
# Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances

Jiangfei Duan, *The Chinese University of Hong Kong;* Ziang Song, *ByteDance;* Xupeng Miao and Xiaoli Xi, *Carnegie Mellon University;* Dahua Lin, *The Chinese University of Hong Kong;* Harry Xu, *University of California, Los Angeles;* Minjia Zhang, *Microsoft;* Zhihao Jia, *Carnegie Mellon University*

https://www.usenix.org/conference/nsdi24/presentation/duan

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

# Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances

Jiangfei Duan[‡♠] Ziang Song[§♠] Xupeng Miao[†♠] Xiaoli Xi[†]
Dahua Lin[‡] Harry Xu[♯] Minjia Zhang[◇] Zhihao Jia[†]

*Carnegie Mellon University*[†] *The Chinese University of Hong Kong*[‡]
*ByteDance*[§] *University of California, Los Angeles*[♯] *Microsoft*[◇]

## Abstract

Deep neural networks (DNNs) are becoming progressively large and costly to train. This paper aims to reduce DNN training costs by leveraging preemptible instances on modern clouds, which can be allocated at a much lower price when idle but may be preempted by the cloud provider at any time. Prior work that supports DNN training on preemptive instances employs a *reactive* approach to handling instance preemptions and allocations after their occurrence, which only achieves limited performance and scalability.

We present Parcae, a system that enables cheap, fast, and scalable DNN training on preemptible instances by *proactively* adjusting the parallelization strategy of a DNN training job to adapt to predicted resource changes before instance preemptions and allocations really happen, which significantly reduces the cost of handling these events. Parcae optimizes *liveput*, a novel metric that measures the *expected* training throughput of a DNN job under various possible preemption scenarios. Compared to existing reactive, throughput-optimized systems, Parcae's proactive, live-optimized solution considers both the throughput of a job and its robustness under preemptions. To optimize liveput, Parcae supports lightweight instance migration and uses an availability predictor to forecast future preemptions. It then uses a liveput optimizer to discover an optimal strategy to parallelize DNN training under predicted preemptions. We evaluate Parcae on a variety of DNNs and preemption traces and show that Parcae outperforms existing spot-instance DNN training systems by up to 10×. More importantly, Parcae achieves near-optimal performance for training large DNNs under frequent preemptions, in which case existing approaches cannot make any progress.

## 1 Introduction

Deep neural networks (DNNs) have surpassed human predictive performance on a spectrum of tasks, including computer vision [18], natural language progressing [14], game playing [44], and content generation [46]. The success of DNNs is associated with progressively increasing energy and financial costs. For example, a single training run of GPT-3 [12], a language model with 175 billion parameters, requires more than 1.5 million GPU hours and costs $4.6 million to train on AWS even with the lowest priced GPUs [37]. While pre-trained models are publicly available and can be fine-tuned for different downstream tasks, training new models is often required for emerging applications and datasets.

Modern cloud platforms provide a variety of cheap *preemptible instances*, which can be leveraged to minimize the monetary cost of DNN training. First, spot GPU instances allow users to take advantage of unused GPU capacity at a price up to 90% lower than on-demand counterparts [1]. Second, modern data centers generally reserve additional GPU capacity for urgent jobs, which can be allocated by other jobs in a preemptible manner [35]. Third, some ML systems [51] support opportunistically running training jobs on inference-dedicated GPUs to maximize resource utilization and preempt these training jobs when inference requests arrive. While this paper focuses on spot GPUs, our techniques can easily generalize to other preemptible resources.

Existing systems that support DNN training on spot instances use a *reactive* approach to handling instance preemption and allocation, and can be categorized into two classes: *checkpoint-* and *redundancy-based* systems. We introduce the two categories and identify the limitations of these reactive approaches in *performance* and *scalability* when applied to DNN training on preemptible instances.

The first line of work uses *checkpoints* to maintain model states during training. For example, Varuna [8] periodically saves model states to persistent storage and loads the latest checkpoint back after a preemption, as shown in Figure 1c. Although Varuna offers promising training throughput when spot instances have low preemption rates, it struggles to make progress when preemptions are frequent. This is due to two reasons: (1) saving and loading checkpoints incur significant IO overhead, particularly as model size increases, making frequent checkpointing costly, and (2) high preemption rates cause training to frequently roll back to the last saved check-

♠ Contributed equally. Work done during internships at CMU.

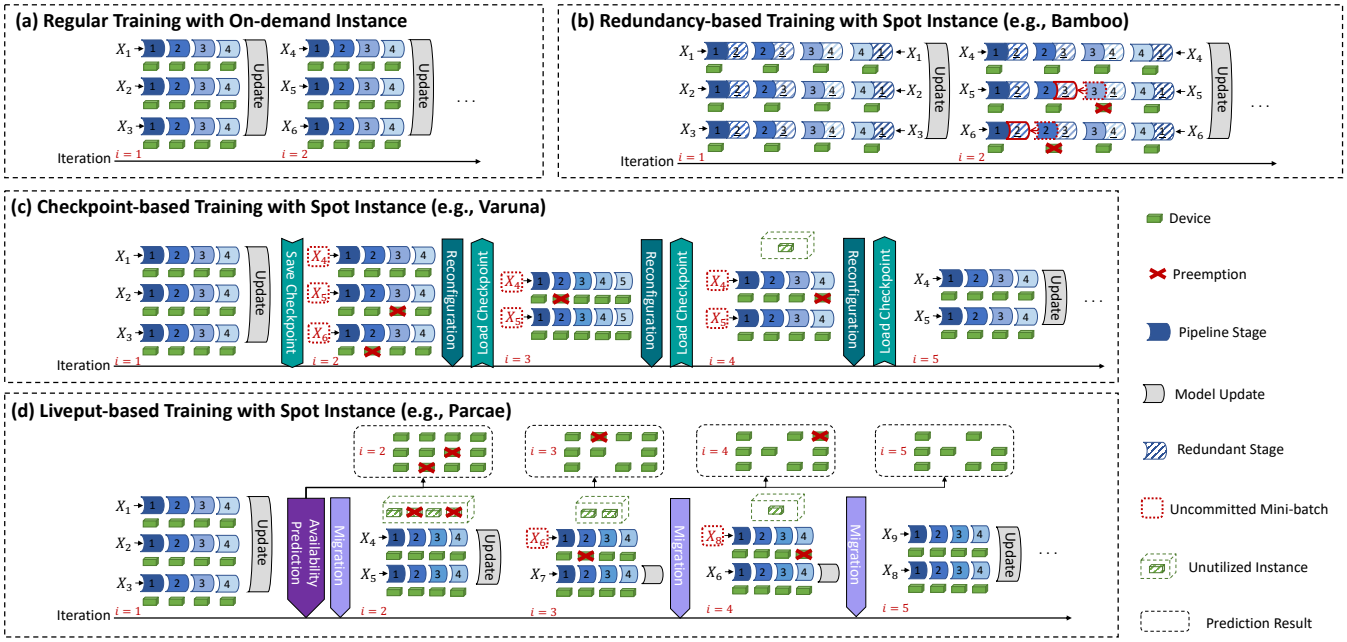



Figure 1: Illustration of pipelined data parallelism training over on-demand and spot instance respectively. Preempted spot instances are marked with red markers. $X_j$ represents the $j$-th mini-batch of input data.

point, resulting in wasted computation as model updates made since the last checkpoint are lost.

The second line of work uses redundant computation to provide resilience in the presence of preemptions. For example, as shown in Figure 1b, Bamboo [47] replicates DNN computations across spot instances by letting each instance in a pipeline perform normal computations over assigned DNN layers (dark boxes) and redundant computations over its successor's layers (striped boxes). Upon an instance preemption, its predecessor has all the information (e.g., layers and activations) to continue DNN training. Although Bamboo achieves higher training throughput than pure checkpointing-based methods when preemption rates are high, its computation efficiency can still be limited. This is because it is difficult to completely hide the overhead of redundant computation through pipeline bubbles, especially for large-scale models (§10.2). Additionally, storing redundant model states increases per-GPU memory consumption. Existing redundancy-based methods such as Bamboo address this challenge by increasing pipeline depth, but this can lead to reduced computation efficiency and increased vulnerability to preemptions.

To address the performance and scalability limitations of existing approaches, this paper presents Parcae, a *proactive*, *liveput-optimized* system for DNN training on spot instances. Parcae combines data and pipeline parallelism for DNN training on spot instances, and maintains identical semantics as on-demand training. A key insight behind Parcae is that different strategies to parallelize DNN training exhibit *diverse robustness* under preemptions. For example, a strategy with long pipelines achieves higher throughput but is more vulnerable to preemptions than a strategy with shorter pipelines.

Parcae is designed to maximize *preemption-aware* throughput in a proactive way. We purpose a formulation of *liveput* for DNN training on preemptible instances, which is the *expected* training throughput of a DNN job under different preemption scenarios. A key advantage of liveput is that it considers both the throughput of a parallel configuration *and its robustness* under preemptions. Figure 1d illustrates how Parcae optimizes liveput. After observing two preemptions ($i = 2$ in the figure), Parcae anticipates that the cloud has reached its capacity limit and expects additional preemptions in the near future. Therefore, instead of maintaining two pipelines each with five instances, which maximizes throughput, Parcae keeps four instances on each pipeline, which is more robust under additional preemptions and maximizes liveput. This allows Parcae to cheaply handle future preemptions using lightweight live migrations ($i = 3, 4$ in the figure).

There are three key challenges Parcae must address to optimize liveput: (1) predicting liveput, (2) handling preemptions, and (3) discovering parallel configurations to maximize liveput. We elaborate these challenges and the main ideas Parcae uses to overcome them.

First, spot instances can be preempted and reallocated due to many reasons (e.g., market price changes, resource constraints) at any time. It is challenging to know ahead of time when and which specific instances will be preempted/allocated by the cloud provider; nor does the cloud provider provides any hints or auxiliary information on how

instance preemption and addition decisions are made. However, estimating the liveput of a parallel configuration requires considering a variety of preemption scenarios.

