22	3.55

Workloads. Table 1 lists model configurations. We adopt GPT-2 and BERT-Large from Bamboo and Varuna, and add three different configurations of GPT-3 from OpenAI [5] to verify its scalability. Although our evaluation only shows transformer models for comparison against two predecessors, Oobleck's design is not limited to transformer language models and can support other DNN models. For all evaluations, we use the Wikitext dataset [25] and TF32 precision.

We also list batch size configurations for each framework in Table 1. The reasons behind the discrepancy in batch sizes are twofold. First, Bamboo needs to store additional model states for redundant computation, requiring 2× memory. Second, Bamboo does not use activation checkpointing,[2] while Varuna and Oobleck do.

7.2 Throughput Under Controlled Failures

We first evaluate the average throughput of Bamboo, Varuna, and Oobleck on various failure scenarios. We set the frequency of failures from once every 6 hours (low rate) to once every 10 minutes (high rate) to cover a wide spectrum of environments [1, 8, 20, 44, 48]. We monotonically reduce the number of available nodes without node recovery and measure average throughput until less than half of the nodes (15 nodes) remain.

Table 2 shows the average throughput for different frequencies of failures. Oobleck outperforms or matches other frameworks for every model in every scenario. Due to static overhead coming from redundant computation, Bamboo's throughput, while stable over different failure frequencies, is consistently low. Also, because they need to hold a large portion of GPU memory for an additional copy of model states for redundant computation and activations, Bamboo cannot train large models due to out-of-memory (OOM) errors.

Bamboo's gap from Varuna was surprising. We believe that it is due to differences in our evaluation environments. We use 200Gbps high-performance networking and ample NVMe storage, while the original evaluation used Amazon EC2 `p3.2xlarge` and `p3.8xlarge` instances with up to 10Gbps network and S3 object storage. High storage throughput in our setup significantly sped up Varuna. Furthermore, we use different batch configurations for Bamboo and Varuna so that each can run at maximum resource utilization; in contrast, Bamboo's evaluation used the same configuration for both, which throttled Varuna's potential throughput.

Overall, Varuna performs comparably to us when either the model is small or failures happen infrequently. For larger models and/or more frequent failures, the higher overhead of loading and saving checkpoints drastically decrease its throughput (13.9× for GPT3-6.7B).

