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
xinjinpk@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 29.6×.

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

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 $29\times$ 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.

- 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
Varuna provides higher training throughput compared to overheads dominate. Additionally, performance degradation [10, 28], loading checkpoints upon restarts is still in the recent proposals improved checkpointing overhead Bamboo, but it has noticeable job restart overhead. Although 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
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 \( b \) (Figure 3). Oobleck’s pipeline template generator first creates a set of pipeline templates \( \mathfrak{G} \). The distributed execution engine instantiates pipelines from the templates and deploys them on the cluster \( \mathfrak{L} \).

4 Oobleck Planning Algorithm

Oobleck tolerates \( f \) simultaneous failures by instantiating \( r \) (\( r \geq f + 1 \)) 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 \( \times \) pipeline stages \( \times \) 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).
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 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 \( n_{p-1}^{\text{max}} \)). When all but one of the \( f+1 \) replicas use \( n_0 \) nodes and the last one uses all the remaining nodes, \( n_{p-1}^{\text{max}} = N - fn_0 \). We now have \( p \) to be the length of the range from \( n_0 \) to \( n_{p-1}^{\text{max}} \).

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 \((l_0, l_{u+1}, \ldots, l_{u-1})\) 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.
To calculate the minimum iteration time of a pipeline, the algorithm considers its critical path and breaks $T$ down into three terms $T_1$, $T_2$, and $T_3$ (Figure 5). $T_1$ 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 $T_1$. We thus split the steady phase into $T_2$ and $T_3$. $T_2$ includes the slowest stage alternating forward and backward, and $T_3$ 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 $T_1$, $T_2$, and $T_3$, the division and combination process can be defined as recursive structures for each term:

$$T_{1s,k,m}(s', u, v, d) = T_{1s,k,m}(s, u, k, m) + T_{1s,k,m}(s' - s, k + 1, v, d - m)$$  \hspace{1cm} (1)

$$T_{2s,k,m}(s', u, v, d | k^*) = (N_b - S' + k^* - 1)(F_{s_k,m} + B_{s_k,m})$$  \hspace{1cm} (2)

$$T_{3s,k,m}(s', u, v, d | k^*) = \begin{cases} T_{3s,k,m}(s, u, k, m | k_1^*) + T_{1s,k,m}(s' - s, k + 1, v, d - m) & \text{if } k^* = k_1^* \\ T_{3s,k,m}(s', u, k, m | k_2^*) & \text{else} \end{cases}$$  \hspace{1cm} (3)

We iterate over $s$, $k$, and $m$ globally, and find a $(s, k, m)$ that minimizes $T_{1s,k,m} + T_{2s,k,m} + T_{3s,k,m}$. Each $T_{1s,k,m}$, $T_{2s,k,m}$, and $T_{3s,k,m}$ is the solution of $T_1$, $T_2$, and $T_3$, 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. $T_2$ depends on the number of microbatches ($N_b$) deployed to the pipeline, which is not yet determined. From 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 $T_3$ is special as it includes $T_1$ in Equation 3. $T_3$ 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. $s_k^* = S_k^*$), it can be broken down to $\sum_{k=k'}^{S-1}(F + B) + \sum_{k=S+1}^{S-1}(F + B)$, each of which represents $T_3$ of the first half and $T_1$ 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 $l_u$, ..., $l_{e-1}$ layers on $d$ GPUs:

$$T_1(1, u, v, d) = F_{s,d} + B_{s,d} = \sum_{k=u}^{v-1}(F_{l_k,d} + B_{l_k,d})$$

$$T_2(1, u, v, d) = 2(F_{s,d} + B_{s,d})$$

$$T_3(1, u, v, d) = F_{s,d} + B_{s,d}$$

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 $\infty$ 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
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 $O((L - n)L^3nM)$.

**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 $\mathcal{X}(p, N)$ be a list of all feasible pipeline sets $[X_0, X_1, ...]$. Each $X_i$ is a set of the number of pipelines to be instantiated $(x_0, x_1, ..., x_{p-1})$, using $p$ number of heterogeneous pipeline templates with the number of nodes specification $(n_0, n_1, ..., n_{p-1})$, so that all $N$ nodes are used by pipelines. A feasible $X_i$ satisfies the following requirements:

1. $N = \sum x_i \cdot n_i$ (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 $\mathcal{X}$. 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:

$$\mathcal{X}(p', N') = \mathcal{X}(p' - 1, N') + \theta(\mathcal{X}(p', N' - n_{p'}), p')$$  \hspace{1cm} (5)$$

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

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

The dynamic programming is done in $O(Np)$ filling all table elements. $\mathcal{X}$ in the bottom-right corner of the table contains all feasible $X_i$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 $X_i$s. We calculate the overall throughput for each $X_i$ 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
we minimize variance between different pipelines’ batch distribution. Let $N_{b,i}$ be the number of microbatches for $i$-th pipeline ($0 \leq i < x$, $x = \sum_{j=0}^{p-1} x_j$) 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:

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=0}^{x-1} (N_{b,i}T_i - N_iT)^2 \\
\text{subject to} & \quad \sum_{i=0}^{p-1} N_{b,i}bX_i = B, \\
& \quad N_{b,i} \in \mathbb{N}
\end{align*}
\]

where $N_iT$ is the average iteration time of all $0 \leq i < x$ 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 batches 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 reinstatement, 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 reinstatiates 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 reinstated 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 reinstated 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 reinstatement, 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
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:

- Oobleck outperforms the state-of-the-art solutions by up to 29.6× 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>
<thead>
<tr>
<th># Params</th>
<th>Global Batch</th>
<th>Microbatch Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-Large [9] 340M</td>
<td>8192</td>
<td>4*</td>
</tr>
<tr>
<td>GPT-2 [36]</td>
<td>345M</td>
<td>8192</td>
</tr>
<tr>
<td>GPT-3 Medium [5] 350M</td>
<td>8192</td>
<td>X</td>
</tr>
<tr>
<td>GPT-3 2.7b [5]</td>
<td>2.7B</td>
<td>1024</td>
</tr>
<tr>
<td>GPT-3 6.7b [5]</td>
<td>6.7B</td>
<td>1024</td>
</tr>
</tbody>
</table>

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.
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).

<table>
<thead>
<tr>
<th>Models</th>
<th>BERT-Large</th>
<th>GPT-2</th>
<th>GPT-3 Medium</th>
<th>GPT-3 2.7b</th>
<th>GPT-3 6.7b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure Frequency</td>
<td>6h</td>
<td>1h</td>
<td>10m</td>
<td>6h</td>
<td>1h</td>
</tr>
<tr>
<td>Bamboo</td>
<td>77.04</td>
<td>75.60</td>
<td>69.84</td>
<td>17.47</td>
<td>17.13</td>
</tr>
<tr>
<td>Varuna</td>
<td>259.57</td>
<td>245.39</td>
<td>168.15</td>
<td>86.42</td>
<td>83.94</td>
</tr>
<tr>
<td>Oobleck</td>
<td>287.10</td>
<td>286.28</td>
<td>282.11</td>
<td>85.59</td>
<td>85.42</td>
</tr>
</tbody>
</table>

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 (29.6× for GPT-3-6.7B).

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.
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.

<table>
<thead>
<tr>
<th># Nodes</th>
<th># GPUs Per Node</th>
<th># Layers</th>
<th>Effective Time</th>
<th>Checkpointing</th>
<th>Idle GPU Waste</th>
<th>Redundant Computation</th>
<th>Reconfiguration</th>
<th>Fallbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>24</td>
<td>0.20</td>
<td>0.94</td>
<td>OOM</td>
<td>0.99</td>
<td>0.08</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>64</td>
<td>1.20</td>
<td>0.15</td>
<td>OOM</td>
<td>1.00</td>
<td>0.66</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>96</td>
<td>2.13</td>
<td>0.06</td>
<td>OOM</td>
<td>1.00</td>
<td>OOM</td>
<td>0.78</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>24</td>
<td>11.35</td>
<td>0.20</td>
<td>68.50</td>
<td>9.65</td>
<td>66.35</td>
<td>109.76</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>64</td>
<td>4.56</td>
<td>0.94</td>
<td>540.36</td>
<td>7.45</td>
<td>108.10</td>
<td>649.67</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>96</td>
<td>4.90</td>
<td>1.00</td>
<td>1,213.63</td>
<td>11.78</td>
<td>176.04</td>
<td>1,213.63</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>24</td>
<td>14.78</td>
<td>0.20</td>
<td>1,477.54</td>
<td>30.11</td>
<td>262.47</td>
<td>1,477.54</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>64</td>
<td>15.59</td>
<td>0.94</td>
<td>OOM</td>
<td>45.80</td>
<td>472.53</td>
<td>2,153.84</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>96</td>
<td>26.24</td>
<td>1.00</td>
<td>OOM</td>
<td>49.25</td>
<td>520.08</td>
<td>3,297.92</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>BERT-Large</th>
<th>GPT-3 6.7b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure Frequency</td>
<td>6h 1h 10m</td>
<td>6h 1h 10m</td>
</tr>
<tr>
<td>Varuna</td>
<td>259.57</td>
<td>4.02</td>
</tr>
<tr>
<td>Varuna (no ckpt)</td>
<td>254.39</td>
<td>2.91</td>
</tr>
<tr>
<td>Oobleck</td>
<td>245.78</td>
<td>0.36</td>
</tr>
</tbody>
</table>

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 95% 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 90% overhead for the higher frequency of failures.

8 Related Works


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 29.6x.

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-1909667, CNS-2104243, and CNS-2106184 as well as gifts from VMware, Google, and Meta.

References


Appendix

A Proof of Nodes Specification Covering All Nodes

We prove the following theorem which shows a finite set of p number of pipeline templates, where the number of nodes is (n_0, n_1, ..., n_p) (n_i < n_{i+1}), can fully cover the node cluster with feasible number of nodes N' any time irrespective of how many failures happen on the cluster as long as its feasibility holds.

**Theorem A.1.** N' nodes ((f+1)n_0 ≤ N' ≤ N) can always be represented as a linear combination of the p pipeline templates with (n_0, n_1, ..., n_p) number of nodes, respectively, if the following two conditions are satisfied:

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

**Proof.** We first formulate an integer linear combination to represent N':

\[ N' = x_0n_0 + x_1n_1 + \ldots + x_{p-1}n_{p-1} \]  

(7)

where \( x_i \) is the number of pipelines to be instantiated from the pipeline template with \( n_i \) number of nodes.

The Frobenius number \( g(n_0, n_1, ..., n_{p-1}) \), the largest number that cannot be represented as a linear combination of Equation 7, has proven to be:

\[ g = \left( \left\lfloor \frac{n_0 - 2}{p - 1} \right\rfloor \right) + d(n_0 - 1) \]  

(8)

if the integer set \( (n_0, n_1, ..., n_{p-1}) \) is an arithmetic sequence, i.e., \( n_i = n_0 + d(i - 1) \) [39].

When we apply both Requirements 1 and 2, \( g = n_0 - 1 \). Fault tolerance threshold \( f \) is a non-negative integer; the minimum feasible number of nodes \( N' = (f+1)n_0 \) is \( n_0 \).

Therefore, any feasible \( N' \) that is larger than \( g \) and can be represented as a linear combination of Equation 7.

B Proof of Guarantee for Pipeline Template Availability When Merging Pipelines

We first show this when failures happen in a single pipeline. When we lose \( k \) nodes (\( k > 0 \)) from a pipeline, where all pipelines have \( n_0 \) nodes and are not able to yield any node, Oobleck instantiates a new pipeline with \( 2n_0 - k \) nodes by merging it with another \( n_0 \)-node pipeline. We prove that a pipeline template with \( 2n_0 - k \) nodes is always available.

**Theorem B.1.** A set of pipeline templates always includes a pipeline template with \( 2n_0 - k \) nodes (\( 2n_0 - k \geq n_0 \)).

**Proof.** A set of pipeline templates has pipeline templates with up to \( N - f n_0 \) nodes (§4.1.1). Assume that we do not have a pipeline template with \( 2n_0 - k \) number of nodes specification, then \( N - f n_0 < 2n_0 - k \) and \( N < (f + 2)n_0 - k \) are assumed to be true. To not break the fault tolerance threshold that we maintain at least \( f + 1 \) model replicas after merging two pipelines, we must have at least \( f + 2 \) replicas, i.e., we should have had at least \( (f + 2)n_0 \) nodes before failures. Since the initial number of nodes \( N \) is always larger than the number of currently remaining nodes, \( N > (f + 2)n_0 \) inequality holds and it contradicts our initial assumption. Therefore, we have a pipeline template with \( 2n_0 - k \) number of nodes.

When failures happen across several pipelines, multiple pipelines can have less than \( n_0 \) nodes. Oobleck repeatedly merges two pipelines until a new pipeline has enough number of nodes. Assume we merged \( m \) pipelines to get enough number of nodes, i.e., \( \sum_{i=0}^{m} n_{p_i} \geq n_0 \). It means merging \( m - 1 \) pipelines was not enough to get \( n_0 \) nodes, i.e., \( \sum_{i=0}^{m-1} n_{p_i} < n_0 \).

With \( n_{p_m} \leq n_0 \), we have an inequality \( n_0 \leq \sum_{i=0}^{m-1} n_{p_i} + n_{p_m} < 2n_0 \). It has already been proved by Theorem B.1 that we have a pipeline template for all numbers in the range.

C Throughput of All Models in Spot Instances

Figure 12 shows throughput of unreported models in the paper due to lack of space, running on Amazon EC2 P3 spot instances and Google a2-highgpu-1g spot instances. Varuna could avoid fallback overhead by successfully checkpointing ahead of preemption in small models, (e.g., BERT-large, GPT-2, and GPT-3 medium), thus Varuna throughput matches Oobleck on average. However, in large models, Oobleck outperforms it. Note that lines are smoothed for visibility and do not precisely represent throughput. Varuna has more spikes down to 0 throughput in reality thus has less throughput.