Instead of predicting preemptions and allocations for individual instances, Parcae uses a two-level approach to forecasting the availability of instances at a coarse granularity. First, the *availability predictor* takes the instance preemption and allocation history as input and only predicts the *number of available instances* in the near future. Second, the *Monte Carlo preemption sampler* uses the predicted instance availability to sample preemptions. This two-level approach allows Parcae to employ a lightweight predictor to forecast spot-instance availability and quickly estimate the liveput of different parallel configurations.

Second, existing checkpoint- and redundancy-based approaches to handling preemptions introduce significant memory and computation overheads. Checkpoint-based systems (e.g., Varuna [8]) omit all model updates since the last checkpoint after each preemption, and periodically saving and loading checkpoints introduce additional overheads. These overheads are substantial even by adopting fine-grained checkpointing mechanisms [32] for better overlapping (see §10.2). Meanwhile, redundancy-based systems (e.g., Bamboo [47]) require redundant computation on each instance even in the absence of preemptions, which decreases training throughput and increases monetary cost due to redundant computations.

To effectively handle preemptions, Parcae uses a lightweight live migration mechanism that allows DNN training to proceed despite losing instances and without introducing redundant computation as done by prior work. To achieve this goal, Parcae's live migration mechanism always uses the same number of samples to update model's parameters in each training iteration and opportunistically reorder samples to avoid redundant computation or restarting training. This approach preserves model accuracy by leveraging the stochastic nature of DNN training — all training samples are drawn independently from an intrinsic data distribution and *reordering samples does not affect model accuracy* [10].

Third, optimizing liveput requires reasoning about instance preemptions and allocations and quickly adapting to new resources allocations while minimizing transition cost. Recent work (e.g., PipeDream [34] and Alpa [55]) has proposed a variety of techniques to automatically discover throughput-optimized parallel configurations for DNN training. However, all these approaches assume a fixed set of GPUs and do not apply to spot-instance training.

To address this challenge, Parcae's *liveput optimizer* formulates the problem of maximizing liveput as an optimization task and uses a novel dynamic programming algorithm to explore the search space of parallel configurations that combine data and pipeline parallelism and discover an *optimal* parallel configuration in the search space.

The above techniques allow Parcae to significantly outperform prior work. Figure 2 compares Parcae against Bamboo and Varuna for training GPT-2 on 32 spot V100 GPU instances on AWS using a collected preemption trace. Parcae outperforms Bamboo and Varuna by 2.4× under the same preemptions. The grey curve shows an *ideal* case, where Parcae knows *all* future preemptions and allocations and maximizes liveput accordingly. Parcae achieves *89%* efficiency of the ideal case. We have evaluated Parcae on a variety of DNN models and preemption traces and shown that Parcae outperforms Varuna by up to 9.9× and Bamboo by up to 10.8×. Moreover, our evaluation shows that Bamboo and Varuna cannot scale to large models — for certain spot-instance traces, both of them cannot make *any* progress for training GPT-3 [12] with 6.7 billion parameters, while Parcae can achieve almost identical performance as its ideal case (i.e., knowing all future preemptions and allocations).

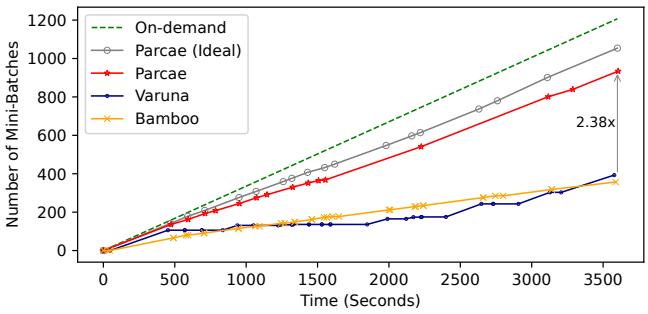



Figure 2: Comparing Parcae and prior work for training GPT-2 [38] on 32 spot GPU instances. Note that Parcae, Bamboo, and Varuna use an identical preemption trace.

This paper makes the following contributions:

- We propose liveput, a novel metric that simultaneously consider the performance and robustness of a parallelization strategy for DNN training on spot instances.
- We build Parcae, a liveput-optimized system for spot-instance training that accurately predicts instance availability, cheaply handles preemptions, and efficiently optimizes training performance under preemptions.
- We evaluate Parcae and show that it outperforms Varuna and Bamboo by up to 9.9× and 10.8×, and supports training large-scale models on spot instances.

## 2 Background

### 2.1 Distributed DNN Training

***Data parallelism.*** Data parallelism [7, 36] is the most widely used parallelization strategy in distributed DNN training. Each GPU has a model replica and performs forward and backward computations for different batches of data samples independently. It requires to synchronize model gradients (e.g., All-Reduce [4]) before mode update.

***Pipeline parallelism.*** Pipeline parallelism [19] partitions DNN model into different stages with data dependency. Each stage is trained on one GPU, and different GPUs communicate

activations and corresponding gradients, which are computed by forward and backward computation respectively, instead of parameter gradients. A mini-batch of training samples is split into multiple micro-batches in pipeline training and pipeline parallelism exploits the opportunity to parallelize the computations of different micro-batches.

***Hybrid data and pipeline parallelism.*** Some studies [15, 33] combine data and pipeline parallelism to further accelerate the training of large models. Given a number of GPUs, the training throughput varies for different parallel configurations, which describes the number of stages and data-parallel pipelines it owns. Some recent systems (e.g., FlexFlow [48], Alpa [55], Galvatron [30]) further involve more complicated model parallelism to benefit distributed training of particular DNNs. However, they can not be applied on spot instance with dynamic device membership. Our approach considers hybrid data and pipeline parallelism, follows Varuna and Bamboo, and leaves the exploration of more fine-grained model parallelism as our future work.

## 2.2 Spot-Instance Training

Recent frameworks [6, 8, 47] exploit cheap but preemptible instances provided by clouds to train DNN models on. TorchElastic [6] focuses on elastic data parallelism training and cannot be adopted to large models, where pipeline parallelism is definitely needed. Since the availability of spot instances varies significantly and frequently, it is critical to decide the parallel configuration for a DNN model in response to preemptions and allocations. Bamboo [47] keeps the pipeline depth fixed and varies the number of pipelines according to the availability of spot instances. This mechanism makes it difficult for Bamboo to utilize spot instances, which have low availability, for large models that require a long pipeline. Varuna [8] introduces job morphing to dynamically change the parallel configuration and maximize throughput for a given number of spot instances. For instances with low preemption rate or models with negligible reconfiguration cost, switching to the optimal parallel configuration is definitely optimal. However, the current spot instance market and DNN models violate the two conditions, making it sub-optimal to always adopt the parallel configuration with the optimal throughput.

# 3 Liveput

This section introduces *liveput*, a new metric for DNN training that describes the *expected* training throughput of a parallelization strategy on spot instances by simultaneously considering its throughput and robustness under preemptions.

## 3.1 Definition of Liveput

To address the challenge mentioned above, we introduce liveput, a novel metric for distributed DNN training on spot instances that considers both the performance of a DNN system as well as potential preemptions.

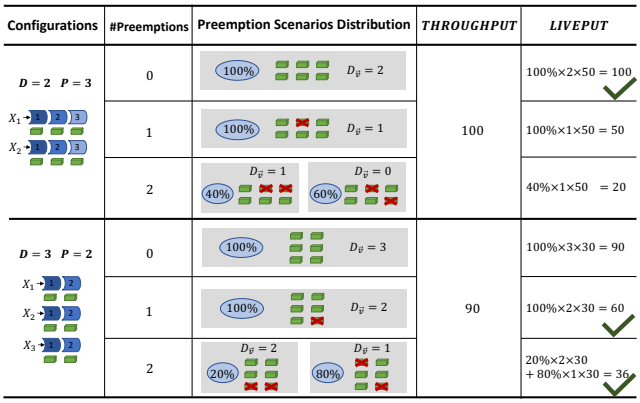



Figure 3: Comparing the liveput and throughput of different parallel configurations and preemption scenarios.

**Definition 1** (Liveput)**.** *Let $(D,P)$ denote the parallel configuration of a DNN training job, where $P$ is the number of pipeline stages, and $D$ is the number of data-parallel pipelines. The* ***liveput*** *of this training job is the expectation of its throughput under all possible preemption scenarios:*

$$\textsc{Liveput}(D,P,\mathcal{V}) = \mathop{\mathbb{E}}_{\vec{v}\sim\mathcal{V}}\left[\textsc{Throughput}(D_{\vec{v}},P_{\vec{v}})\right] \quad (1)$$

*where $\mathcal{V}:\{0,1\}^{D\times P}\rightarrow[0,1]$ is the probability distribution of all preemption scenarios. Each $\vec{v}$ is an preemption indicator vector, $v_k=1$ if instance $k$ will be preempted and $v_k=0$ otherwise.* $\textsc{Throughput}(D_{\vec{v}},P_{\vec{v}})$ *is the throughput of the new parallel configuration $(D_{\vec{v}},P_{\vec{v}})$ after preemption $\vec{v}$.*

Note that we follow prior work [8, 47] and focus on data- and pipeline-parallel DNN training in this paper, while the liveput definition can easily generalize to other parallel configurations such as model [21] and reduction [48] parallelism.

## 3.2 Comparing Liveput and Throughput

A key advantage of liveput is that it considers how the performance of a parallel configuration changes under different preemption scenarios. Figure 3 demonstrates this advantage with a DNN training example on six spot instances with two possible parallel configurations: $\{D=2,P=3\}$ and $\{D=3,P=2\}$. For simplicity, we assume the throughput of a pipeline with three (or two) stages is 50 (or 30) samples/second and ignore the parameter synchronization cost. We compare the two parallel configurations under three preemption scenarios: (a) no preemption, (b) one preemption, and (c) two preemptions. We also assume that the preemption probabilities of all instances are the same.