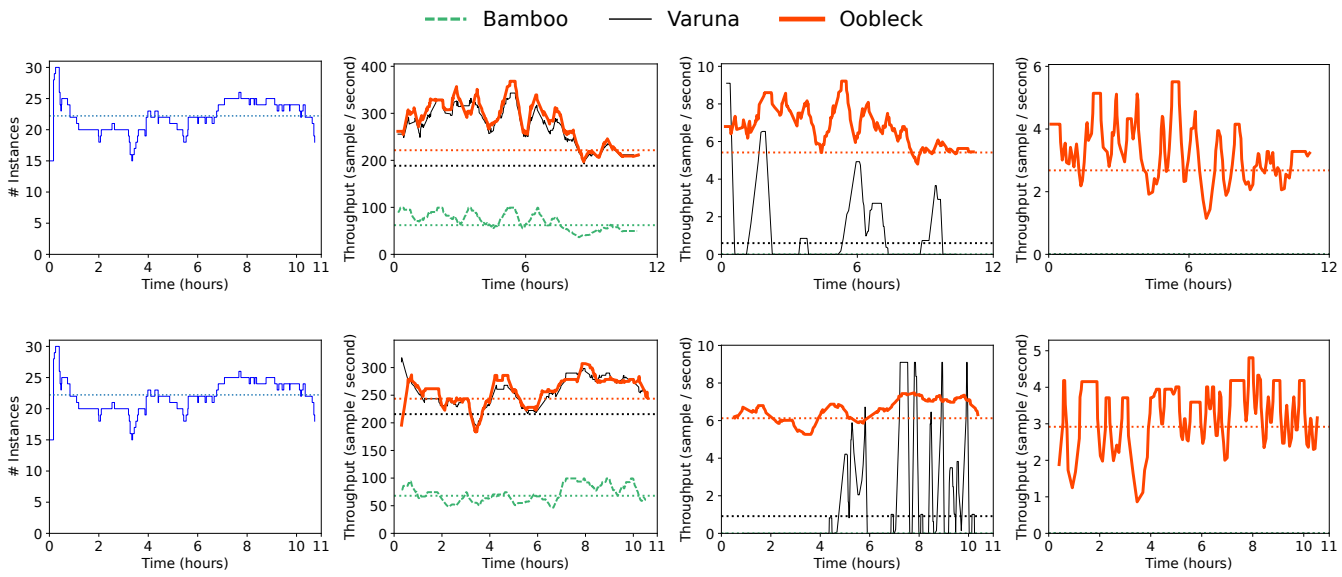
7.3 Throughput in Spot Instance Traces

Next, we borrow real traces of node availability changes of spot instances from the Bamboo repository [46] and use their tools to replay the trace for 12 hours [48]. Events in the trace had been gathered from Amazon EC2 `P3` spot instances (`p3.2xlarge` and `p3.8xlarge`) and Google Cloud Platform (GCP) `a2-highgpu-1g` spot instances. Node preemption events happen every 7.7 minutes and 10.3 minutes, on average, for EC2 and GCP spot instances, respectively. Unlike experiments earlier where the number of available nodes monotonically decreases (§7.2), these traces include node addition events too. The actual experiments took place in our cluster where we simulated the availability events.

Figure 10 represents throughput changes for some models. See Appendix C for results from other models; omitted models are similar to the BERT-Large model. Note that each data point in lines is an average throughput of a short time window for visibility. As such, it may not represent 0 throughput, which happens during reconfiguration or full restart. BERT-Large, the smallest model, has the least amount of checkpointing overhead; as such, the performance of Varuna is similar to Oobleck. However, as models become larger, even in our high-performance storage setup, Varuna takes increasingly longer to store and load checkpoints. Also, it starts suffering from fallbacks because it fails to finish checkpointing within the preemption grace period, decreasing its throughput more drastically. For example, for GPT-3 with 6.7 billion parameters, Varuna hung over the entire time and could not make training progress. Frequent changes in node availability trigger Varuna's full reconfiguration more frequently, wasting resources for saving and loading checkpoints which decreases its throughput.

Bamboo cannot run any model larger than GPT-2.

[2] This is because Bamboo's design choice stems from imbalanced memory consumption due to different amount of activations across stages. Activation checkpointing [7] drastically reduces memory consumption by activations and it conflicts with Bamboo's design.



(a) *Node availability* **(b)** *BERT-Large* **(c)** *GPT-3 2.7b* **(d)** *GPT-3 6.7b*

Figure 10. *Throughput changes in spot instances environment, EC2* `P3` *instances (top) and GCP* `a2-highgpu-1g` *instances (bottom), with various models. Note the different Y-axes scales for different models. Horizontal dotted lines represent average throughput. Bamboo could not run any GPT-3 models, while Varuna failed for GPT-3 6.7b.*

Table 3. *Oobleck planning latency (in seconds) with various numbers of layers and nodes. BERT-Large, GPT-2, and GPT-3 Medium have 24 layers, while GPT-3 2.7b and 6.7b have 32 layers.*

# Nodes	# GPUs Per Node	# Layers 24	32	64	96
8	1	0.28	0.71	9.65	68.50
	4	0.41	1.15	11.58	74.56
	8	0.54	1.50	20.98	109.76
16	1	3.37	7.45	66.35	540.36
	4	4.56	10.41	108.10	649.67
	8	4.90	11.78	176.04	1,213.63
24	1	11.35	30.11	262.47	1,477.54
	4	14.78	45.80	472.53	2,153.84
	8	15.59	49.25	520.08	3,297.92

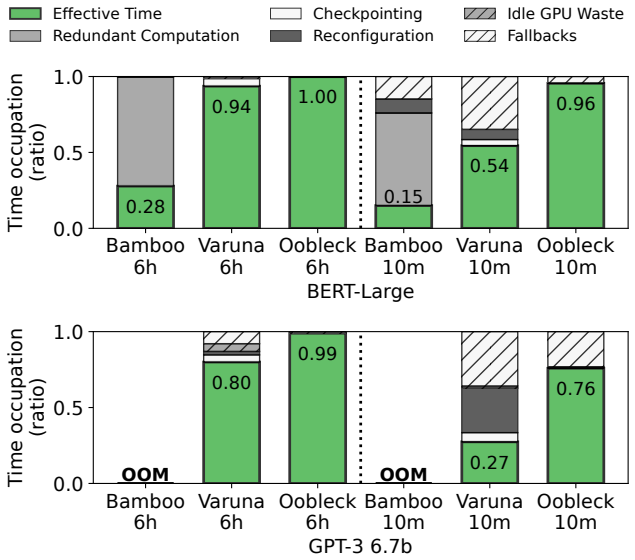


Figure 11. *Time occupation breakdown of Bamboo, Varuna, and Oobleck running BERT-Large and GPT-3 6.7b model.*

7.4 Ablation Study

7.4.1 Overhead of Oobleck Planning. We run the planning algorithm (§4) to create a single pipeline template using various model specifications (number of layers) and node specifications (number of nodes and GPUs per node) to see its scalability. Table 3 shows the planning algorithm latency with various numbers of layers and nodes. Considering the estimated end-to-end training time of large models using hundreds of GPUs is ~100 days [5, 30], the planning overhead is marginal (< 0.1%). Even if more GPUs are used for training (i.e., thousands of GPUs), the number of nodes for each pipeline template does not increase significantly. This is because Oobleck simply instantiates more of the smaller pipelines and utilizes data parallelism. Also, once a pipeline template is generated, the creation of subsequent templates can drastically be accelerated, adding negligible time, thanks to the usage of memoization and intermediate caches (§4.1.2).

Table 4. *Throughput (samples/s) for Varuna, Varuna with no checkpointing overhead, and Oobleck running BERT-Large and GPT-3 6.7b.*

	BERT-Large			GPT-3 6.7b		
Failure Frequency	6h	1h	10m	6h	1h	10m
Varuna	260.11	246.03	173.52	4.14	2.69	0.26
Varuna (no ckpt)	275.88	273.85	264.16	4.61	4.29	1.98
Oobleck	287.10	286.28	282.11	4.33	4.22	3.55

7.4.2 Throughput Breakdown. Figure 11 shows the impact of each overhead on training throughput. Varuna's overheads include restarting overhead (reinitialization and loading a checkpoint), saving checkpoints, and throughput loss due to idle GPUs and fallbacks. Bamboo has significant overhead from redundant computation and reconfiguration overhead for data copy. Redundant computation in Bamboo is shown to have more than 50% overhead because it includes several indirect components that cannot clearly be separated – for example, pipeline bubble due to an increased number of pipeline stages to store redundant model states and imbalanced pipeline stages for balancing memory. Oobleck only has a small copying overhead for missing layers after pipeline reinstantiation.

All the frameworks experience fallback overhead, losing some training progress due to failures happening in the middle of iteration. It is more severe in Varuna because it has to fall back to the last checkpoint, while Bamboo and Oobleck lose at most one iteration. Varuna suffers significantly from much such waste occupying up to 75% of wall clock time, while Oobleck can achieve effective throughput of at least 75% of the no-failure scenario.