Figure 3 compares the throughput and liveput of the two parallel configurations under the three preemption scenarios. Throughput is independent of instance preemptions; therefore, $\{D=2,P=3\}$ achieves a higher throughput than $\{D=3,P=2\}$ for all cases. On the other hand, liveput considers the amount of possible preemptions as well as the distri-

bution of these preemptions over spot instances. When there is no preemption (i.e., fixed resource allocation), liveput is equivalent to throughput. Once a concrete future preemption scenario is given as a prior condition, the corresponding liveput can be treated as the effective throughput after such a preemption. For example, under preemption of 1 or 2 instances, the configuration $\{D=3,P=2\}$ achieves higher effective throughput than $\{D=2,P=3\}$. Intuitively, due to data dependencies between the pipeline stages, longer pipelines are more vulnerable to preemptions, since a single preemption would invalidate an entire pipeline within a mini-batch, and shorter pipelines exhibit better elasticity and resilience under frequent preemptions. Existing throughput-optimized approaches fail to consider this trade-off when estimating training efficiency and may make suboptimal decisions.

# 4 Parcae Overview

Figure 4 shows an overview of Parcae, a liveput-optimized system for DNN training on spot instances. Computing liveput requires predicting instance preemptions and allocations. Since predicting instance-wise availability is infeasible (§5.1), Parcae uses a two-level approach to forecasting the availability of all instances at a coarse granularity, where an *availability predictor* takes the instance preemption/allocation history as input and only predicts the *number of available instances* in the future, and the Monte Carlo *preemption sampler* uses the predicted availability to sample preemptions and allocations.

Parcae's *liveput optimizer* takes the predicted instance availability as input and discovers a parallel configuration to maximize the liveput of the DNN model. The liveput optimizer formulates the problem of maximizing liveput as an optimization task and uses a dynamic programming algorithm to discover an optimal parallel configuration.

To migrate across different parallel configurations and handle potential preemptions, Parcae uses three *live migration* strategies. These migration strategies leverage statistical robustness of DNN training, allow Parcae to significantly reduce migration and preemption overheads compared to existing checkpoint- and redundancy-based systems.

For the rest of this paper, we introduce Parcae's availability predictor in §5, live migration strategies in §6, and liveput optimizer in §7. §8 describes how Parcae handles exceptional cases where actual preemptions mismatch Parcae's predictions. Finally, we discuss Parcae's design and implementation on modern clouds in §9 and evaluate its performance in §10.

# 5 Availability Predictor

## 5.1 Instance-wise Availability Unpredictability

There are several factors that affect spot-instance preemptions and allocations, including the types of the instances a user requires and their availability zones, the price of the current spot instance market, and competitions from other users. Most existing approaches to predicting the availability of spot instances focus on estimating their prices [16, 17], which cannot

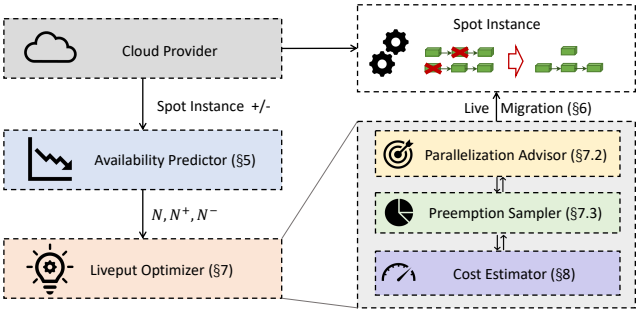



Figure 4: An overview of Parcae.

be used to estimate their lifetime. Prior work [16, 27, 31, 53] has also tried to predict the reliability of spot instances based on historical data collected from cloud providers. These attempts rely heavily on the cloud behaviour, which varies across cloud providers and availability zones within a cloud. Moreover, for a new cloud or zone, applying these data-driven approaches before running a job is expensive and time consuming. As a result, accurately forecasting individual instances' preemptions (i.e., $\vec{v}$ in Definition 1) is impractical since clouds currently do not support specifying preferences on the instance preemption order (i.e., which instances to preempt first), nor do they provide any auxiliary information that can help understand preemption and allocation decisions.

## 5.2 Statistical Availability Prediction

To make Parcae a general and practical DNN training system on spot instances, the only visible and reliable information is the past preemption/allocation records of the current user-submitted training job. Instead of forecasting when and which instance will be preempted in the future, Parcae uses a coarse-grained time-series forecasting approach. We observe that it is possible to predict the *total amount* of available spot instances for short time intervals in the future and benefit Parcae's proactive optimization performance.

***Problem formulation.*** We split the timeline of a training job into equally sized intervals, where the length of an interval $T$ is a hyper-parameter. For the $i$-th interval, we define a tupe $(N_i, N_i^+, N_i^-)$ to represent the number of available instances, newly allocated instances, and preempted instances within the $i$-th interval, respectively. We assume that node preemptions and allocations only happen at the beginning of each time interval and that all available spot instances are stable within a time interval; this assumption is reasonable since each interval is small (e.g., 1 minute). Therefore, we have $N_i = N_{i-1} + N_i^+ - N_i^-$ $(i > 0)$. Instead of predicting $N_i^+$ and $N_i^-$, Parcae's availability predictor only forecasts a sequence of $N_i$ (i.e., overall availability) and uses $N_i$ to derive $N_i^+$ and $N_i^-$. This design is based on an important observation that a cloud does not preempt existing instances and allocate new instances at the same time, therefore $N_i^+ = \max(0, N_i - N_{i-1})$ and $N_i^- =$

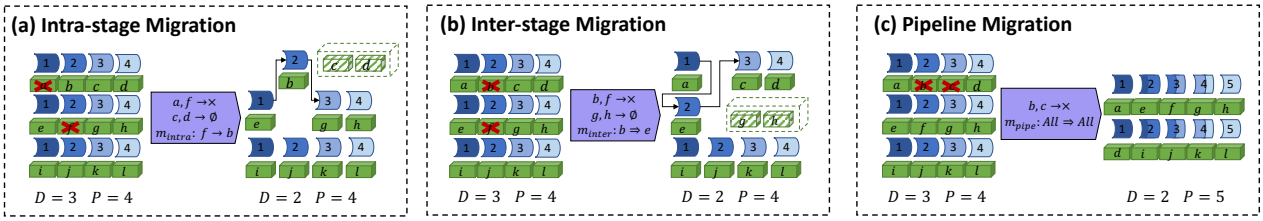
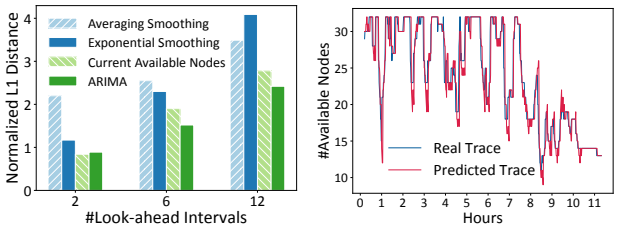



Figure 6: Illustrations of different migration strategies over a $3 \times 4$ parallel configuration facing 2 preempted instances.




(a) ARIMA vs. other models (b) ARIMA-predicted vs. real trace

Figure 5: (a) Comparison of normalized L1 distance of predictive performance for ARIMA and other solutions (H= 12, lower is better). (b) Comparison between ARIMA-predicted trace (H= 12, I= 4) and the ground truth.

$\max(0, N_{i-1} - N_i)$. Formally, in the time-series forecasting problem, an agent takes the instance availability trace in the past $H$ intervals as input and forecasts the instance availability for the future $I$ time intervals:

$$(N_i, \cdots, N_{i+I-1}) = \text{PREDICTION}(N_{i-H}, \cdots, N_{i-1}). \quad (2)$$

Note that $(N_i, \cdots, N_{i+I-1})$ can be used to derive the predicted instance preemptions and allocations for the next $I$ intervals.

Limited input data prevents Parcae from using complex prediction models such as deep neural networks. Instead, we propose to leverage lightweight statistical algorithms (e.g., moving averaging, exponential smoothing, current available nodes) and empirically study their performance in Figure 5a (more details are in Appendix B). We select the auto-regressive integrated moving average (ARIMA) algorithm [11] as our availability predictor due to its superior performance. We observe that ARIMA can faithfully describe the tendency of instance availability, as shown in Figure 5b. Finally, our evaluation on collected real-world preemption/allocation traces further verifies that the ARIMA predictor can help Parcae achieve near-optimal liveput (§10.2).

# 6 Live Migration

This section describes the *proactive* migration mechanism of Parcae. Existing checkpoint- and redundancy-based approaches handle preemptions *reactively*, leading to significant overheads. Instead, we design several fine-grained *live migration* strategies to *proactively* handle different future preemption scenarios. Given the preemption prediction results, Parcae could schedule efficient adjustments in advance to adapt to the dynamic instance availability.

## 6.1 Pipeline-aware Preemption Mapping

Before introducing live migration, we first discuss the *preemption mapping* step in Parcae. Recall that the outputs of the availability predictor (§5) only include statistical information (i.e., the number of preemptions or allocations during a time interval). However, the impact of an instance preemption highly depends on the instance's position in the data- and pipeline-parallel topology. Therefore, instance-wise preemption predictions (i.e., $\vec{v}$ in Definition 1) is still necessary to make efficient live migration decisions.

To bridge this gap, Parcae uses a probabilistic model to reason about the mapping from preemption events to actual instances. This preemption mapping is essential for data- and pipeline-parallel training because of the unique dependencies between instances. In particular, instances in the same pipeline have sequential dependencies for both forward and backward computation, and instances in the same stage have synchronization dependencies for parameter synchronization. For each preemption event, Parcae assumes that all spot instances may be preempted with the same probability (see the example in Figure 3). Note that such an assumption can be replaced by more accurate estimations when additional preemption information is provided by cloud providers.

## 6.2 Migration Strategies

Parcae uses three migration strategies (Figure 6) to handle preemptions: *intra-stage*, *inter-stage*, and *pipeline* migration.

***Intra-stage migration.*** In pipeline-parallel training, instances in the same stage maintain the same shard of model parameters. Therefore, when an instance is preempted, Parcae can opportunistically divert an available instance from the same stage in another broken pipeline. This intra-stage migration allows Parcae to re-establish a complete pipeline. As shown in Figure 6 (a), when instances $a$ and $f$ are preempted, Parcae can replace $f$ by moving $b$ to the second pipeline (e.g., $f \rightarrow b$), resulting in two complete pipelines. Intra-stage migration only requires updating the communication routing (e.g., $\rightarrow$) of a few instances and does not involve transferring parameters since $b$ and the preempted instance $f$ share the same model parameters and states.

***Inter-stage migration.*** When intra-stage migration does not help recover broken pipelines, Parcae opportunistically performs inter-stage migrations, which moves intstances across stages. Figure 6 (b) shows an inter-stage migration,

where instances $b$ and $f$ are preempted, and Parcae moves $e$ from the first stage to the second stage of the first pipeline (e.g., $b \Rightarrow e$), resulting in two complete pipelines. Inter-stage migration requires transferring model parameters (e.g., $\Rightarrow$) as the instances keep the model parameters and states of different stages. Both intra- and inter-stage migrations preserve pipeline depth and manage to recover as many data-parallel pipelines as possible.

***Pipeline migration.*** Changing the pipeline depth is an important choice for maximizing training efficiency. Compared with the other two migration strategies, pipeline migration requires repartitioning the DNN model into a different number of pipeline stages, which involves significant migration overheads as instances need to broadcast their model parameters (e.g., $All \Rightarrow All$). Pipeline migration is similar to the reconfiguration mechanism in prior work (e.g., Varuna [8], Bamboo [47]) to handle instance preemptions.

Parcae makes migration decisions by considering the current parallel configuration, the new optimized parallel configuration and the actual preemptions. Given the probabilistic mapping of predicted preemptions, Parcae automatically renewals the optimal parallel configuration and the migration strategy (§7.2). Once the prediction mismatches with the actual availability, Parcae adjusts the parallel configuration as well as the corresponding migration strategies for adaptation (§8). The actual migration decisions are finalized when preemptions really happen, and Parcae leverages the grace period (e.g., 30s on Azure [2]) to perform these migrations.

# 7 Liveput Optimizer

This section describes Parcae's *liveput optimizer*, which determines the parallel configurations of training a DNN model on spot instances to maximize its liveput.

## 7.1 Problem Definition

We formulate liveput maximization as an optimization problem, where the objective is to discover a sequence of parallel configurations to maximize the committed training samples in expectation of spot instance availability. The sequence length is set to be consistent with the number of time intervals predicted by the availability predictor (Section 5). Formally, the objective function $\Phi$ is the accumulated number of committed training samples during the $I$ time intervals:

$$\Phi(\mathbf{D},\mathbf{P} \mid \mathbf{N}) = \sum_{i=0}^{I-1} \phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1}), \quad (3)$$

where $N_i$ is the predicted number of available instances (see Section 5) at the $i$-th time interval. Recall that Parcae derives $N^-_{i+1}$ (i.e., the number of instances to be preempted) and $N^+_{i+1}$ (i.e., the number of instances to be launched) from $N_i$ and $N_{i+1}$. In addition, the preemption distribution $\vec{v}_{i+1}$ (Definition 1) is generated from $N_i$ and $N^-_{i+1}$ using the probabilistic preemption model developed in Section 6.1. Finally $\phi$ calculates the number of committed samples within a interval:

$$\begin{aligned} &\phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1}) \quad (4)\\ &\quad = \mathop{\mathbb{E}}_{\vec{v}_{i+1}} [\textsc{Liveput}(D_{i+1},P_{i+1} \mid \vec{v}_{i+1}) \times T_{\text{eff}}],\\ &T_{\text{eff}} = T - T_{\text{mig}}(D_i,P_i,D_{i+1},P_{i+1} \mid \vec{v}_{i+1}), \end{aligned}$$

where $T$ and $T_{\text{eff}}$ are the length of the time interval and effective training time after migrations, respectively, and $T_{\text{mig}}$ is the migration overhead. Note that $\phi$ extends liveput by making the preemption distribution $\vec{v}_{i+1}$ a prior. With these definitions, the objective of the liveput optimizer is:

$$\arg\max_{\mathbf{D},\mathbf{P}} \Phi(\mathbf{D},\mathbf{P} \mid \mathbf{N}) \quad (5)$$

where $\mathbf{N} = \{N_1,N_2,\cdots,N_I\}$ is the output of the availability predictor, and Parcae discovers a sequence of parallel configurations $(\mathbf{D},\mathbf{P})$ to maximize liveput.

## 7.2 Parallelization Advisor

Parcae uses a dynamic programming algorithm to explore the optimization space and discovers an *optimal* sequence of parallel configurations. Specifically, let $F(i+1,D_{i+1},P_{i+1})$ represent the maximal number of committed training samples at the end of the $i$-th time interval, which uses parallel configuration $(D_{i+1},P_{i+1})$. We start from $F(0,D_0,P_0) = 0$ and have the following optimal substrates:

$$\begin{aligned} &F(i+1,D_{i+1},P_{i+1}) \quad (6)\\ &= \max_{\forall D_i \times P_i \le N_i} \left\{ \begin{array}{c} F(i,D_i,P_i) + \\ \phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1}) \end{array} \right\}, \end{aligned}$$

and figure out the final target as $\max_{\forall D_I \times P_I \le N_I} \{F(I,D_I,P_I)\}$.

The DP algorithm considers all possible parallel configurations that satisfy resource constraints (i.e., $D_i \times P_i \le N_i$), and $\phi(D_i,P_i,N_i \mid D_{i+1},P_{i+1},N_{i+1})$ is the product of two terms in Equation (4). Here the exploration adapts a similar search space as Varuna with a size of $O(N \log N)$, which is large enough for most recent large DNNs consisting of a stack of homogeneous layers. It is also possible to extend to a larger search space (e.g., Alpa) for more complicated workloads. The first term $\textsc{Liveput}$ can be replaced by $\textsc{Throughput}(D_{i+1},P_{i+1})$, where $(D_{i+1},P_{i+1})$ is the new parallel configuration after live migration. Note that $(D_{i+1},P_{i+1})$ should be a feasible model partition that satisfies the device memory capacity. For unfeasible cases that violate memory constraints, their $\textsc{Throughput}$ is set to be zero.

The second term $T_{\text{eff}}$ depends on the preemption distribution, $(D_i,P_i)$, $(D_{i+1},P_{i+1})$, and the migration strategy to transit from $(D_i,P_i)$ to $(D_{i+1},P_{i+1})$. Given a pair of parallel configurations $(D_i,P_i)$ and $(D_{i+1},P_{i+1})$, there may exist multiple migration strategies with different overheads $T_{\text{mig}}$, and the cost of each migration strategy also depends on the DNN workload. Parcae uses a *cost estimator* (Section 9.4) to estimate $T_{\text{mig}}$ for different migration strategies. If the pipeline depth changes (i.e., $P_{i+1} \neq P_i$), Parcae infers that pipeline mi-

gration is performed. Otherwise, $T_{\text{mig}}$ should be attributed to either inter- or intra-stage migrations. When both of them are applicable, Parcae selects the one with lower migration cost. In the absence of preemptions (i.e., $N_{i+1} = N_i$), there can be no migration cost if $(D_{i+1}, P_{i+1})$ equals to $(D_i, P_i)$.

### 7.3 Preemption Mapping Sampler

As introduced in Section 6.1, preemption mapping is necessary to reason about live migration, since preemptions at different positions in the data- and pipeline-parallel topology require different migration strategies. Given $N_i$ spot instances, among which $N_{i+1}^-$ are to be preempted, the number of possible preemption mappings on a $D \times P$ topology grows exponentially in $N_{i+1}^-$. The large preemption mapping space makes it infeasible to explicitly consider all preemption scenarios or analyze the exact solutions mathematically.

To address this issue, Parcae uses sampling techniques to explore the mapping space and quickly discovers reasonable accurate approximations. Specifically, Parcae applies Monte Carlo (MC) sampling over the large space of all preemption scenarios and randomly samples $\vec{v}$ while preserving $N_{i+1}^- = \sum_{j=1}^{N_i} v_j$. For each sampled $\vec{v}$, Parcae identifies the corresponding migration costs. Parcae ensembles multiple trails of sampling to approximate the expectation in Equation (4). Note that this sampling step can be done offline in advance, therefore it does not block the dynamic programming optimization procedure. This allows parallelization advisor to quickly compute new parallel configurations and migration strategies during spot-instance training.

## 8 Exception Handling

This section describes how Parcae handles exceptional cases where actual spot-instance preemptions mismatch Parcae's predictions or the suggested parallel configuration is not compatible with the available spot instances.

***Parallelization adaptation.*** Compared to prior work, Parcae proactively adjusts parallel configurations by predicting instances' availability and planning live migrations ahead. However, if actual preemptions rarely differ from predictions, the liveput optimizer may not work on available spot instances. To address this issue, Parcae includes a *configuration adaptation* step to adjust the target parallel configuration before live migration. Specifically, when the number of actual available spot instances is greater (or less) than the predicted $N_i$, Parcae adds (or drops) data-parallel pipelines while preserving the pipeline depth. When available spot instances cannot even formulate a single pipeline, it will try to re-partition the pipeline into fewer stages. This adaptation ensures a feasible configuration without significant migration overheads, performing at least as well as existing throughput-optimized approaches that reactively handle preemptions when predictions go wrong.

***Fault tolerance.*** Even if the predictions align well with actual preemptions, there still exist rare cases where the migration strategies do not work. For example, if all instances in one stage are preempted, both inter- and intra-stage migration cannot recover this stage's status. Parcae uses a cheap, in-memory checkpointing mechanism (§9.3) to handle these cases. In addition, for the extreme cases where the number of available instances is less than the minimum feasible pipeline depth $P$, the training process has to be suspended until new spot instances are available.

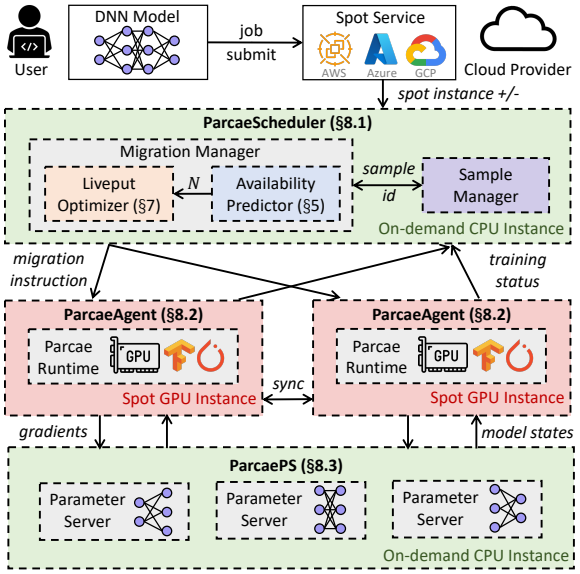


Figure 7: Overview of Parcae's design and implementation.

## 9 Parcae's Design and Implementation

Parcae consists of three main components as illustrated in Figure 7. First, ParcaeScheduler (§9.2) runs persistently on one on-demand CPU instance, determining the migration schedule based on our liveput optimizer and availability predictor. It also manages the training data samples to maintain the training semantics. Second, each spot GPU instance runs a ParcaeAgent (§9.2), which performs assigned training workload, monitors training progress, and executes the migration strategies issued by the ParcaeScheduler. Third, ParcaePS (§9.3) runs on several on-demand CPU instances to keep model checkpoints for rare rollback cases.

Parcae's implementation consists of $\sim 8K$ LoC in Python and takes PyTorch [25] as the default runtime. Communications between ParcaeScheduler and ParcaeAgent use `etcd` [5], a distributed key-value store. We implement live migration strategies by modifying DeepSpeed [40]. We show the workflow of ParcaeScheduler and ParcaeAgent in Algorithm 1 and introduce the detailed components as follows.

### 9.1 ParcaeScheduler

ParcaeScheduler has two major components: a *migration manager* and a *sample manager*. The former is responsible for parallelization and live migration, and the latter handles the data samples distribution.

***Migration manager.*** As shown in Algorithm 1, the migration manager keeps receiving instance availability information (i.e., preemption or allocation interruptions) from the cloud provider and updating the current number of available instances (line 3). As discussed in §8, the parallel configuration $(D_i, P_i)$ computed in the previous iteration using predicted availability may be incompatible with the current instances' availability. To handle this exception, ParcaeScheduler first adjusts the target parallel configuration (line 4) and then generates the required migration strategy $S_i$ based on the current and target configurations $(D_{i-1}, P_{i-1})$ and $(D_i, P_i)$. Note that the adaptation step (line 4) is performed before generating the migration strategy (line 5) so it will not involve re-adjustment overheads. Next, the availability predictor will forecast the number of available instances for a series of future intervals (i.e., $\mathbf{N}_{i+1}, \cdots, \mathbf{N}_{i+I}$) based on the historical information (line 7). Finally, the liveput optimizer makes parallelization suggestions for the following time interval using the prediction (line 8). The workflow continues until the training job is completed.

The handling of instance preemption and allocation interruptions are slightly different. Allocations are *controllable* as they only occur after we consciously send requests to the cloud, although they may not always succeed. We let a new instance join after its ParcaeAgent is successfully initialized. In contrast, preemptions are *passive* and may interrupt instances at any time, which requires additional mechanisms to handle various exceptions. Fortunately, the clouds usually provide a small grace period to inform the preemption before it happens. As the duration is usually enough to finish a mini-batch's training, we utilize the preemption notice to simplify the implementation and enforce instances to be preempted only at the mini-batches' boundaries. Parcae also handles rare failures that may interrupt training process, in which case ParcaeScheduler restarts training using the latest checkpoint in ParcaePS, avoiding losing model updates.

***Sample manager.*** The training dataset is divided into mini-batches of fixed size and trained by DNNs iteratively. Each mini-batch of samples are "committed" after each iteration. However, preemptions may terminate training at any time, resulting in uncommitted mini-batches (Figure 1). To guarantee the same training semantics as on-demand instances, the sample manager tracks each data sample, records all uncommitted samples' indices, and makes them rejoin the training process later. This guarantees that all data samples are trained exactly once per epoch, preserving identical theoretical convergence property as the original data feeding order. We also provide a convergence experiment in Figure 16 to verify its training correctness.

## 9.2 ParcaeAgent

A ParcaeAgent runs on each spot GPU instance to interact with ParcaeScheduler as shown in Algorithm 1. It repeatedly receives a migration instruction from the ParcaeScheduler (line 13). If no migration is required, the ParcaeAgent requests a batch of training samples and starts model training (line 15-16). Otherwise, it performs the assigned migration instruction (line 14). ParcaeAgent manages to reuse the current model states to alleviate checkpoint overheads and rollbacks. For example, intra-stage migration is implemented by rebuilding communication groups and reusing previous model states on each GPU. For inter-stage and pipeline migration, additional costs are required for loading the latest model states from other instances via GPUs' peer-to-peer communications. Specially, if all instances of a stage are preempted, all the ParcaeAgents have to roll back to a previous checkpoint. In this way, ParcaeScheduler automatically generates the most efficient migration strategy and let the ParcaeAgents transit to the target parallel configuration. Note that, the ParcaeScheduler also notifies a ParcaeAgent if it will be preempted or stay idle (i.e., $N_i - D_i \times P_i$ instances will be idle) by sending a halt or termination instruction to the ParcaeAgent.

**Algorithm 1** Workflow of Parcae components.

```
    ▷ ParcaeScheduler
 1: function MigrationManager(D_0, P_0)
 2:     for i in 1, 2, 3, ··· do
 3:         N_i ← Receive availability info from cloud provider
 4:         (D_i, P_i) ← AdjustParallelConfiguration(N_i)
 5:         S_i ← GetMigrationStrategy ((D_{i-1}, P_{i-1}), (D_i, P_i))
 6:         Send migration strategy S_i to all ParcaeAgents
 7:         N_{i+1}, ··· , N_{i+I} ← AvailPredictor(N_{i-H+1}, ..., N_i)
 8:         (D_{i+1}, P_{i+1}) ← LiveputOpt((D_i, P_i), N_i, ..., N_{i+I})
 9:         if job completes then
10:             break
    ▷ ParcaeAgent
11: function ParcaeRuntime(model, batch_size)
12:     while job does not complete do
13:         Receive migration instruction m from ParcaeScheduler
14:         Apply migration instruction m if m is not empty
15:         X, Y ← DataLoader(batch_size)
16:         Train(model, X, Y)
```

## 9.3 ParcaePS

Parcae needs checkpoints to handle rare cases as introduced in §8. Unlike prior checkpointing approaches relying on expensive cloud storage (e.g., S3 on AWS), Parcae employs several cheap on-demand CPU instances (e.g., `c5.4xlarge` instance, $0.68\$/hour$) to maintain the latest model states in their DRAM. Instead of directly communicating model states and weights as prior checkpointing approaches, the ParcaePS maintains an up-to-date checkpoint by iteratively synchronizing gradients with spot GPU instances to update the model states the on CPU side (e.g., parameters and optimizer states), which reduces communication by $5\times$ for stateful optimizers (e.g., Adam [23]) in the FP16 format [41]. Parcae also partitions gradients into small pieces for better overlapping and prevents bandwidth competition with cross-stage activation transfer.

Table 1: Overview of the four trace segments evaluated.

| Trace | $H_AD_P$ | $H_AS_P$ | $L_AD_P$ | $L_AS_P$ |
|---|---|---|---|---|
| *Availability* | High | High | Low | Low |
| *Preemption intensity* | Dense | Sparse | Dense | Sparse |
| #avg instances | 27.05 | 29.63 | 16.82 | 14.60 |
| #preemption events | 9 | 6 | 8 | 3 |
| #allocation events | 8 | 5 | 12 | 0 |
| length | 1h | 1h | 1h | 1h |

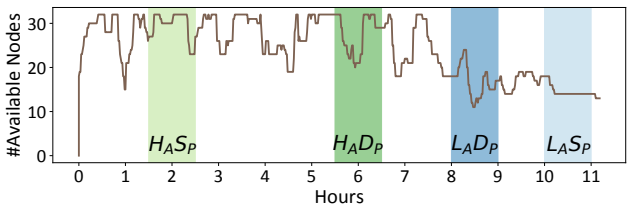


Figure 8: The complete trace and segments of four scenarios.

## 9.4 Cost Estimator

We develop a *cost estimator* to estimate migration cost by considering different preemption scenarios and parallel configurations. We conduct an empirical study to profile the migration cost and find that it varies across several factors (details in Appendix A). Some of these terms have relatively fixed overheads like CUDA context initialization (less than 10s). The communication group updating and model building costs are associated with the parallel configuration (less than 30s). The model transfer cost varies considerably according to the preemptions (up to 60s). We consider the instance network topology for each preemption scenario and adopt an $\alpha - \beta$ model [49] to accurately estimate the communication cost.

# 10 Evaluation

## 10.1 Experimental Setup

***DNNs.*** We select five popular DNNs for various applications. ResNet-152 [18] and VGG-19 [45] are CV tasks, and we use CIFAR-100 [24] as the training dataset. BERT [14], GPT-2 [38], and GPT-3 [12] are popular model architectures for NLP tasks, and we evaluate them on WikiText-2 [28]. We use GPT-2 and GPT-3 including 1.5 and 6.7 billion model parameters respectively. More setting details are in Appendix C.

***Traces.*** Due to the dynamic availability of spot instance, it is almost impossible to evaluate different systems on real spot instances multiple times and expect consistent dynamic environments. Instead, to make a fair comparison, we take the real spot instance availability traces and replay them on regular instances. Specifically, we collect a 12-hour trace on a 32-instance cluster with `p3.2xlarge` instances on AWS. Inspired by Bamboo [47], we extract representative segments from the whole trace for our evaluation. We design two new measurements for each segment, including the *availability* (i.e., the average number of instances) and the *preemption intensity* (i.e., the number of instance preemption and allocation events). Table 1 and Figure 8 show four extracted 1-hour trace segments based on different availability and preemption intensity. Traces with over 70% available instances are high availability (i.e., $H_A$) traces, otherwise have low availability (i.e., $L_A$). Dense preemption intensity traces (i.e., $D_P$) have around 20 instance preemption and allocation events, but sparse preemption intensity traces (i.e., $S_P$) only have few. We replay these four trace segments on 32 on-demand V100-16GB GPU instances to simulate spot-instance clusters.

Table 2: Comparison of monetary cost ($\times 1e^{-6}$USD) for different models and approaches. We report per-image cost for ResNet and VGG and per-token cost for BERT and GPT.

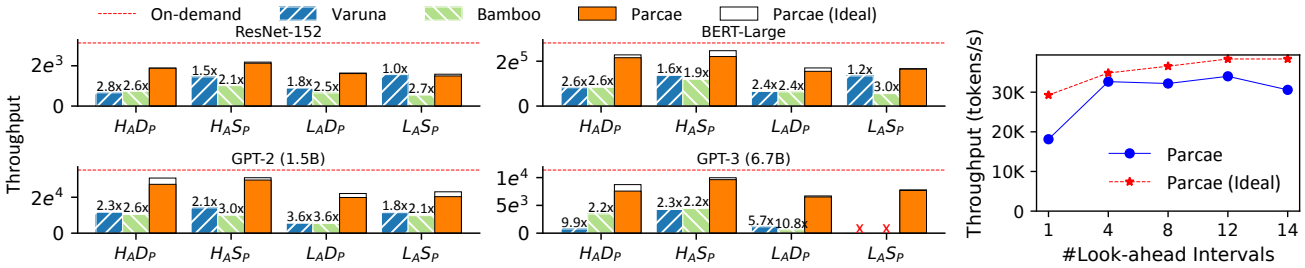
| Model | Trace | On-Demand | Varuna | Bamboo | Parcae |
|---|---|---|---|---|---|
| ResNet | $H_AD_P$ | 8.68 (2.3×) | 10.86 (2.8×) | 9.77 ( 2.6×) | 3.81 (1×) |
| | $H_AS_P$ | 8.68 (2.4×) | 5.32 (1.5×) | 7.61 ( 2.1×) | 3.62 (1×) |
| | $L_AD_P$ | 8.68 (3.2×) | 4.89 (1.8×) | 6.72 ( 2.5×) | 2.71 (1×) |
| | $L_AS_P$ | 8.68 (3.4×) | 2.43 (1.0×) | 6.96 ( 2.7×) | 2.54 (1×) |
| VGG | $H_AD_P$ | 12.43 (2.7×) | 12.10 (2.6×) | 12.11 ( 2.6×) | 4.62 (1×) |
| | $H_AS_P$ | 12.43 (2.7×) | 6.52 (1.4×) | 13.12 ( 2.8×) | 4.66 (1×) |
| | $L_AD_P$ | 12.43 (3.4×) | 5.43 (1.5×) | 9.40 ( 2.6×) | 3.66 (1×) |
| | $L_AS_P$ | 12.43 (4.0×) | 3.37 (1.1×) | 8.88 ( 2.9×) | 3.11 (1×) |
| BERT | $H_AD_P$ | 0.10 (2.9×) | 0.09 (2.6×) | 0.09 ( 2.6×) | 0.03 (1×) |
| | $H_AS_P$ | 0.10 (2.8×) | 0.06 (1.6×) | 0.06 ( 1.9×) | 0.03 (1×) |
| | $L_AD_P$ | 0.10 (3.4×) | 0.07 (2.4×) | 0.07 ( 2.4×) | 0.03 (1×) |
| | $L_AS_P$ | 0.10 (4.2×) | 0.03 (1.2×) | 0.07 ( 3.0×) | 0.02 (1×) |
| GPT-2 | $H_AD_P$ | 0.62 (2.9×) | 0.49 (2.3×) | 0.55 ( 2.6×) | 0.21 (1×) |
| | $H_AS_P$ | 0.62 (3.0×) | 0.44 (2.1×) | 0.62 ( 3.0×) | 0.21 (1×) |
| | $L_AD_P$ | 0.62 (3.5×) | 0.63 (3.6×) | 0.64 ( 3.6×) | 0.18 (1×) |
| | $L_AS_P$ | 0.62 (4.1×) | 0.27 (1.8×) | 0.31 ( 2.1×) | 0.15 (1×) |
| GPT-3 | $H_AD_P$ | 2.39 (2.5×) | 9.35 (9.9×) | 2.07 ( 2.2×) | 0.94 (1×) |
| | $H_AS_P$ | 2.39 (3.0×) | 1.81 (2.3×) | 1.74 ( 2.2×) | 0.80 (1×) |
| | $L_AD_P$ | 2.39 (3.6×) | 3.81 (5.7×) | 7.28 (10.8×) | 0.67 (1×) |
| | $L_AS_P$ | 2.39 (4.8×) | - | - | 0.49 (1×) |

## 10.2 End-to-End Evaluation

We first compare the end-to-end training performance between Parcae and existing SOTA spot-instance training systems including Bamboo [47] (redundancy-based) and Varuna [8] (checkpoint-based). We also compare with on-demand instances training approach. The results are displayed in Figure 9a and Table 2. In all experiments, Parcae looks ahead 12 intervals based on the availability predictor, while Parcae (Ideal) looks ahead 12 intervals based on truth traces.

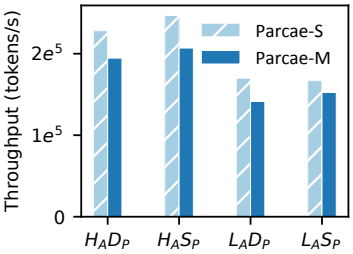
Parcae significantly outperforms both Bamboo and Varuna in terms of throughput for almost all the models and preemption traces. On average, Parcae delivers an overall of 2.59× higher throughput than Varuna and 3.0× than Bamboo. Apparently, Parcae is much more economical than Varuna and Bamboo as it completes more samples with the same monetary costs. Compared with on-demand instances, Parcae is 3.24× cheaper, and Parcae (Ideal) even achieves competitive throughput, e.g., only 14.2% lower for GPT-2 on high availability traces. The results also show that the performance of Parcae is quite close (i.e., up to 13.3%) to Parcae (ideal).

The performance improvement mainly comes from two aspects. First, Parcae's liveput optimized configurations balance the trade-off between throughput and available duration, in-




(a) Parcae throughput. (b) The effect of $I$ in predictor.

Figure 9: (a) Training throughput comparison among existing frameworks and Parcae on four traces. The dotted on-demand line shows the best throughput with on-demand instances. The numbers over the bars represent the speedup of Parcae over Varuna and Bamboo respectively. (b) The GPT-2 training throughput for the $H_AD_P$ trace with different look-ahead intervals.




| Trace | Monetary Cost Parcae-S | Parcae-M |
|---|---|---|
| $H_AD_P$ | **3.13** | 3.78 |
| $H_AS_P$ | **3.12** | 3.71 |
| $L_AD_P$ | **2.57** | 3.28 |
| $L_AS_P$ | **2.27** | 3.00 |

Figure 10: The comparison of throughput and monetary cost ($\times 1e^{-8}$USD/token) of BERT for Parcae on single-GPU instances (Parcae-S) and multi-GPU instances (Parcae-M).

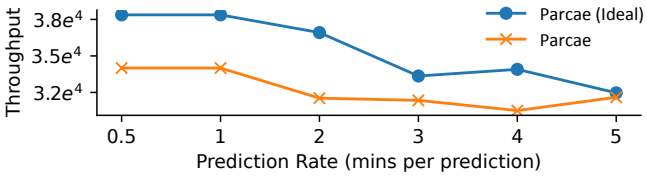



Figure 11: GPT-2 training throughput (tokens/s) using the $H_AD_P$ trace with different prediction rates.

stead of greedily doing expensive reconfiguration like Varuna. While Bamboo maintains a fixed long pipeline depth (e.g., 16 for GPT-2), leading to many unutilized instances, especially for low availability traces. Second, the migration mechanism in Parcae is highly efficient to handle preemptions. Varuna is designed for low preemption environments and relies on shared storage (e.g. S3) to save and load checkpoints. Although Varuna overlaps checkpoint saving with training iterations, when preemptions happen, it requires rolling back to the last checkpoint and loses tens of seconds' (i.e., the duration of one complete checkpointing) training progress for large models. To recover from preemptions, Varuna needs to load the last checkpoint from persistent storage and restart training, which is also expensive. Bamboo is designed for high preemption environments based on redundant computation. It can efficiently handle preemptions, but the redundant computation is inefficient and brings additional synchronization overheads between redundant and normal modules.

***Multi-GPU instances.*** To demonstrate the generality of Parcae, we also evaluate Parcae on multi-GPU instances. Unfortunately, we fail to collect meaningful multi-GPU spot instance traces on the cloud (e.g. `p3.8xlarge` with 4 V100 GPUs) as they show extremely low availability recently. Instead, we propose to generate the 4-GPU instance based on the single GPU trace by accumulating every four preemption or allocation events. Each 4-GPU instance is allocated at the first allocation event and preempted at the last preemption event. In this way, multi-GPU instance trace will have higher GPU hours than the single GPU trace in total. For multi-GPU instances, we follow prior work [39, 43] using pipeline parallelism only for inter-nodes. Figure 10 shows the training throughput and cost for different trace segments. Although our trace generation favors multi-GPU instances in theoretical availability, Parcae on single GPU instance still performs better in terms of both throughput and monetary cost. The major reason is that preempting one 4-GPU instance will interrupt 4 pipelines, significantly slowing down training. Besides, unutilized 4-GPU instances are also significant as it takes four times more GPUs to increase a new pipeline.

## 10.3 Breakdown Analysis

***Look-ahead interval length.*** Figure 9b shows the results of training GPT-2 with different numbers of look-ahead intervals on the $H_AD_P$ trace. Here Parcae looks back past 12 intervals and predicts the next $1, 4, 8, 12$, and 14 intervals respectively. The results show that Parcae (Ideal) keeps improving by considering longer futures and achieves the best performance when looking ahead 12 intervals. It shows the benefits of liveput-optimized configurations by considering future preemptions and allocations. On the other hand, Parcae exhibits a slightly different pattern, where its performance improves significantly by looking ahead 4 intervals compared with 1 interval ($1.8\times$). As Parcae looks ahead more intervals, the prediction error increases as we evaluated in Figure 5a. Figure 9b shows that Parcae can still yield significant improvement compared with looking ahead 1 interval, and achieves best performance by looking ahead 12 intervals. Overall, Parcae's throughput is 12.8% lower than that of the ideal case.

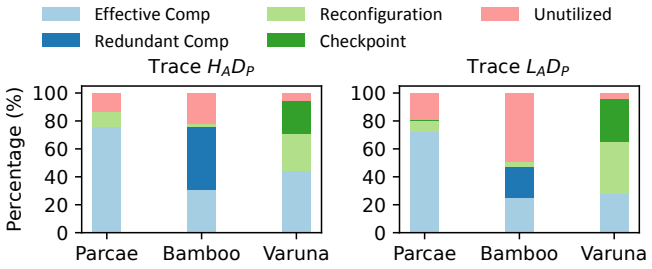



Figure 12: GPU hours breakdown of GPT-2 execution.

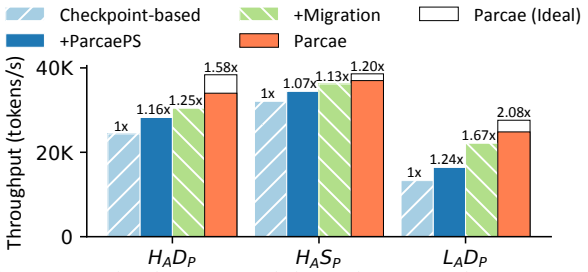



Figure 13: The decomposed throughput speedup on GPT-2.

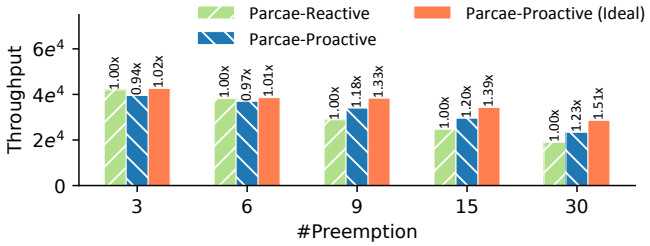



Figure 14: The throughput comparison between Parcae and Parcae-Reactive under different preemption intensity.

The result demonstrates that looking ahead longer can indeed help Parcae make more optimized decisions, and that there is still room to improve our availability predictor.

***Prediction rate.*** Figure 11 shows the results of training GPT-2 with different prediction rates on the $H_AD_P$ trace. As the prediction rate decreases, so do the training throughput achieved by Parcae and Parcae (Ideal). Fortunately, the execution time of the liveput optimizer is much less than one minute, which allows Parcae to use a high prediction rate and optimize frequently (i.e., per minute) for better performance.

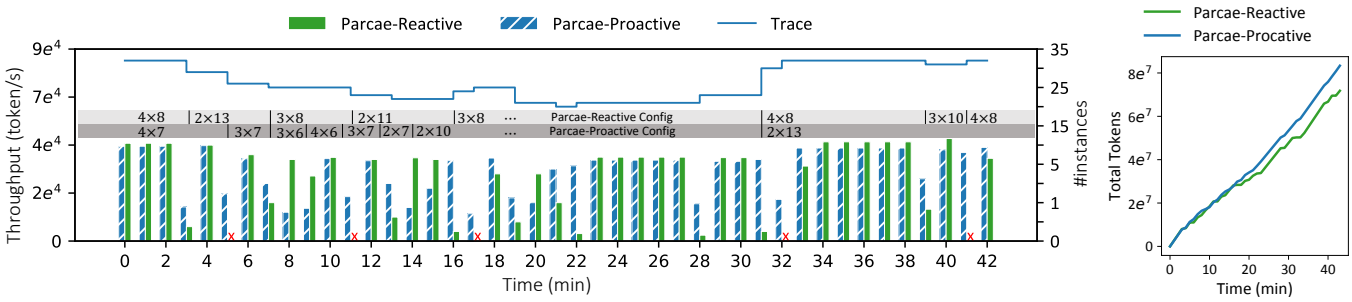
***GPU hours breakdown.*** To further understand the performance and drawbacks of different approaches, we breakdown the GPU hours of GPT-2 training into five components (Figure 12). The results demonstrate that Parcae spends the majority of GPU hours performing effective computation (i.e., committed mini-batches). In contrast, Bamboo spends more than 40% GPU hours on redundant computation on $H_AD_P$, while wastes more than 50% GPU hours on $L_AD_P$. Similarly, Varuna takes a long time to handle preemptions, including checkpointing and reconfiguration. As a result, their unutilized parts are quite small compared with Parcae. The results also align with the disadvantages we mentioned in §10.2.

***Parcae components analysis.*** Figure 13 shows how each component contributes to the performance improvements, using GPT-2 as an example. We start from a checkpoint-based approach with throughput-optimized execution plans. By adding ParcaePS and migration strategies, we improve the throughput by 13%-67%. Especially for trace $L_AD_P$ with low availability, it leaves little room for parallel configuration variation. When there are frequent preemption and allocation events, the migration allows training to make more progress than frequently triggering the costly reconfiguration. Finally, adopting liveput optimized parallel configurations improves an additional 25.5% over migration mechanisms.

## 10.4 Proactive v.s. Reactive

***Preemption Tolerance.*** We evaluate the performance of Parcae (i.e., Parcae-Proactive) and Parcae-Reactive with GPT-2 on a synthetic preemption trace Figure 14. The auxiliary baseline (i.e., Parcae-Reactive) is created by disabling the liveput optimization in Parcae and only enabling the parallelization adaptation mechanism (§8). Parcae-Reactive can be classified as a throughput-optimized system and used to highlight the advantages of our proactive, liveput-based approach. We generate the synthetic trace from the $H_AS_P$ trace by scaling the number of preemption events from 3 to 30 within one hour. The performance gap between Parcae-Reactive and Parcae-Proactive becomes larger as the preemption intensity increases, showing that our proactive approach can be more effective for scenarios with more frequent preemptions.

***Case study.*** As a case study, we compare the liveput-optimized Parcae with throughput-optimized Parcae-Reactive in detail using GPT-2 and partial $H_AD_P$ trace. Figure 15a shows each interval's instance availability, parallel configuration ($D \times P$), and average throughput as time elapses. We observe that for intervals with stable availability, Parcae-Reactive can select configurations with relatively higher throughput. However, greedily selecting execution plans that maximize throughputs suffers when preemptions or allocations happen because it neglects high reconfiguration costs. It can barely make training progress when the available instances frequently change. In contrast, Parcae carefully chooses parallel configurations by considering the future instance availability and adapting efficient migration strategies accordingly to ensure high training efficiency while mitigating expensive reconfiguration. For example, in the first 8 intervals, Parcae selects a pipeline depth of 7 and avoids changing pipeline depth as Parcae-Reactive does (e.g., 8 and 13). Although resulting in some unused instances, the progress made is still larger than running with Parcae-Reactive because of its reconfiguration overheads. Similar observation exists in the last 10 consecutive intervals, where Parcae maintains the same parallel configurations but leverages lightweight inter- and intra-stage migrations to adapt to dynamic preemptions




(a) Parallel configuration ($D \times P$) and average throughput inside each interval (i.e., 1 minute). (b) Accumulated tokens.

Figure 15: The comparison between Parcae-Reactive and Parcae-Proactive approaches for GPT-2 on $H_A D_P$ trace.

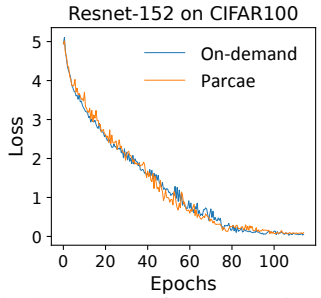



Figure 16: The loss curve of ResNet-152 on CIFAR100.

and allocations. As a result, Parcae achieves 16% more accumulated tokens within 40 minutes (Figure 15b).

## 10.5 Convergence Preservation

Figure 16 visualizes the convergence curve of training loss between Parcae running on spot instances and the baseline on on-demand instances. We observe that both convergence rates are very close, and Parcae reaches the same training loss of 0.058 as the baseline after training for 110 epochs. This verifies that the Parcae design and implementation align with the model convergence.

# 11 Related Work

***Dynamic preemptible instances.*** There is a trend of using preemptible instances on modern clouds for cheap service. Tributary [16] studies the latency issues from preemptions and proposes to switching preemptible offerings from clouds with different preemption likelihoods. BurScale [9] employs autoscaling to handle transient queuing in web service traffic. SciSpot [22] presents a reliability model for temporally constrained preemptions to optimize the job scheduling for scientific computing. HotSpot [42] transparently migrates spot VMs in lower price and achieves higher cost-efficiency. Snape [52] improves spot resources' availability by dynamically mixing on-demand VMs with spot eviction predictions. SpotServe [29] realizes fast and reliable serving of LLMs on cheap preemptible instances with dynamic reparallelization and optimal context migration. Prior work has demonstrated the cost benefits of spot instances in cloud computing and motivates the following related research.

***Preemptible distributed DNN training.*** Recently, using preempible instances for machine learning tasks is becoming popular as they are much more cost effective, like what is done in Varuna [8] and Bamboo [47]. SageMaker in AWS [3] automatically pauses the training job when a spot instance is interrupted and resumes from the checkpoint in S3 if the spot instance becomes available again. CM-DARE [26] analyzes distributed training under transient cloud GPU servers and provides a performance modeling methodology. SpotTune [27] leverages spot instances to paralleize hyper-parameter tuning for ML models. SkyPilot [54] migrates training workload to spot resources from other clouds and relaunch jobs using the periodical checkpoint from cloud storage. These approaches make meaningful explorations in this direction but are still suffering from the limited performance due to preemptions.

Oobleck [20] and Gemini [50] are concurrent works for quick failure recovery in distributed DNN training. Oobleck introduces pipeline reinstantiation with precomputed pipeline templates. Gemini uses in-memory checkpoints and orchestras checkpoint traffic schedule. Both are reactive approaches. Besides, Gemini targets dedicated instances and relies on high network bandwidth to reduce checkpointing time, while the bandwidth is low for spot instances.

# 12 Conclusion

In this work, we present an efficient distributed training system over spot instances, Parcae. The key idea is to proactively adjust the parallelization strategy using a novel metric, liveput, considering both training throughput and instance availability. With holistic system mechanisms and implementation optimizations, Parcae significantly outperforms checkpoint- and redundancy-based solutions in evaluations.


# Acknowledgement

We thank the anonymous reviewers and our shepherd, Le Xu, for their comments and helpful feedback. This material is based upon work supported by NSF awards CNS-2147909, CNS-2211882, and CNS-2239351, and awards from Amazon, Cisco, Google, Meta, Oracle, Qualcomm, and Samsung.

## References


[1] Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot/.

[2] Use azure spot virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms.

[3] Amazon sagemaker spot training. https://docs.aws.amazon.com/sagemaker/latest/dg/model-managed-spot-training.html, 2018.

[4] Nvidia nccl. https://developer.nvidia.com/nccl, 2021.

[5] Operating etcd clusters for kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/, 2021.

[6] Pytorch elastic. https://github.com/pytorch/elastic, 2021.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

[8] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[9] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 126–138, 2019.

[10] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[11] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[15] 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. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 431–445, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. Tributary: spot-dancing for elastic services with latency slos. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 1–14. USENIX Association, 2018.

[17] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 589–604, 2017.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[20] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 382–395. ACM, 2023.

[21] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[22] Jcs Kadupitiya, Vikram Jadhao, and Prateek Sharma. Scispot: Scientific computing on temporally constrained cloud preemptible vms. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[24] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[25] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12).

[26] Shijian Li, Robert J Walls, and Tian Guo. Characterizing and modeling distributed training with transient cloud gpu servers. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 943–953. IEEE, 2020.

[27] Yan Li, Bo An, Junming Ma, Donggang Cao, Yasha Wang, and Hong Mei. Spottune: Leveraging transient resources for cost-efficient hyper-parameter tuning in the public cloud. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 45–55. IEEE, 2020.

[28] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[29] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. *Proceedings of ASPLOS Conference*, 2024.

[30] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3):470–479, 2023.

[31] Ashish Kumar Mishra, Brajesh Kumar Umrao, and Dharmendra K Yadav. A survey on optimal utilization of preemptible vm instances in cloud computing. *The Journal of Supercomputing*, 74(11):5980–6032, 2018.

[32] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7937–7947. PMLR, 2021.

[35] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. Ras: Continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 505–520, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[37] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *CoRR*, abs/2104.10350, 2021.

[38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[41] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[42] Supreeth Shastri and David E. Irwin. Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 493–505. ACM, 2017.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[44] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[45] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[46] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[47] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. *CoRR*, abs/2204.12013, 2022.

[48] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022.

[49] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[50] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: fast failure recovery in distributed training with in-memory checkpoints. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 364–381. ACM, 2023.

[51] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and

Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[52] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, et al. Snape: Reliable and low-cost computing with mixture of spot and on-demand vms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 631–643, 2023.

[53] Sheng Yang, Samir Khuller, Sunav Choudhary, Subrata Mitra, and Kanak Mahadik. Scheduling ml training on unreliable spot instances. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pages 1–8, 2021.

[54] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.

[55] 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. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 559–578. USENIX Association, 2022.

Table 3: Overview of the five DNNs evaluated.

| Model | mini-batch | micro-batch | Dataset |
|---|---|---|---|
| ResNet-152 [18] | 2048 | 32 | CIFAR-100 [24] |
| VGG-19 [45] | 2048 | 32 | CIFAR-100 [24] |
| BERT-Large [14] | 1024 | 8 | WikiText-2 [28] |
| GPT-2 (1.5B) [38] | 128 | 1 | WikiText-2 [28] |
| GPT-3 (6.7B) [12] | 64 | 1 | WikiText-2 [28] |

Table 4: Migration costs in our experiments on AWS.

| Cost Terms | Magnitude ($s$) | Interfering Factor |
|---|---|---|
| Start process | $< 1$ | Instance state |
| Rendezvous | $0 \sim 10$ | |
| Init CUDA context | $0 \sim 10$ | |
| Load data | $0 \sim 10$ | Dataset |
| Build model | $0 \sim 10$ | Model, Configuration |
| Update comm. groups | $0 \sim 20$ | |
| Model states transfer | $0 \sim 60$ | Model, Configuration Preemption Scenario |

# A Addition Details of Migration Costs

Table 4 lists detailed costs of migrations and their magnitudes. All of them are profiled multiple times and averaged over five DNN models (see Table 3).

# B Additional Details of ARIMA

The ARIMA time-series forecasting algorithm is sensitive to trivial perturbations in inputs, which may impede its understanding of essential patterns from previous instance history. We introduce a few optimizations to ensure its predictions are faithful. First, we flatten random spikes that last for only 1-2 intervals in previous instance history, since such trivial noise will likely cause abrupt rise and falls in prediction. ARIMA also likes to simulate the tendency of the entire input curves. When input curves have multiple "hops", we ensure that ARIMA only learns from the most recent variations that are indeed beneficial for prediction. Second, though ARIMA can accurately capture intermediate fluctuations, its prediction can be so steep that it easily hits the upper and lower boundaries of available instances on intervals of sudden increase and decrease. To do so, we set upper and lower boundaries to limit the predicted curves based on observations of all spot instance traces we have. Additionally, our empirical study on traces indicate most intervals have a limitation on the extent of growth. Thus, we would also apply such constraints on predictions. We also apply additional penalty to flatten excessively steep predictions such as their predictions follow the essential patterns of AWS traces. We take care to reset ARIMA mispredictions when the generation deviates seriously from the input. With these rules and modifications, we ensure the ARIMA model can sufficiently describe future scenarios by learning from the past history.

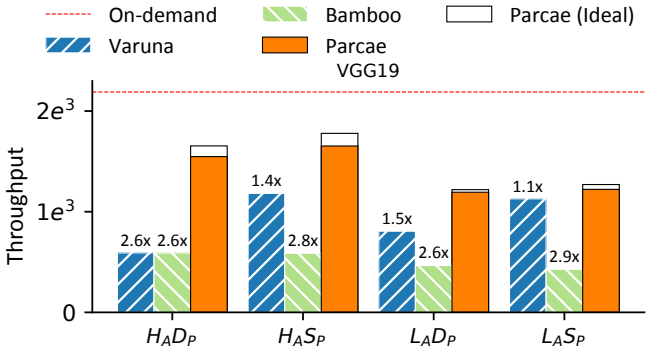



Figure 17: Training throughput comparison of VGG19 among existing frameworks and Parcae on four traces. The dotted on-demand line shows the best throughput with on-demand instances. The numbers over the bars represent the speedup of Parcae over Varuna and Bamboo respectively.

# C Additional Experimental Details

## C.1 End-to-End Evaluation Setting

We select five popular DNNs for various applications and summarize them in Table 3. For all the models, we used Adam optimizer with half precision (i.e., FP16) for training.

***Parallel Configuration.*** Parcae and Varuna will adjust parallel configurations according to instance availability during training. The parallel configuration of Parcae is decided by migration manager, while it is decided by job morphing for Varuna. We follow the settings of Vauna and first run a one-time profiling to collect primitive parameters of the hardward and the DNN model. Varuna will automatically decide the optimal parallel configuration considering DNN models and number of availability instances. Table 5 summarizes the parallel configurations used for Bamboo in our evaluation. Bamboo maintains a fixed pipeline depth and its redundant computation consumes a huge amount of memory. For different models, we tuned the number of pipeline stages and partitions to find an optimal parallel configuration for Bamboo. We find it requires at least 20 stages for Bamboo to run GPT-3 even with activation checkpointing [13] enabled, and Bamboo performs best for $P = 23$.

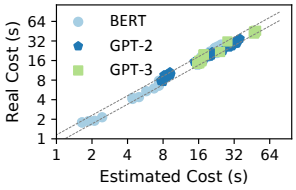
***VGG Results*** Figure 17 shows the end-to-end evaluation results of VGG19. Parcae significantly outperforms Varuna and Bamboo, except for trace $L_AS_P$, where Varuna achieves comparable performance with Parcae. We move these results in the appendix due to the limited page space.

## C.2 Parcae Components Evaluation

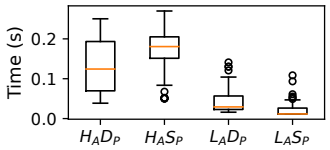
***Cost Estimation Accuracy.*** The cost estimator estimates migration cost for different preemption scenarios and parallel configurations. An accurate estimator is important for accurate liveput optimization. We compare the estimated migration cost predicted by cost estimator with the real migration time measured by actual executions. Figure 18a shows the results

Table 5: The parallel configuration of Bamboo in evaluation.

| Model | D | P |
|---|---|---|
| ResNet-152 [18] | 8 | 4 |
| VGG-19 [45] | 8 | 4 |
| BERT-Large [14] | 4 | 8 |
| GPT-2 (1.5B) [38] | 2 | 16 |
| GPT-3 (6.7B) [12] | 1 | 23 |




(a) The accuracy of cost estimator.




(b) The cost of migration advisor.

Figure 18: (a) Comparison between the estimated and actual reconfiguration time for different models. (b) Optimization time of looking ahead 12 intervals for GPT-2.

for different DNN models. The dashed lines indicate a relative difference of $-15\%$ and $15\%$ between real and estimated migration cost, respectively. The results demonstrate that our cost estimator is appropriate to evaluate the migration cost for different preemption scenarios and models.

***Optimization Cost.*** ParcaeScheduler periodically runs online liveput optimization to suggest the parallel configuration for the next interval. We evaluate the optimization time it takes to look ahead 12 intervals for one run on one CPU machine. Figure 18b shows the results of GPT-2 on different trace segments. Overall, one optimization takes less than 0.3 seconds, which is negligible compared with interval length. Therefore, the liveput optimization will not delay the training process.