7.4.3 Impact of Checkpointing Overhead. Overhead of fault tolerance in Varuna mostly comes from serialized checkpointing and full restart. CheckFreq [28] recently introduced checkpointing optimization by pipelining checkpointing with computation, and it can improve the throughput of checkpoint-based training. Here, we go further by *completely* removing the overhead of checkpointing and analyzing the impact of failures only. Because of lower checkpointing overhead, we also increase the frequency of checkpointing from every 10 iterations to every 2 iterations. We define full restart overhead as framework initialization plus loading the last checkpoint overhead. While checkpointing overhead during training can be hidden by overlapping it with computation, the overhead of loading a checkpoint cannot be overlapped with computation; this is because computation cannot begin until the entire checkpoint is loaded.

Table 4 compares Varuna, Varuna with no checkpointing overhead, and Oobleck running the BERT-Large model and GPT-3 6.7b model. Although Varuna could increase its throughput, it still suffers from up to 60% overhead for the higher frequency of failures.

8 Related Works

Elastic training. Horovod Elastic [14] and TorchElastic [34] restart training upon failure and recovery. CoDDL [16] balances resource efficiency and short job priority in elastic resource sharing problems. Aryl [21] enables elastic resource sharing between inference and training workloads. Pollux [35] considers both resource utilization and statistical efficiency of training jobs when adaptively allocating resources. These works are all limited to elastic resource sharing for data-parallel training of small models that fit within a single GPU.

Distributed training with spot instances. Varuna [1] uses hybrid parallelism for distributed training with cheaper spot cloud instances. It reconfigures training when one or more failures happen. Bamboo [48] introduces redundant computation (RC) in pipeline parallelism to provide resilience in the presence of frequent preemptions of training with spot instances. Oobleck matches or significantly outperforms them for a wide range of model sizes and failure frequencies.

Large model training. Numerous proposals in recent years have attempted to optimize large model training through diverse mechanisms [11, 15, 22, 29, 30, 33, 37, 38, 42, 45, 47, 56]. However, they do not provide fault tolerance out-of-the-box and are orthogonal to Oobleck.

9 Conclusion

In this paper, we introduced Oobleck, a resilient distributed Large model training framework with guaranteed fault tolerance. Oobleck co-designs planning and execution for fast failure recovery and high throughput by introducing pipeline templates that are carefully designed during planning and reused during training execution. It achieves efficient failure recovery by reinstantiating pipeline(s) from the pipeline templates and copying missing model states from pipeline replicas without requiring a full restart from checkpoints. Oobleck outperforms state-of-the-art fault-tolerant distributed training solutions Bamboo and Varuna by up to 13.9×.

Acknowledgements

We would like to thank the SOSP reviewers, our shepherd Keval Vora, and SymbioticLab members for their insightful feedback. This work is in part supported by NSF grants CNS-1909067, CNS-2104243, and CNS-2106184 as well as gifts from VMWare, Google, and Meta.

References

[1] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3492321.3519584

[2] Romain Beaumont. 2022. Large Scale OpenCLIP: L/14, H/14 AND G/14 Trained on LAION-2B. https://laion.ai/blog/large-openclip/

[3] Stas Bekman. 2022. The Technology Behind Bloom Training. https://huggingface.co/blog/bloom-megatron-deepspeed

[4] Richard Bellman. 1952. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences (PNAS)*. https://doi.org/10.1073/pnas.38.8.716

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper_files/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[6] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. 2021. *Pyomo–Optimization Modeling in Python*. Springer Science & Business Media. https://doi.org/10.1007/978-3-030-68928-5

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG]

[8] Anwesha Das, Frank Mueller, and Barry Rountree. 2021. Systemic Assessment of Node Failures in HPC Production Platforms. In *International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS49936.2021.00035

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. https://doi.org/10.18653/v1/N19-1423

[10] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi22/presentation/eisenman

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3437801.3441593

[12] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in Large Scale Systems: Long-Term Measurement, Analysis, and Implications. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. https://doi.org/10.1145/3126908.3126937

[13] William E Hart, Jean-Paul Watson, and David L Woodruff. 2011. Pyomo: Modeling and Solving Mathematical Programs in Python. *Mathematical Programming Computation (MPC)* (2011). https://doi.org/10.1007/s12532-011-0026-8

[14] Horovod. 2019. Elastic Horovod. https://horovod.readthedocs.io/en/stable/elastic_include.html

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper_files/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html

[16] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi21/presentation/hwang

[17] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc19/presentation/jeon

[18] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc22/presentation/jia-xianyan

[19] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. 2023. Merak: An Efficient Distributed DNN Training Framework With Automated 3D Parallelism for Giant Foundation Models. *Transactions on Parallel and Distributed Systems (TPDS)* (2023). https://doi.org/10.1109/TPDS.2023.3247001

[20] Sungjae Lee, Jaeil Hwang, and Kyungyong Lee. 2022. SpotLake: Diverse Spot Instance Dataset Archive Service. In *International Symposium on Workload Characterization (IISWC)*. https://doi.org/10.1109/IISWC55918.2022.00029

[21] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2022. Aryl: An Elastic Cluster Scheduler for Deep Learning. arXiv:2202.07896 [cs.DC]

[22] Shigang Li and Torsten Hoefler. 2021. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3458817.3476145

[23] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of VLDB Endowment (VLDB)* (2020). https://doi.org/10.14778/3415478.3415530

[24] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2022. Persia: An Open, Hybrid System Scaling Deep Learning-Based Recommenders up to 100 Trillion Parameters. In *International Conference on Knowledge Discovery and Data Mining (KDD)*. https://doi.org/10.1145/3534678.3539070

[25] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=Byj72udxe

[26] Microsoft. 2022. Varuna. https://github.com/microsoft/varuna

[27] MinIO. 2023. MinIO: High Performance Object Storage for AI. https://github.com/minio/minio

[28] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *USENIX Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast21/presentation/mohan

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3341301.3359646

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and

Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. https://doi.org/10.1145/3458817.3476209

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091 [cs.IR]

[32] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[33] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc20/presentation/park

[34] PyTorch. 2020. Torch Elastic. https://pytorch.org/elastic/latest/

[35] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi21/presentation/qiao

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language Models are Unsupervised Multitask Learners. https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/SC41405.2020.00024

[38] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. https://doi.org/10.1145/3458817.3476205

[39] Jorge L. Ramírez Alfonsín. 2005. *The Diophantine Frobenius Problem*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780198568209.001.0001

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *International Conference on Knowledge Discovery and Data Mining (KDD)*. https://doi.org/10.1145/3394486.3406703

[41] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *Proceedings of Machine Learning and Systems (MLSys)*. https://proceedings.mlsys.org/paper_files/paper/2022/hash/7c98f9c7ab2df90911da23f9ce72ed6e-Abstract.html

[42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc21/presentation/ren-jie

[43] J. B. Roberts. 1956. Note on Linear Forms. *Proc. Amer. Math. Soc.* (1956). https://doi.org/10.2307/2032755

[44] Bianca Schroeder and Garth A. Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *Transactions on Dependable and Secure Computing (TDSC)* (2010). https://doi.org/10.1109/TDSC.2009.4

[45] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. arXiv:2201.11990 [cs.CL]

[46] UCLA System. 2023. Bamboo. https://github.com/uclasystem/bamboo

[47] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper_files/paper/2021/hash/d01eeca8b24321cd2fe89dd85b9beb51-Abstract.html

[48] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi23/presentation/thorpe

[49] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. 2022. Machine Learning Model Sizes and the Parameter Gap. arXiv:2207.02852 [cs.LG]

[50] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi22/presentation/weng

[51] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[52] 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 *International Conference on Distributed Computing Systems (ICDCS)*. https://doi.org/10.1109/ICDCS47774.2020.00018

[53] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

[54] Tianyu Zhang, Kaige Liu, Jack Kosaian, Juncheng Yang, and Rashmi Vinayak. 2023. Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. *Proceedings of the VLDB Endowment (VLDB)* (2023). https://doi.org/10.14778/3611479.3611514

[55] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arXiv:2304.11277 [cs.DC]

[56] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin