https://crad.ict.ac.cn
62(12): 1−17, 2025

AdaptDNN: 一个自适应可伸缩的大模型分布式训练系统

刘国栋[1,2] 朱家祺[1,2] 高梓源[1,2] 包云岗[1,2,3] 王 卅[1,2]

[1](处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

[2](中国科学院大学计算机科学与技术学院 北京 100049)

[3](北京开源芯片研究院 北京 100080)

(liuguodong19z@ict.ac.cn)

AdaptDNN: An Adaptive and Scalable LLM Distributed Training System

Liu Guodong[1,2], Zhu Jiaqi[1,2], Gao Ziyuan[1,2], Bao Yungang[1,2,3], and Wang Sa[1,2]

[1] (*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[2] (*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 100049)

[3] (*Beijing Institute of Open Source Chip*, *Beijing* 100080)

Abstract As the number of parameters in deep learning models continues to increase, the cost of training also keeps rising. To reduce training costs, using spot instances provided by cloud service providers has become a viable solution. Spot instances are priced at only 30% of normal instances, which can significantly lower training costs. However, despite the low cost of elastic instances, there is a risk of them being reclaimed at any time, posing new challenges to the stability of the model training system. To address the fault tolerance issue in the context of elastic instances, existing work mainly falls into two categories: checkpoint-based fault tolerance and redundancy-based fault tolerance. Checkpoint-based solutions incur substantial overhead, while redundancy-based solutions impose certain limitations on the model's parallelism strategy, leading to suboptimal training efficiency. We propose a solution for training in the context of spot instances that leverages the grace period of spot instances to back up training progress, thereby reducing fault tolerance overhead. It also employs the bottleneck alleviation approach to adjust the parallelism strategy, maximizing the use of available cluster resources and enhancing training efficiency. Experimental results show that this solution not only achieves low-cost fault tolerance but also ensures training efficiency, allowing for efficient completion of model training tasks in the context of spot instances and reducing overall training costs.

Key words deep learning; neural network; distributed training; spot instance; fault tolerance

摘 要 随着深度学习模型参数量的不断增加,训练成本也在不断上升.为了减少训练成本,使用云服务厂商提供的弹性实例训练模型成为了一个可行的解决方案.弹性实例的价格仅为正常实例的30%,可以有效降低训练成本.虽然弹性实例价格低廉,但随时都有被回收的风险,对模型训练系统的稳定性提出了新的挑战.为了解决弹性实例场景下的容错问题,现有的工作主要有2类解决方案,分别是基于存盘点的容错和基于冗余性的容错.基于存盘点的方案开销较大,而基于冗余性的方案则对模型的并行策略有一定的限制,导致训练效率并非最优.AdaptDNN是一种自适应可伸缩的大模型分布式训练系统.在弹性实例训练场景,AdaptDNN利用弹性实例宽限期完成训练进度的备份,降低容错开销;并利用"瓶颈消除"思

收稿日期: 2024-06-21; **修回日期**: 2025-05-09

基金项目: 中国科学院战略性先导科技专项(XDA0320000, XDA0320300); 国家自然科学基金项目(62172388)

This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (XDA0320000, XDA0320300) and the National Natural Science Foundation of China (62172388).

通信作者: 王卅(wangsa@ict.ac.cn)

想调整模型并行策略,最大化利用集群可用资源,提升训练效率. 实验结果表明,AdaptDNN 既能实现低成本容错,又能保证模型训练效率,从而可以在弹性实例场景高效完成模型训练任务,降低模型训练成本.

关键词 深度学习;神经网络;分布式训练;弹性实例;容错

中图法分类号 TP316

DOI: 10.7544/issn1000-1239.202440560 **CSTR:** 32373.14.issn1000-1239.202440560

近年来深度学习发展迅速,随着模型结构的不断更迭,以及模型参数量的不断增加,深度学习模型在越来越多的领域表现出卓越的性能,例如聊天机器人 ChatGPT、视频生成工具 Sora 等. 随着模型规模的不断增长,训练模型所需算力也不断增加,根据 OpenAI 的统计数据[1], 自 2012 年以来,深度学习模型训练所需的算力每 18 个月翻 35 倍,与单个计算设备的算力之间存在着显著的差距. 因此,大模型的训练往往需要在大规模的 GPU 集群上进行分布式训练,并使用各类并行方案对训练进行优化,比如数据并行[2]、张量并行[3] 和流水线并行[4] 等. 然而,GPU 集群的购买及租用成本往往十分昂贵,例如,使用亚马逊提供的价格最低的 GPU 完整训练一次 GPT-3 175B[5] 模型,都需要 150 万 GPU 小时,耗费约 460 万美元[6].

如何有效控制或降低大模型高昂的训练成本成为学术界和工业界都必须面对的一个关键问题. 学术界的一个思路是利用弹性实例训练,近年来亚马逊等云厂商提供了一种价格更为低廉的弹性实例(spot instance). 弹性实例提供了远低于正常实例的租用费用(仅 30%),相应的代价是这些实例随时可能被回收,用户需要自行应对实例变化. 尽管不确定性高,弹性实例以其低廉的价格仍旧吸引了大量研究者的关注,如 Bamboo[7] 等工作已实现在弹性实例上进行低成本训练. 对于已经拥有大规模 GPU 集群的工业界,它们的思路是通过训推混部[8] 来缓解大模型训练带来的资源压力,一些模型训练任务可能被调度到模型推理的集群,利用闲置 GPU 时间进行训练. 然而与弹性实例训练面临的问题相同,当推理集群负载增加,训练任务也将面临资源被随时收回的可能性.

当部分弹性实例被回收时,如果训练系统不具有自适应可伸缩性,无法根据不同的实例数量调整并行方案,训练将会一直处于停滞状态,直到新的实例再次加入训练集群,造成其他可用实例被长时间闲置. 同时,被回收实例上产生的训练进度将被丢失,进一步增加模型训练的时间. 因此,本文提出一个问题:如何在随时会被回收的弹性实例场景中,实现自适应可伸缩性的深度学习模型训练? 具体而言,要解决这个问题主要面临三大挑战:

1)模型部署时. 在实例弹性变化的场景下,如何进行大模型的高效并行部署.

2)实例回收时. 如何尽可能减少训练进度的丢失,保障实例回收后的“故障”恢复.

3)实例回收后. 如何基于现有可用实例快速寻找出新的高效并行方案,调整部署继续执行训练任务.

对于第 1 个挑战,现有的模型分布式训练系统,在设计并行方案时往往假定集群的计算资源是固定的,不存在计算资源可变的情况. 因此,现有系统所采用的并行方案往往缺乏灵活性,无法适用于弹性实例场景. 例如,由 NVIDIA 设计的 Megatron-LM[9] 训练系统,在对 GPT-3 模型进行分布式训练时,会将 GPT-3 模型切分为多个相等大小的流水线阶段,并为每个流水线阶段分配相同数量的计算节点,从而实现负载均衡. 然而,如果我们直接将该方案应用于弹性实例场景,当某个流水线阶段的部分实例被回收后,我们可能无法充分利用剩余的所有可用实例. 例如,当我们有 16 个实例时,Megatron-LM 可以构造长度为 8 的流水线,每个流水线阶段分配 2 个实例,当某个实例被回收时,由于 Megatron-LM 要求每个阶段的计算资源相同,此时每个阶段只能使用 1 个实例,从而造成 7 个可用实例被闲置.

在本文中,我们提出了一种异构流水线并行方案,在异构流水线中,每个流水线阶段都可以分配到不同数量的计算资源,相应地,每个流水线阶段的负载量和模型的并行方式都可以不同. 例如,当仅剩 15 个实例时,我们可以构造一条长度为 5 的流水线,其中每个阶段分别有 4, 4, 4, 2, 1 个实例,同时我们可以为每个阶段分配不同数量的神经网络层. 这样的异构流水线使得我们的系统能适应不同数量的计算资源,当部分弹性实例被回收,我们总能利用剩余可用实例数量找到可行的并行方案.

对于第 2 个挑战,当弹性实例被回收时,该实例的所有数据都将变为不可用的状态. 这本质上是一个容错问题,可以使用故障恢复的策略继续训练模

型. 目前已有的分布式训练的故障恢复策略主要有2类：一类是基于Checkpoint（存盘点）[10-11]的故障恢复策略[7,12]，另一类是基于冗余性的故障恢复策略. 基于Checkpoint的策略会每隔一段时间保存一次模型的参数到磁盘，当故障发生时再加载最新的模型参数. 该策略会丢失自上一次保存Checkpoint以来所有的训练进度，因此减少丢失进度的方法就是提高保存Checkpoint的频率. 然而保存Checkpoint的时候，训练需要暂停以防止模型参数被修改，因此频率太高将影响训练的正常进行. 基于冗余性的故障恢复，主要是利用模型数据并行时不同实例参数的冗余性，当故障发生时，从其他实例恢复丢失的模型参数. 而模型的参数是每一个训练轮次后更新1次，因此基于冗余性的策略将丢失1个轮次的训练进度.

本文提出一种基于宽限期（grace period）的自适应故障恢复机制，在不引入冗余性的前提下，实现快速的故障恢复. 主要思想如下：云厂商在提供弹性实例时都会给出一定的宽限期，例如，在实例被回收前30 s时会提前通知. 我们可以在宽限期内完成被回收节点重要数据的传递，包括当前训练轮次产生的梯度值、上一轮次更新后的模型参数等. 通过传输梯度值，我们可以实现不丢失训练进度的故障恢复. 如果宽限期的时间不足以完成所有数据的传输，则我们将综合基于冗余性和Checkpoint的机制，在给定宽限期内选择最高效的恢复方案.

对于第3个挑战，当某个弹性实例被回收后，模型的并行训练方案需要重新调整部署，以适应新的实例数量. 目前已有的自动并行方案如Alpa[13]和Aceso[14]等都是离线搜索，只考虑并行方案的训练性能，不关注并行方案之间调整的开销，而对并行方案进行在线调整时，调整开销、故障频率等因素都会影响整体性能. 例如，当实例被频繁回收时，更好的调整策略是尽快找到可行的并行方案，并最小化模型的参数传递开销，而非寻找吞吐量最高的并行方案. 除此之外，现有工作由于必须基于一定的限制进行调整，使得并行方案的性能本身也并非最优. 例如，基于冗余性的故障恢复策略由于必须使用数据并行，当实例被回收时，剩余节点如果无法构造足够数量的数据并行组，则无法完成并行方案的调整.

本文提出了一种在线高效并行方案调整算法，该算法基于离线并行方案搜索算法Aceso[14]并对其进行优化，使其能够兼顾调整开销、实例回收频次等因素实现在线高效并行方案调整. 为达到这一目标，我们提出一个新的指标“下一次故障前的有效轮次数”，用于评估在弹性实例场景下一个并行方案的实际训练效率. 该指标综合考虑了一个并行方案的训练性能，以及调整为该方案所需的时间开销，并综合历史数据预估下一次实例回收发生的时间间隔，从而计算得到一个并行方案在下一次实例回收前可以完成的训练轮次数. 我们将“下一次故障前的有效轮次数”作为Aceso搜索算法的新目标函数，从而实现兼顾调整开销、实例回收频次的同时，在线高效搜索并行方案，提高了实例被回收后的训练效率.

我们在不同规模的GPT-3模型上进行了实验，实验结果表明，与当前最好的工作Oobleck[12]相比，AdaptDNN能显著提升弹性实例场景下的训练效率. 当可用实例数量在8~16之间弹性伸缩时，AdaptDNN将训练的总体吞吐量提升1.4倍，将并行方案的调整开销降低至25%，同时，使用同样数量的GPU，AdaptDNN可训练的模型大小是现有工作的2.5倍.

1 相关工作

近年来，由于模型规模、训练数据和集群规模的逐渐增加，大模型的分布式训练成为了一个备受关注的课题. 各类分布式训练策略的提出，使得大模型分布式训练的性能得到了显著提升. 随着集群规模的进一步增大，训练的性能不再是唯一需要关注的指标，训练任务的可伸缩性和容错能力在集群资源可能发生变化的情况下尤为重要. 如何在使用各类高效并行方案的情况下，使系统具有弹性伸缩的能力，从故障中快速恢复，成为了一个具有挑战性的研究问题. 本节将介绍现有的一些工作在该问题上的探索，主要包括各类常见的分布式训练策略（1.1节）、无故障时的弹性伸缩（1.2节）以及大模型训练的故障恢复（1.3节）.

1.1 大模型分布式训练策略

深度学习模型的训练过程，一般采用“反向传播算法”. 该算法的过程如下：首先将训练数据输入神经网络，让网络的权重（即模型的参数值）与传入的数据进行一系列的运算，最终将输出的结果与数据的标签传入损失函数计算损失值，然后根据损失值计算梯度值，并通过链式法则，将梯度值反向传播到所有神经网络层，并更新权重值. 一般训练用的数据集会被划分为多个批次，模型会在每个批次上重复该过程，一个批次的计算往往也被称为一个训练轮次（iteration）. 经过多个轮次的训练，最终即可得到一个在所有数据上表现良好的模型.

近年来深度学习模型的参数量、训练数据量都在快速增长，单个节点的内存空间有限，已经难以支持完整模型的训练，因此出现了多种分布式训练策略.

1)数据并行. 数据并行[15]是指将模型的参数复制多份，每台机器保存一份完整的模型参数，但将训练数据分成多份，每台机器只保存一部分训练数据. 所有机器分别进行运算，各自在一部分训练数据上得到梯度值，在每个批次的数据算完后多机之间同步一下梯度值. 数据并行实现简单，但最大的局限性是不能够减少内存资源的占用，每台机器拥有完整的模型参数，因此对于单机无法容纳的模型，数据并行是无法应用的. 因此出现了对模型参数进行划分的张量并行和流水线并行.

2)张量并行. 张量并行[16-19]是将某些算子的参数切分为几个部分，每台机器各保存一部分，这样每台机器算出来的结果将是不完整的，需要在每次完成计算后进行一次结果的同步. 张量并行虽然能极大程度减少每台机器内存的占用，但一般会引入比数据并行更大的通信开销，因为数据并行只需要在每一批数据计算完毕之后同步梯度值，而张量并行需要在每一个训练数据的运算过程中同步中间激活值. 因此，张量并行对通信带宽要求较高，一般只应用于带有高速通信链路的节点内部.

3)流水线并行. 为了解决张量并行引入的通信开销问题，学术界又提出了流水线并行[20-22]. 流水线也是对模型参数的分割，但并不会将每一层神经元的参数分成几份，而是在模型的层与层之间进行切分，类似 CPU 中的指令流水线，将整个模型的计算分为多个流水线阶段，每台机器各负责其中一个阶段的计算，只需要在机器之间传递中间激活值即可，因此流水线并行的通信开销比模型并行更小.

4)混合并行. 以上 3 种并行策略各有利弊，近年来很多工作尝试全部或部分地结合了这 3 种方案，在不同的场景下找到了更为高效的分布式训练方式. 例如 FlexFlow[3]结合了数据并行和模型并行，DAPPLE[23]结合了数据并行与流水线并行，Megatron-LM[9]，Alpa[13]，Aceso[14]结合了数据并行、模型并行、流水线并行.

1.2 可伸缩的分布式训练

对模型分布式训练的可伸缩性研究，主要分为 2 类：一类工作关注单个训练任务的可伸缩性；另一类工作对深度学习集群的调度器进行研究，通过动态调节不同任务的资源，提高集群的整体利用率.

1)单个任务的可伸缩性. 现有的很多大模型训练框架已经原生支持了模型训练的弹性伸缩，例如 Torch Elastic[24]，Eastic Hovorod[25]等，但此类框架只支持对数据并行的弹性伸缩，即在模型使用数据并行训练时，通过修改数据并行组的大小以及每组的批处理大小，从而实现模型训练在不同资源条件下的扩容和缩容. 此外，还有一些研究工作[26]对任务伸缩的开销进行了分析，提出了更加高效的伸缩机制，但也缺乏对张量并行和流水线并行的支持.

2)集群任务的可伸缩性. 许多集群调度器的工作通过对深度学习任务特性进行分析，总结出不同的调度器优化目标，例如基于反馈调节机制的 Gandiva[27]、综合考虑训练性能和模型收敛效率的 Pollux[28]以及应用 Serverless 思想优化集群调度的 ElasticFlow[29]等，此类工作更关注集群整体利用率而非单个任务的训练效率，且对单个训练任务的调节方式也是基于数据并行的可伸缩性.

此类弹性伸缩工作往往主动调整训练任务的资源和并行方案，不具有应对未知故障的容错能力，因此无法被应用到弹性实例场景.

1.3 大模型训练的故障恢复

分布式系统的故障恢复，往往需要依赖于某种形式的冗余性. 一类被广泛采用的方式是通过周期性地保存 Checkpoint 来维护冗余备份. 近年来有更多针对大模型分布式训练的特性的工作，提出了其他类型的冗余性，主要包括利用模型参数的冗余和引入冗余计算来实现快速的故障恢复.

1)基于 Checkpoint 的故障恢复. 利用 Checkpoint 保存和加载模型状态，是模型故障恢复的常用方法. 该方案会在模型训练时，每隔一定的训练轮次将模型的状态写入磁盘. 当故障发生时，系统会从磁盘中读取最新的模型状态. Checkpoint 的写入和加载都需要进行磁盘读写，耗费的时间较长，因此保存 Checkpoint 过于频繁会影响模型训练的性能. 为解决该问题，一些工作提出了基于内存的 Checkpoint 优化方案[30-32]，先将 GPU 中的模型状态同步写入到主机的内存中，再将主机内存的数据异步写入到磁盘中. 异步写入磁盘时不会阻塞训练任务，从而减小写入 Checkpoint 对性能的影响. 一些其他工作[33-35]则进一步优化了在不同并行方案之间保存和加载 Checkpoint 的性能. 然而，基于内存的 Checkpoint 方案仍有不足. 首先，将模型状态从 GPU 写入到主机内存的过程仍会阻塞训练任务，因此过于频繁地保存 Checkpoint 仍然会影响模型训练速度. 其次，当故障发生时，该方案可能会丢失训练的进度. 因为在加载模型状态时，需要使用最新的完整的 Checkpoint. 如果在主机内存异步写入磁

盘时发生故障，那么该 Checkpoint 不可用. 系统只能使用上一次保存的模型状态，这会丢失一些轮次的训练状态，造成较高的故障恢复时间.

2）基于参数冗余性的故障恢复. 为了提高故障恢复效率，避免保存和加载 Checkpoint 的开销，一些工作提出使用数据并行内在的冗余性进行恢复. 在数据并行中，包含一个模型完整状态的机器构成一个数据并行组，不同数据并行组之间的模型状态相同，因此当发生故障且故障节点数小于数据并行组个数时，总能从剩余数据并行组中恢复丢失的状态. Oobleck[12] 利用数据并行的冗余性实现了模型训练时的故障恢复，并结合流水线并行将模型参数分割到多台机器中，实现了大模型的弹性训练. 对于每个数据并行组，Oobleck 会利用分治法计算出合适的流水线并行方案来分割模型. 当出现节点故障时，Oobleck 会根据剩余的节点中包含的模型状态，重新分配流水线并行方案，并根据新方案在不同节点间传递模型的状态，从而实现模型的重构. Parcae[6] 也利用了数据并行的冗余性进行故障恢复，但与 Oobleck 不同的是 Parcae 会预测未来节点损坏概率并提前做出并行方案的调整. SlipStream[36] 则是在故障发生后不调整并行方案，而是将发生故障的数据并行组的一部分计算任务重新路由到其余并行组，实现了故障状态下的负载均衡. 但是，利用数据并行实现冗余的方案存在一定的局限性. 首先，该方案会限制系统能训练的模型尺寸. 在总 GPU 内存资源有限的情况下，模型尺寸越大，能支持的数据并行组的个数越少，因此同等规模的 GPU 集群此类方案能够支持的模型大小会严重受限. 其次，该方案在故障恢复后会丢失一部分训练进度. 由于数据并行组仅在每轮训练结束时才进行模型状态的同步，如果节点故障发生在一轮训练的中间时刻，那么该轮训练的进度将在故障恢复后丢失. 系统只能回退到上一轮训练后的模型状态重新执行.

3）基于冗余计算的故障恢复. 除了利用参数的冗余性，也有一些其他工作通过引入计算的冗余性实现故障恢复. Bamboo[7] 通过在训练流水线中填充冗余计算提高故障恢复效率. 核心思想是让每个节点在完成自身的计算任务之外，额外多做一份其他节点的计算任务，将计算结果作为冗余备份. 在故障发生时直接使用已备份数据恢复故障节点的该步计算，使得训练能够继续进行. 但 Bamboo 存在多方面不足，过多的冗余计算降低了训练的吞吐量，消耗的 GPU 显存过大使得对更大的模型训练支持性差. 根据 Oobleck 的实验数据，Bamboo 所能支持的模型大小仅为 Oobleck 的 1/20.

虽然已有工作可以在传统的容错场景实现高效的故障恢复，但使用弹性实例进行训练时，这些工作的效率并不高. 主要原因是从冗余备份进行恢复的做法会导致每次恢复都丢失一段时间的训练进度，而弹性实例的故障更加频繁，因此会导致大量的计算被浪费. 根据 Bamboo 的测试数据[7]，一个 64 卡的弹性实例集群，在 24 h 内发生了 127 次抢占. 如果使用 Checkpoint 进行故障恢复，约有 77% 的时间被浪费在加载 Checkpoint 和重算丢失的训练进度. 虽然 Oobleck 通过利用数据并行的冗余参数进行恢复，但每次恢复后仍然会丢失一个训练轮次的进度. Bamboo 通过引入冗余计算降低训练进度的丢失，但冗余计算却会导致大量的算力被浪费.

另一方面，弹性实例的频繁故障往往需要训练系统有快速调整训练方案的能力. 现有工作的并行方案大多缺乏灵活性，基于冗余计算的 Bamboo 选择将模型切分为均等的流水线，当集群中 GPU 数量难以被均等切分时，该类方案往往无法利用所有 GPU 进行训练，造成了大量计算资源的闲置. 基于 Checkpoint 的方案大多也采用了此类均等流水线的设计[11]，另一部分工作[30] 则不对并行方案进行调整，而是选择维护额外的资源池，用于替换故障节点，此类方案也造成了大量的资源闲置，进一步增加了训练成本. 基于参数冗余性的 Oobleck 通过构造多条长度不同的流水线实现了对 GPU 的充分利用，但该方案对数据并行有强依赖，导致训练性能可能不是最优.

与现有工作不同，本文将根据弹性实例所具有的“宽限期”特性，有针对性地提出一套故障恢复方案，通过利用“宽限期”进行故障恢复，可以实现更细粒度的状态保存，降低恢复开销. 同时，本文也提出了一套灵活可变的并行方案以及在线的并行方案调整算法，可充分利用所有硬件资源，从而在降低恢复开销的同时，保证了训练的性能.

2 系统概览

AdaptDNN 是一个自适应可伸缩的大模型分布式训练系统. AdaptDNN 使用了一种灵活的异构流水线并行方案（第 3 节），可以适应不同的实例数量，在集群可用资源发生变化时能够充分利用所有计算资源. 同时，AdaptDNN 采用基于宽限期的自适应故障恢复机制（第 4 节），从而避免了频繁的 Checkpoint 开

销以及在系统中引入冗余性的开销. 最后, AdaptDNN 采用了一个在线高效并行方案调整算法(第 5 节), 从而可以快速找到资源变化时新的并行方案, 实现对训练任务的快速恢复.

如图 1 所示, AdaptDNN 在对模型进行分布式训练时(①), 会在 CPU 端同时开始搜索集群中每个节点被回收后新的并行方案(②). 当底层的运行时系统被告知某一节点被回收时(③), AdaptDNN 会按照预先找到的并行方案调整模型的部署, 并在节点之间传输缺失数据, 从而实现在宽限期内快速完成故障恢复(④), 然后底层运行时系统按照新的并行方案继续训练(⑤).

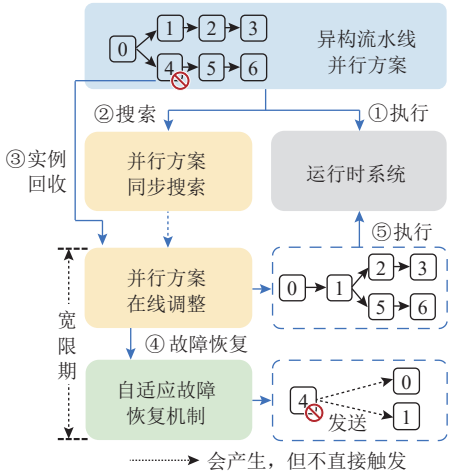


Fig. 1 System overview

图 1 系统概览

3 模型部署时: 异构流水线并行方案

现有的模型分布式训练相关工作都是利用混合并行来充分利用硬件资源. 但现有工作往往假定集群的计算资源是稳定的, 因此采用的并行方案往往缺乏适用于计算资源可变场景的灵活性. 如图 2 所示, Megatron-LM 提出, 在 Transformer[37] 层内部使用张量并行缓解单个设备的内存压力; 在 Transformer 层之间使用流水线并行缓解模型划分带来的通信压力; 最后使用数据并行将模型扩展到更多设备上进行训练. 整个模型训练的配置, 可以用: *tp*(张量并行大小)、*dp*(数据并行大小)和 *pp*(流水线并行大小)3 个参数来指定, 只要满足 *tp*, *dp*, *pp* 的乘积等于 GPU 数量即可. 因此这样的并行方案也被称为“3D 并行”. 然而将 3D 并行应用到弹性实例集群中, 可能造成集群计算资源的闲置. 例如, 当 1 个或多个实例被回收, 剩余实例的数量可能无法被分解为任何 *tp*, *dp*, *pp* 的乘积(假设 *tp*, *dp*, *pp* 都大于 1).

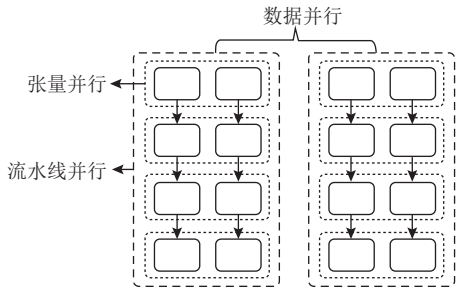


Fig. 2 3D parallelism

图 2 3D 并行

为了能让模型的并行方案适应于任意数量的 GPU, 我们提出了异构流水线并行方案. 异构流水线是指, 流水线的不同阶段可能被分配不同数量的 GPU, 因此, 每个阶段也可能用不同的方式来组合使用数据并行和张量并行. 如图 3 所示, 初始流水线中包含 16 个 GPU, 如果我们用每阶段的 GPU 数量来表示该流水线, 则可以将其表示为 4-4-4-4. 在某时刻, 流水线第 2 阶段的 2 个实例被回收, 则我们可以将流水线调整为 4-2-4-4. 如果此时流水线第 3 阶段的 1 个 GPU 进一步被回收, 第 3 阶段仅剩 3 个 GPU, 考虑到大部分底层运行时系统仅支持偶数数量的模型切分, 我们可以进一步将第 3 阶段拆分成 2 个阶段, 则流水线可以表示为 4-2-2-1-4.

流水线计算资源的异构性也意味着负载的异构性. 持有更多计算资源的阶段应该承担更多的计算

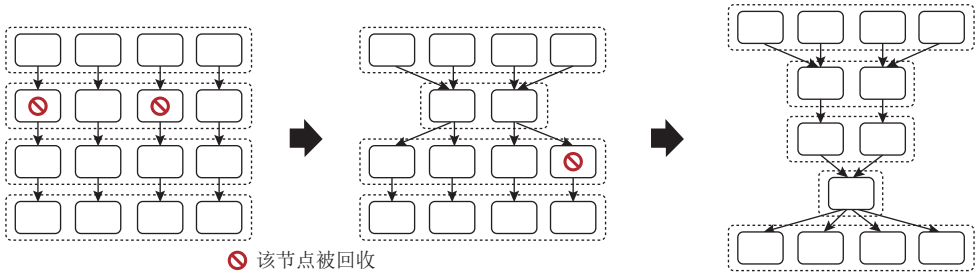


Fig. 3 Heterogenous pipeline

图 3 异构流水线

任务. 然而流水线负载的划分不能简单地跟计算资源数量成正比, 因为不同阶段采用的不同并行方案可能导致不同的计算效率(例如, 采用张量并行时会引入通信开销, 这就导致使用张量并行阶段的执行时间变长). 异构流水线使得并行方案的搜索空间进一步增大, 在完整的搜索空间里寻找最优解, 将引入极高的搜索开销. 受到自动并行系统 Aceso[14] 的启发, 我们选择将异构流水线方案的搜索问题映射为一个资源调优问题, 即从一个初始方案出发, 通过在流水线不同阶段之间重新分配计算资源和计算任务, 逐步优化其中的瓶颈阶段, 直至性能无法被进一步优化为止. 关于搜索算法的更多细节将在第 5 节进行介绍.

4 实例回收时: 利用宽限期实现故障恢复

4.1 弹性实例宽限期

云服务厂商在提供弹性实例时, 往往会给出一定的宽限期, 即在实例回收之前的一定时间会被提前通知. 例如, 亚马逊公司给出的宽限期约为 2 min, 谷歌和微软云公司给出的宽限期约为 30 s[10].

本文提出利用弹性实例宽限期来传输模型重构所需要的数据. 在传统容错场景中, 由于不存在这样的宽限期, 导致必须使用一定的冗余性来完成故障恢复, 而冗余备份本身则带来了较大的开销. 在弹性实例的场景中, 由于宽限期的存在, 我们可以在实例回收真正发生之前有针对性地完成实例数据的传输和保存, 大大减少了容错的成本, 提高了系统的性能.

4.2 利用宽限期实现更细粒度的故障恢复

如图 4 所示, 现有工作在进行故障恢复时, 往往是对模型的参数进行重新部署, 例如 Checkpoint 方案会加载上一次保存的模型参数, 基于冗余性的方案 Oobleck 也是在设备之间传输缺失的模型参数. 但模型参数是每一个训练轮次完成后才会更新一次, 这使得现有方案在完成故障恢复后, 都会丢失至少 1 个训练轮次的进度, 需要重新计算自上一次参数更新以来产生的梯度值. 因此, 现有工作的故障恢复时间包含了参数读取/传输开销以及重做丢失计算的开销. Checkpoint 机制和 Oobleck 的故障恢复时间可以分别表示为

$$T_{\text{ckpt}} = T_{\text{load}} + T_{\text{redo}}, \tag{1}$$

$$T_{\text{oobleck}} = T_{\text{w}} + T_{\text{redo}}, \tag{2}$$

其中 T_{load} 是加载 Checkpoint 的时间, T_{redo} 是重做丢失的计算的时间, T_{w} 是在节点间传输模型参数的时间.

由于从磁盘中读取 Checkpoint 的带宽较低, 因此 T_{load} 往往比 T_{w} 更大. 而且, 由于 Checkpoint 的频率较低, 但 Oobleck 传输的模型参数每个训练轮次都会更新, 因此 Checkpoint 方式的 T_{redo} 比 Oobleck 更大. 当模型规模较大时, 一个轮次的时间也会较长, 丢失的计算进度也会增多, 这导致 T_{redo} 进一步增大.

由于弹性实例场景下宽限期的存在, 我们可以在故障真实发生之前提前进行处理, 因此我们提出一种更加细粒度的模型重构方案: 在实例被回收之前的宽限期内, 按照预先找到的新的并行方案, 将被回收实例在本轮次产生的梯度值和它持有的模型参

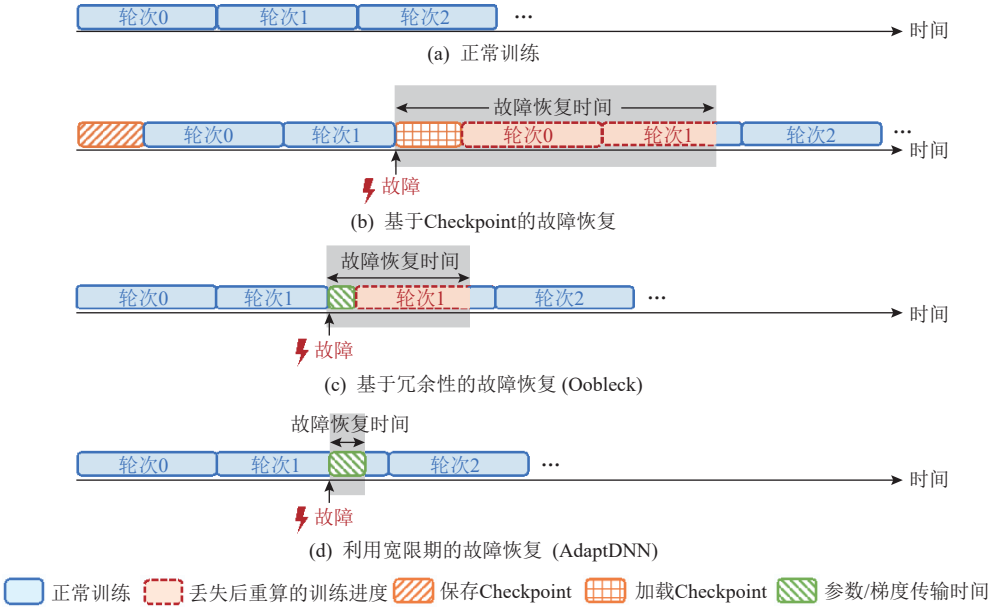


Fig. 4 Comparison of model reconstruction overhead of different fault-tolerant methods

图 4 不同容错方式的模型重构开销对比

数传输给其他节点. 因此, 当故障恢复完成后, 其他节点持有本轮次的训练进度, 模型可以从停止的地方接着训练, 不会损失训练进度, 也就不需要重做任何计算. 此时, 重构开销可以表示为

$$T_{\text{AdaptDNN}} = T_{\text{g}} + T_{\text{w}}, \tag{3}$$

其中T_{g}是在节点间传输梯度值的时间.

4.3 基于不同宽限期和带宽的自适应故障恢复机制

尽管我们可以利用弹性实例的宽限期进行故障恢复, 但当网络带宽较低或需要传输的参数量和梯度值较多时, 宽限期的时间可能不足以完成所有数据的传输. 因此, 我们对需要传输的数据进行了分析, 并结合基于冗余性的故障恢复和 Checkpoint 机制, 提出能适应于不同宽限期和网络带宽的故障恢复机制. 该机制的核心思想如下 3 方面.

1)在宽限期内传输“无冗余性”的参数即可. 我们发现并非所有参数都具有同样的传输优先级. 在 AdaptDNN 采用的异构流水线并行方案中, 模型每一层的冗余性可能是不同的. 当采用数据并行时, 该层的模型参数天然具有冗余性; 当采用流水线并行与张量并行, 则不存在冗余性. 具有冗余性的参数, 在宽限期结束后也能从其他设备恢复, 因此当宽限期不足以完成所有的参数传输, 我们只要保证在宽限期内完成 “无冗余性”的参数传输即可. 由于每轮次产生的梯度值也不具有冗余性, 因此也需要在宽限期内传输.

2)宽限期较短时, 回退到基于冗余性的故障恢复. 如果我们无法在宽限期内完成所有梯度值和“无冗余性”的参数传输, 则无法采用细粒度的模型重构机制. 此时, 我们采用基于冗余性的恢复机制: 在宽限期内仅完成“无冗余性”的参数传输, 在宽限期后从其他节点恢复冗余参数即可. 由于无法在宽限期内传输本轮次产生的梯度值, 此时的故障恢复与基于冗余性的恢复机制相同, 有额外的重计算开销.

3)宽限期较短且模型不存在冗余性, 回退到 Checkpoint 机制. 如果我们无法在宽限期完成“无冗余性”的参数传输, 则基于冗余性的恢复机制也无法使用(因为整个系统中模型的状态不再完整), 此时我们回退到使用 Checkpoint 进行恢复.

因此, AdaptDNN 的自适应重构机制的故障恢复开销可以表示为

$$T_{\text{AdaptDNN}} = \begin{cases} T_{\text{g}} + T_{\text{w}}, & \text{if } T_{\text{g}} + T_{\text{wn}} \leqslant T_{\text{grace}}, \\ T_{\text{redo}} + T_{\text{w}}, & \text{if } T_{\text{wn}} \leqslant T_{\text{grace}} < T_{\text{g}} + T_{\text{wn}}, \\ T_{\text{redo}} + T_{\text{load}}, & \text{if } T_{\text{grace}} < T_{\text{wn}}, \end{cases} \tag{4}$$

其中T_{wn}是指传输“无冗余性”参数所需时间, T_{grace}是宽限期大小.

5 实例回收后: 基于“瓶颈消除”思想调整并行方案

在部分实例被回收后, 为了能继续模型的训练, 我们需要在剩余可用资源条件下调整并行方案. 由于异构流水线引入的复杂性, 使得并行方案的搜索空间进一步增大, 基于动态规划的一些自动并行工作难以被应用到我们的场景. 受到自动并行工作 Aceso 的启发, 我们发现弹性实例场景下的并行方案搜索问题可以映射为一个资源调优问题, 并使用“瓶颈消除”思想来求解.

但 Aceso 也无法被直接应用到弹性实例的场景, 因为 Aceso 是离线搜索, 其搜索目标为性能最优的并行方案, 而忽略了并行方案调整的时间开销. 而这一开销对于弹性实例场景至关重要(6.4 节), 因为在网络带宽较低、宽限期较小或者故障频繁的场景下, 性能最优的方案可能并不是最优解. 因此, 我们提出了一种高效的并行方案在线调整算法, 通过使用新的搜索目标“下一次故障前有效的轮次数”, 从而实现了在性能与开销之间的权衡, 确保了在弹性实例场景下的训练效率.

虽然我们可以通过“瓶颈消除”的思想提高并行方案的搜索速度, 但如果在实例被回收时才搜索新的并行方案, 仍会占用宽限期的时间, 可能导致我们无法在宽限期内完成故障恢复所需的数据传输. 我们发现, 搜索过程是可以与模型的训练过程同步进行的. 因此, 当训练开始时, 我们在 CPU 端同步开始搜索集群中每个节点被回收后的最优并行方案. 当实例回收发生时, 我们可以直接按照预先找到的方案调整模型的部署, 从而使宽限期的时间可以完全用于模型的故障恢复.

5.1 瓶颈消除思想

“瓶颈消除”是计算机领域的经典思想, 在流处理系统中就被用于优化系统性能[38-40]. Aceso 提出使用“瓶颈消除”思想迭代优化给定并行方案的性能, 直至无法找到更好的方案为止. 其核心思想如下: 当给定一个初始的并行方案, 通过性能模型评估其中哪一部分是性能瓶颈, 并采用与之对应的策略优化该性能瓶颈. 例如, 当一个流水线阶段的内存压力过大, 可以通过增加该阶段的张量并行大小来降低内存占用, 或者将较为空闲的流水线阶段的计算资源

重新分配给该阶段使用.

图 5 展示了使用“瓶颈消除”思想搜索并行方案的一个例子. 一个拥有 4 个算子的模型, 其初始并行方案为 2 个流水线阶段, 但这样的切分会导致流水线阶段 1 的内存压力过大, 使得阶段 1 成为了整个系统的性能瓶颈. 因此, 可以将阶段 2 的 2 个 GPU 重新分配给阶段 1, 从而缓解阶段 1 的内存压力. 但此时阶段 2 成为了新的性能瓶颈, 因为其算力减少, 导致了执行时间变长, 我们可以进一步选择将阶段 2 的一个算子重新分配给阶段 1 执行, 由此以来, 我们就实现了 2 个阶段的负载均衡, 找到了一个比初始方案更好的并行方案.

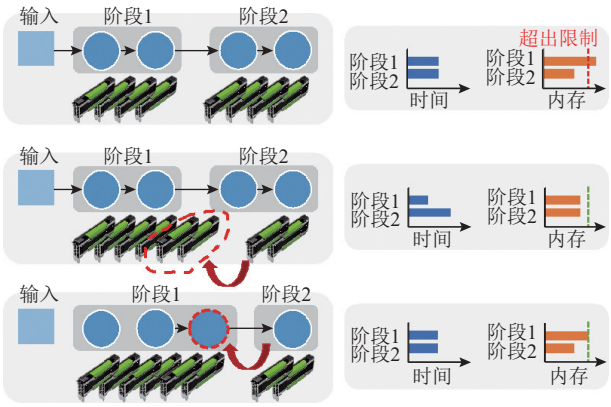


Fig. 5 Searching for parallel schemes by using bottleneck alleviation idea

图 5 使用“瓶颈消除”思想搜索并行方案

我们发现, “瓶颈消除”的思想也能被应用于弹性实例场景, 用于搜索实例被回收后新的并行方案. 与 Aceso 的离线搜索不同, 弹性实例场景中的性能瓶颈是随着实例回收而动态产生的. 如图 6 所示, 当某些弹性实例被回收时, 我们可以假设其所在流水线阶段的其他实例承担该阶段的所有计算任务, 因此

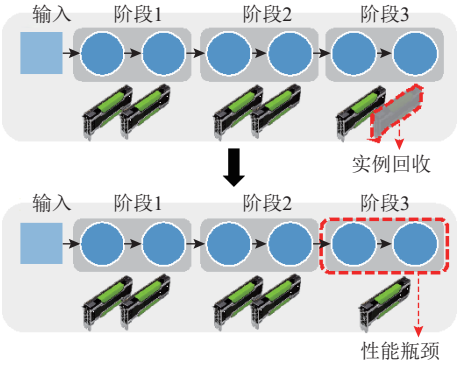


Fig. 6 The first performance bottleneck in fault tolerance scenario

图 6 容错场景下的第 1 种性能瓶颈

该阶段将成为一个性能瓶颈(计算量不变, 但算力减少了). 我们可以将此并行方案作为初始并行方案来优化该瓶颈阶段的性能, 并重复这样的“发现瓶颈-消除瓶颈”过程, 直至找到一个比初始方案性能更好的方案.

当某些弹性实例被回收时, 也可能出现另一种性能瓶颈. 如图 7 所示, 某一流水线阶段的所有实例可能都被回收, 导致该阶段无计算资源. 此时, 可以假定该阶段的计算任务被其相邻阶段的实例承担, 则该相邻阶段将成为一个性能瓶颈. 类似地, 我们可以应用“瓶颈消除”思想优化该瓶颈.

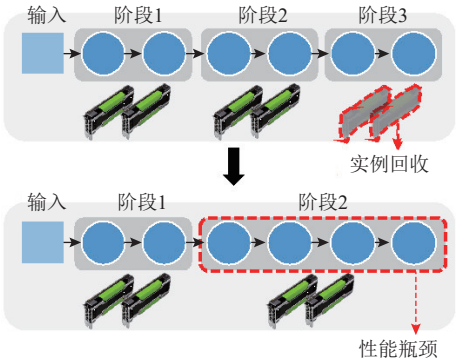


Fig. 7 The second performance bottleneck in fault tolerance scenario

图 7 容错场景下的第 2 种性能瓶颈

5.2 瓶颈消除策略

弹性实例场景的瓶颈往往是由于某个流水线阶段丢失了部分实例而造成的负载与算力的不均衡. 只要能实现不同流水线阶段之间的算力/负载均衡, 避免某一个阶段与其他阶段的性能差异过大, 就能提高流水线的整体性能. 因此, 最简单的调整方式是重新选择流水线的切分位置, 使得各个阶段的负载量与其算力成正比, 在不调整并行方式的情况下重新实现负载均衡. 然而, 直接按照不同阶段的实例数量比例切分流水线, 仍然可能不是最优方案, 因为不同阶段所采用的并行方式不同, 相同负载量的计算时间可能有所不同(例如, 张量并行会引入大量的 all-reduce 通信, 同样的并行度下, 张量并行的时间往往大于数据并行的时间[9]). 为了能更有效地消除瓶颈, 我们采用了以下基于贪心的负载均衡策略.

策略 1. 负载均衡. 尝试将瓶颈阶段的一部分负载分配给当前时间最短的流水线阶段, 用性能模型评估重新分配后总性能是否更优, 重复此过程, 直至无法进一步提升性能为止.

该策略的伪代码实现如下:

算法 1. 负载均衡策略.

输入: 并行方案 *Config*, 瓶颈阶段 *BottleStage*, 硬件内存上限 *MemLimit*;

输出: 新的并行方案.

① *CurTime* ← *Config. time*;
② while true
③ *IdleStage* ← *GetIdleStage*(); /*利用性能模型得到当前空闲(时间最短)阶段*/
④ *BottleStage.layers* −−; /*减少瓶颈阶段负载(*layer*)数量*/
⑤ *IdleStage.layers* ++; /*增加空闲阶段负载(*layer*)数量*/
⑥ *Time*, *Mem* ← *PerfModel*(); /*利用性能模型评估 *Config* 的总时间及内存*/
⑦ if *Time* < *CurTime* and *Mem* < *MemLimit*
⑧ *CurTime* ← *Time*;
⑨ else
⑩ 无法进一步提升性能, 撤销对 *Config* 的上一步修改, 并返回 *Config*;
⑪ *BottleStage.layers* ++;
⑫ *IdleStage.layers* −−;
⑬ return *Config*;
⑭ end if
⑮ end while

然而, 使用策略 1 在某些场景下可能无法找到性能更优的并行方案. 例如, 假如瓶颈阶段持有某个内存占用量较大的神经网络层(例如嵌入层). 如果不为该瓶颈阶段分配更多算力用于张量并行, 则该瓶颈阶段将无法被分配更多的负载, 造成算力的浪费.

AdaptDNN 采用了与 Aceso 类似的“资源置换”(resource trading)思想来消除瓶颈: 造成性能瓶颈的原因往往是某一类资源不足, 主要包括计算、通信和内存资源. 常见的一些并行方案的调整策略则是在不同的资源之间进行“资源置换”. 例如, 增加张量并行度, 就是通过引入额外的通信来减少内存资源的使用; “重计算”策略则是通过“以算代存”的形式, 在反向传播过程中重做前向计算, 从而避免保存前向计算产生的中间结果, 实现了以额外的计算来减少内存占用的效果. 我们发现, 各类“资源置换”操作往往会引起多个流水线阶段的并行方式/负载量发生变化, 因此我们将“资源置换”操作分为 2 个类别, 并分别提出以下 2 个瓶颈消除策略.

策略 2. 改变并行方式. 当内存资源成为瓶颈, 减少数据并行, 增加张量并行(以通信换内存); 当计算资源成为瓶颈, 减少张量并行, 增加数据并行(以内存换计算); 当通信资源成为瓶颈, 增加数据并行, 减少张量并行(以内存换通信). 如果瓶颈阶段的实例数量有限, 并行度无法进一步增加, 则将当前最空闲(时间最短)的流水线阶段的部分实例重新分配给瓶颈阶段.

策略 3. 改变负载总量. 当内存资源成为瓶颈, 减少训练的微批次大小(micro-batch size)或在瓶颈阶段启动“重计算”机制(以计算换内存, 减少微批次大小和“重计算”都会增加计算时间, 但降低内存用量); 当计算资源成为瓶颈, 增加训练的微批次大小或在瓶颈阶段关闭“重计算”机制(以内存换计算).

改变并行方式和负载总量后, 流水线的不同阶段之间仍然可能存在算力/负载不均衡的现象, 因此我们选择将策略 1~3 结合起来使用: 每次应用策略 2 或策略 3 后, 我们都会应用 1 次负载均衡策略, 如图 8 所示, 寻找在新的算力分布、负载总量下最优的流水线切分策略.

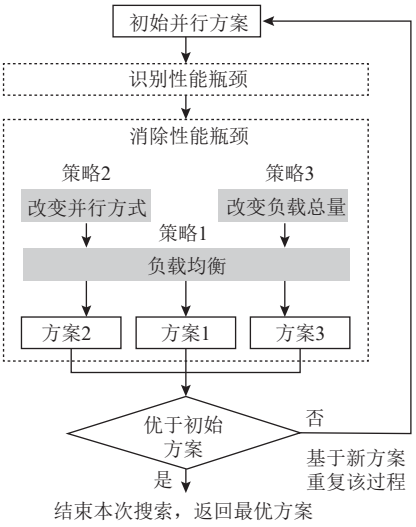


Fig. 8 Workflow of bottleneck alleviation algorithm

图 8 “瓶颈消除”算法流程

在搜索过程中, 我们需要评估每一个并行方案在给定硬件环境上的运行时间与内存占用, 从而评估哪一类资源造成了性能瓶颈. 我们使用 Aceso 提供的基于测量的性能模型提前测量每个算子在不同的切分方案下的内存占用和执行时间, 在搜索瓶颈消除策略时直接复用测量结果, 即可得到整个模型的资源用量情况.

5.3 如何选择方案: 性能与重构开销的权衡

不同的瓶颈消除策略将会带来多种不同的并行方案. 我们往往难以提前知道哪一种策略对当前的瓶颈阶段效果最好, 因此我们会尝试每一种策略并

从这些策略产生的所有并行方案中选择最好的方案. 已有的自动并行工作往往会选择性能最优的方案, 因为这些工作都是离线搜索, 以并行方案的性能为搜索目标, 不考虑从旧的并行方案到新的并行方案的调整时间, 但这一开销在故障恢复的场景尤其重要. 例如, 当集群故障频率较高或者弹性实例回收频率较高时, 如果故障恢复开销过高, 则下一次故障发生前的实际可用于训练的时间将会减少, 可能导致训练进度变慢, 此时使用一个性能不是最优但恢复开销低的方案可能反而会带来更多的收益.

为了实现弹性实例场景下的并行方案在线调整, AdaptDNN 在选择方案时会综合考虑并行方案的性能与故障恢复开销. 我们提出了一个新的指标 *iters* (下一次故障前有效的轮次数)作为选择并行方案的依据, 其计算方式为

$$iters = \frac{MTTF - T_{\text{AdaptDNN}}}{T_{\text{iteration}}}, \tag{5}$$

其中 $MTTF$ (mean time to failure)是平均故障时间, 基于历史数据计算得到. T_{AdaptDNN} 是故障恢复开销, 由式(4)计算得到; $T_{\text{iteration}}$ 是新方案的每轮次训练时间, 由性能模型得到.

由此, AdaptDNN 在调整并行策略时总会朝着下一次故障发生前有效的轮次数最大的方向搜索并行方案, 从而实现在任何故障频率下, 系统都能按照对训练进度最有利的方向进行快速调整.

5.4 算法实现

优化某个瓶颈阶段有可能造成其他阶段的性能下降, 甚至劣化并行方案的整体性能, 因此 AdaptDNN 采用了一个迭代式的算法, 即当一次瓶颈消除并未找到优于初始方案的并行方案时, 会重复执行“发现瓶颈-消除瓶颈”的策略, 直至达到给定的最大搜索深度.

AdaptDNN 使用“瓶颈消除”思想调整并行方案的流程如图 8 所示, 该算法的伪代码实现如下:

算法 2. 瓶颈消除算法 *BottleResolve*.

输入: 初始并行方案 *InitConfig*, 初始有效轮次数 *InitIters*, 最大搜索深度 *MaxHops*;

输出: 如果找到有效轮次数更多的方案, 则输出新的方案; 否则, 输出 None.

① if *MaxHops* > 0 then
② *BottleStage* ← *IdentifyBottlenck*(*InitConfig*); /*利用性能模型得到当前的瓶颈阶段*/
③ C_1 ← *ReBalance*(*InitConfig*); /*利用策略 1(负载均衡)得到新方案 C_1*/
④ C_2 ← *ReParallelize*(*InitConfig*, *BottleStage*), C_2 ← *ReBalance*(C_2); /*利用策略 2(改变并行方式+负载均衡)得到新方案 C_2*/
⑤ C_3 ← *ReWorkloads*(*InitConfig*, *BottleStage*), C_3 ← *ReBalance* (C_3); /*利用策略 3(改变负载总量+负载均衡)得到新方案 C_3*/
⑥ *new_configs* ← [C_1, C_2, C_3];
⑦ *new_configs*. *sort*(); /*将新方案按有效轮次数从高到低排序*/
⑧ if *new_configs*[0]. *iters* > *InitIters*
⑨ return *new_configs*[0];
⑩ end if
⑪ for *new_config* in *new_configs* do
⑫ *result* = *BottleResolve*(*new_config*, *InitIters*, *MaxHops*−1); /*重复该过程, 直至达到最大搜索深度*/
⑬ if *result* is not None then
⑭ return *result*;
⑮ end if
⑯ end for
⑰ end if
⑱ return None.

使用算法 2 消除性能瓶颈后, 新的并行方案中可能会存在新的性能瓶颈, 我们可以基于新方案再次调用算法 2, 从而持续性改进并行方案的性能. 在弹性实例场景下, AdaptDNN 会在 CPU 端持续性重复调用算法 2, 直至无法找到更好的方案或者发生实例回收. 实例回收时, AdaptDNN 会使用目前已找到的最优方案进行故障恢复.

6 实验与结果

6.1 实验设置

1)实验环境. 我们的实验集群包含 16 张 V100 (16 GB)GPU, 由 4 台 GPU 服务器组成, 其中每台服务器配备 4 张 GPU. 节点内部 GPU 通过 PCIE 互联, 节点间通过 100 Gbps IB 网卡[41] 互联. 实验平台具体配置如表 1 所示.

2)实验模型. 我们选用了目前被广泛应用的 GPT-3 模型, 并测试了 GPT-3 不同的规模, 包括 350×10^6, 1.3×10^9, 2.7×10^9, 6.7×10^9.

3)参数设置. 本文主要关注模型训练的性能(包括每轮次训练时间、训练吞吐量等指标), 不考虑将模型训练到某一准确率的总时间. 影响模型训练性能的实验参数是模型采用的精度、数据批次大小

Table 1 Experimental Platform Configuration

表 1 实验平台配置

配置	参数
处理器	Intel® Xeon® Gold 5218R CPU
GPU	NVIDIA V100-PCIE (16 GB)
操作系统	Ubuntu 18.04.5 LTS
CUDA 版本	11.6
PyTorch 版本	2.0.0

(batch size)和序列长度(sequence length),本文后续实验所选用的参数如表 2 所示. 对于其他不影响训练性能的参数(例如学习率等),我们按照 Megatron-LM 的默认配置进行设置.

Table 2 DNN Model Parameters

表 2 DNN 模型参数

配置	参数
参数量	350×10^6, 1.3×10^9, 2.7×10^9, 6.7×10^9
精度	FP16
数据批次大小	1 024
序列长度	2 048

4)基准系统. 我们分别与基于 Checkpoint 的容错机制和基于冗余性的容错机制进行了对比. 其中,我们采用了 Megatron-LM 实现的 Checkpoint 功能来测量模型保存和读取 Checkpoint 的时间. 因为 Checkpoint 机制与模型并行方案无关,为了消除 AdaptDNN 和 Megatron-LM 在模型训练上的性能差异,我们使用 AdaptDNN 的每轮次训练时间作为 Checkpoint 机制的模型训练时间. 对于基于冗余性的容错机制,我们选用了目前最先进的 Oobleck 系统进行对比.

5)对比方法. 我们将通过模拟实例回收,对比不同系统在同样的故障模式下的训练吞吐量以及故障恢复开销(6.2~6.3 节). 进一步地,我们将使用实测的数据模拟一个弹性实例集群在一段时间内的平均训练吞吐量,用于对比各个系统在真实场景下的性能与容错能力(6.4 节). 最后,我们将通过消融实验对比本文提出的并行方案在线调整算法与传统离线搜索算法在弹性实例场景下的性能(6.5 节).

6.2 弹性实例场景下的系统吞吐量对比

我们测试了不同的系统在弹性实例场景下的训练吞吐量. 我们通过主动进行 GPU 回收,来模拟真实场景下实例数量的弹性变化.

1)实验设计. 我们从 16 个 GPU 开始训练模型,每隔 2 个训练轮次后,在下一个轮次进行了一半时,结束其中一个 GPU 上的训练进程. 重复此过程直至仅剩 8 个 GPU 为止. 我们测量了不同系统在不同 GPU 数量下的每轮次训练时间,并测量从发现节点回收开始,到继续训练之间的故障恢复时间. 对于 Checkpoint 机制,我们每隔一个轮次保存一次模型状态.

2)对比指标. 我们将对比不同系统在实验过程中的总体训练吞吐量:

$$\text{总体训练吞吐量}=\frac{\text{自训练开始完成的样本数量}}{\text{总共消耗的时间}}.$$

其中,总共消耗的时间包含了正常训练时间与故障恢复时间.

3)实验结果. 图 9 显示了实验过程中每次发生节点回收后系统可用的 GPU 数量与该时刻的总体训练吞吐量. 从中我们可以发现,AdaptDNN 在所有场景下的总体吞吐量都优于基准系统. 与 Checkpoint 相比,AdaptDNN 最高实现了 1.87 倍性能提升(GPT-3 6.7B),与 Oobleck 相比,AdaptDNN 最高实现了 1.44 倍性能提升(GPT-3 1.3B). 并且,AdaptDNN 实现了使用相同数量的 GPU 训练比 Oobleck 更大的模型:Oobleck 在 GPT-3 2.7B 模型的 GPU 数量下降到 10 卡以下便无法部署,GPT-3 6.7B 模型更是完全无法部署. 而 AdaptDNN 可以支持 GPT-3 6.7B 模型,并且最少支持 14 卡进行部署.

6.3 系统性能分解对比

我们进一步对 6.2 节中的实验进行性能分解,从而更加详细地对比不同系统的性能差异.

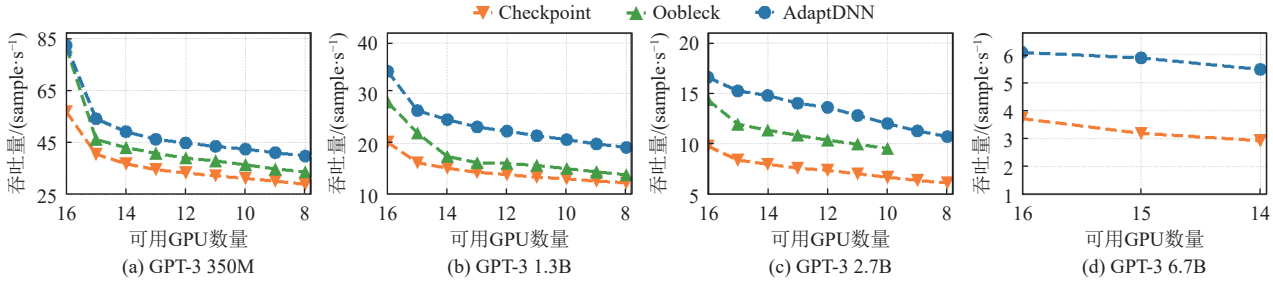
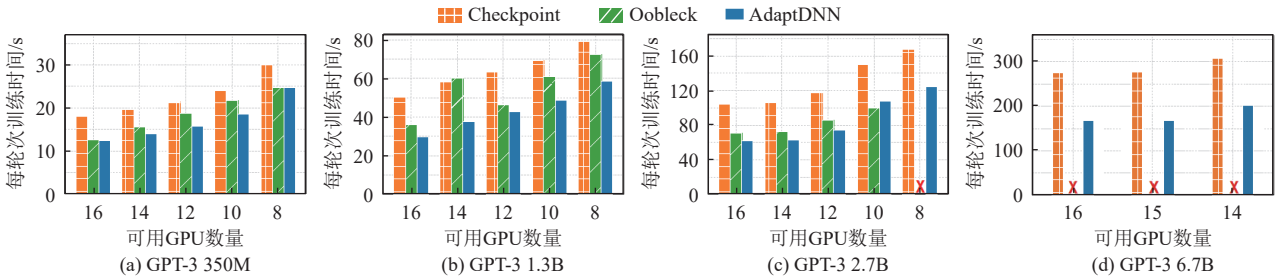


Fig. 9 Overall throughput comparison of training system

图 9 训练系统总体吞吐量对比

1）对比指标. 我们主要对比可训练的模型大小下不同系统在不同数量的 GPU 下的每轮次训练时间以及实例被回收时的故障恢复时间.

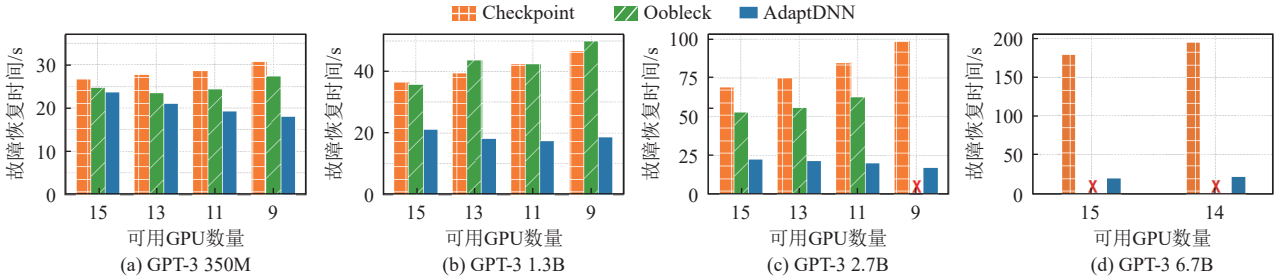
2）实验结果. 实验结果如图 10 和图 11 所示.



注：“X”表示无法执行.

Fig. 10 Comparison of training time per round

图 10 每轮次训练时间对比



注：“X”表示无法执行.

Fig. 11 Comparison of time for fault recovery

图 11 故障恢复时间对比

从图中我们可以得出以下结论：

1）AdaptDNN 的训练时间更短. 如图 10 所示，AdaptDNN 在绝大多数场景下比 Oobleck 和 Checkpoint 的每轮次训练时间更短，最好性能是基于 Checkpoint 方式的 1.7 倍，是 Oobleck 的 1.6 倍. 基于 Checkpoint 方式由于读写磁盘的带宽较低，因此会对训练时间造成一定程度的延长，而 Oobleck 由于对并行方案有一定的要求（必须使用数据并行）因此并未采用最优的并行方案，导致性能不是最优.

2）AdaptDNN 能支持更大的模型. 如图 10 所示，在现有 16 张 V100GPU 卡的实验环境下，AdaptDNN 可以支持 GPT-3 6.7B，而 Oobleck 最高支持 GPT-3 2.7B；对于 GPT-3 2.7B，Oobleck 最少需要 10 个 GPU，而 AdaptDNN 可以用 8 个 GPU 完成训练. 因为 Oobleck 需要使用数据并行，导致内存压力较大，而 AdaptDNN 对并行方案无要求，因此可以训练更大的模型.

3）AdaptDNN 的故障恢复时间更小. 如图 11 所示，我们比较了不同系统在 GPU 数量发生变化时的故障恢复开销，AdaptDNN 的开销最低仅为 Oobleck 的 25%，为 Checkpoint 的 11%. 因为 Oobleck 将丢失当前轮次的部分训练进度，在完成模型的重新部署后需要重新计算丢失的进度，而基于 Checkpoint 的容错机制从磁盘读取 Checkpoint 的带宽低、开销大，并且也需要重新计算丢失的进度，导致故障恢复的开销极高.

6.4 基于真实 trace 的模拟实验

在 6.2 节中，我们测试了逐渐减少集群中 GPU 数量时，不同系统的训练吞吐量. 在本节中，进一步测试 AdaptDNN 和各基准系统在真实弹性实例集群中的性能.

1）实验 trace. 使用了 Bamboo 开源的 2 段真实 trace[7] 进行模拟实验. 这 2 段 trace 分别记录了 AWS 的 2 个不同型号的 32 卡弹性集群在 12 h 内的可用实例数量. 由于本实验所用的集群只有 16 卡，我们对该 trace 进行了按比例缩放，使得实例数量的变化在 8~16.

2）实验设计. 在 2 段不同的 trace 上分别模拟了 GPT-3 1.3B 和 GPT-3 2.7B 模型的训练过程. 我们测量了每个系统在每一次实例数量发生变化前后的训练吞吐量，以及对并行方案调整的开销，然后基于这些实测性能数据，利用 Bamboo 提供的模拟器，模拟了 12 h 内系统的训练吞吐量变化情况.

3）对比指标. 将对比不同系统在 12 h 内的平均

训练吞吐量(sample/s)以及每个时刻的吞吐量. 每时刻的吞吐量通过计算一个较小时间窗口内的平均吞吐量得到.

4)实验结果. 各系统的吞吐量随着实例数量变化的趋势如图 12 所示, AdaptDNN 在绝大多数时刻都展现出了更好的训练吞吐量. 在实例数量发生变化时, AdaptDNN 吞吐量的下降幅度相比基准系统更小, 体现出了对故障更强的容忍能力和快速的恢复能力.

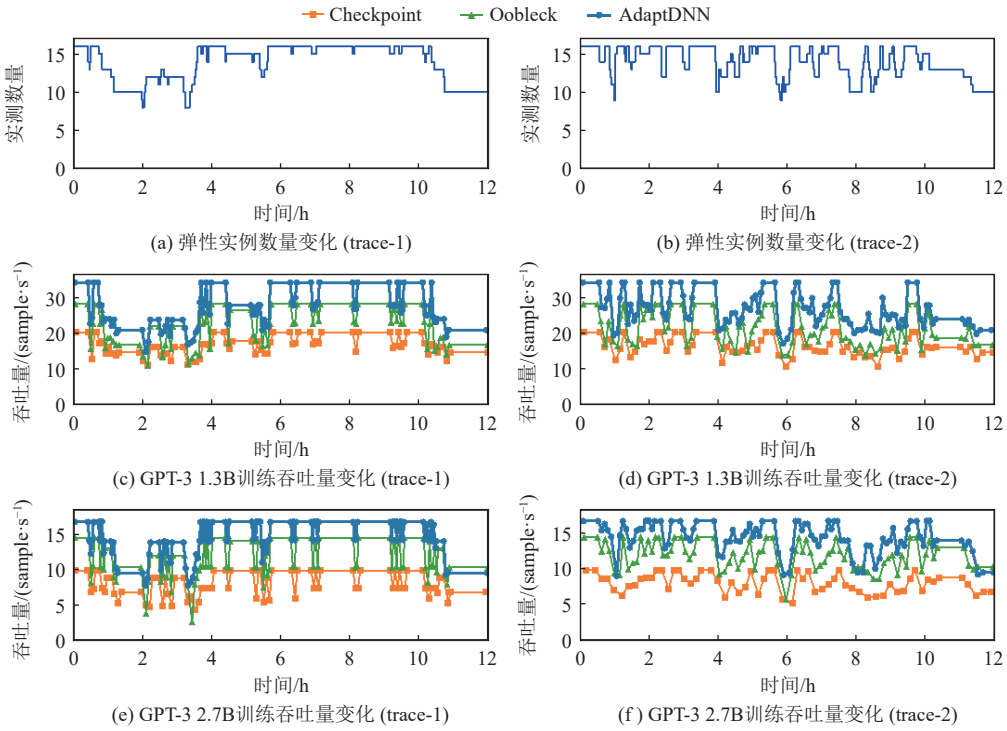


Fig. 12 Comparison of training throughput under real spot instance trace

图 12 真实弹性实例 trace 下的训练吞吐量对比

各系统 12 h 内的平均吞吐量如表 3 所示. AdaptDNN 在所有测试场景下都表现出了相比基准系统更好的性能. 其中, AdaptDNN 相比 Checkpoint 方式, 最高达到了 1.79 倍的训练吞吐量(GPT-3 2.7B, trace-2), 相比 Oobleck 最高达到了 1.27 倍训练吞吐量(GPT-3 1.3B, trace-2). 我们发现, AdaptDNN 在 trace-2 相比基准系统的性能提升更大, 主要原因是 trace-2 的实例回收更加频繁, 基准系统的恢复开销较高.

Table 3 Average Throughput Comparison

表 3 平均吞吐量对比

模型	GPT-3 1.3B		GPT-3 2.7B	
	trace-1	trace-2	trace-1	trace-2
Checkpoint	17.13	16.38	8.21	7.90
Oobleck	22.83	21.15	12.07	11.79
AdaptDNN (本文)	27.89	26.92	14.22	14.15

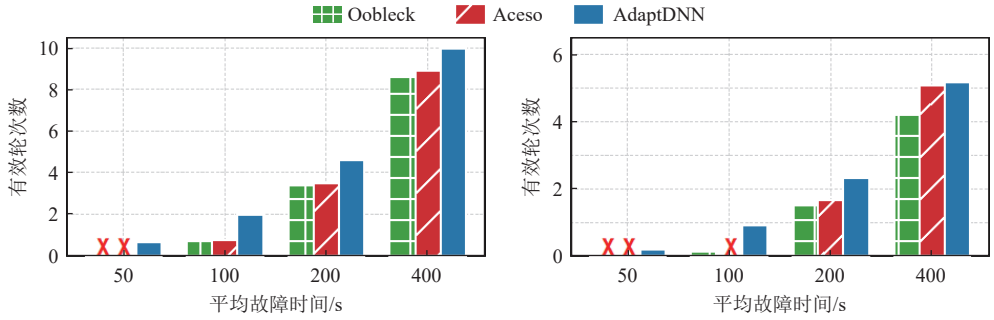
6.5 消融实验

AdaptDNN 在搜索模型的调整方案时引入了“下一次故障前有效的轮次数”指标作为优化的目标函数, 而该指标与平均故障时间(mean time to failure, *MTTF*)和故障恢复开销有关. 进一步地, 故障恢复开销又与节点间网络通信带宽和宽限期大小相关. 因此, 在本节中, 我们进行了一系列消融实验, 分别测试 *MTTF*、网络带宽和宽限期大小对故障恢复的影响.

1)基准系统. 我们与 Oobleck 和 Aceso 进行了对比, Oobleck 和 Aceso 都是基于最优性能搜索模型的调整方案. Checkpoint 机制可以与任意并行方案组合使用, 其本身不包含对调整方案的搜索, 因此不作为本次消融实验的基准系统.

2)对比指标. 比较不同系统在完成故障恢复后的下一次故障前有效的轮次数(*iters*).

实验 1. *MTTF* 对有效轮次数的影响. 我们保持宽限期为 30 s 和网络带宽为 1 Gbps. 测试了 4 种不同的 *MTTF*: 50 s, 100 s, 200 s, 400 s. 结果如图 13 所示, 当 *MTTF* 较小时(例如 *MTTF*=50 s), 即故障频繁发生, 此时仅以性能为搜索目标的 Oobleck 和 Aceso 无法找到可行的重构方案, 或找到的方案只能产生较小有效轮次. 而 AdaptDNN 由于将 *MTTF* 引入搜索的目标函



(a) GPT-3 1.3B, GPU数量从16变为15　(b) GPT-3 2.7B, GPU数量从16变为15

注：“X”表示无法完成故障恢复.

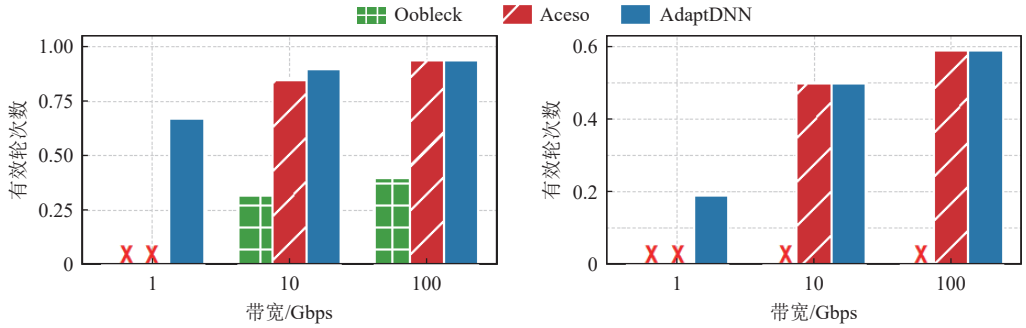
Fig. 13　Influence of *MTTF* on *iters*

图 13　平均故障时间对有效轮次数的影响

数，因此在各种 *MTTF* 下都能找到故障恢复的可行方案. 当 *MTTF* 较大时，则故障发生并不频繁，各系统的性能接近.

实验 2. 网络带宽对有效轮次数的影响. 我们保持宽限期为 30 s 和 *MTTF* 为 50 s，测试了 3 种不同的网络带宽：1 Gbps, 10 Gbps, 100 Gbps. 结果如图 14 所示，当网络带宽较低时，Oobleck 和 Aceso 找到的调整方案无法在宽限期内完成，而 AdaptDNN 总能找到可行的重构方案. 当模型规模增加，模型参数量增多，故障恢复开销也随之上升，由于 Oobleck 搜索时不考虑故障恢复开销，因此在 *MTTF* 为 50 s 且模型较大时（GPT-3 2.7B），在 3 种带宽下都没有找到可行的调整方案.

实验 3. 宽限期对有效轮次数的影响. 我们保持 *MTTF* 为 50 s 和带宽为 1 Gbps，测试了 3 种不同的宽限期：30 s, 60 s, 120 s. 如图 15 所示，当宽限期较小时，Oobleck 和 Aceso 无法找到可行解. 另外，由于此时的

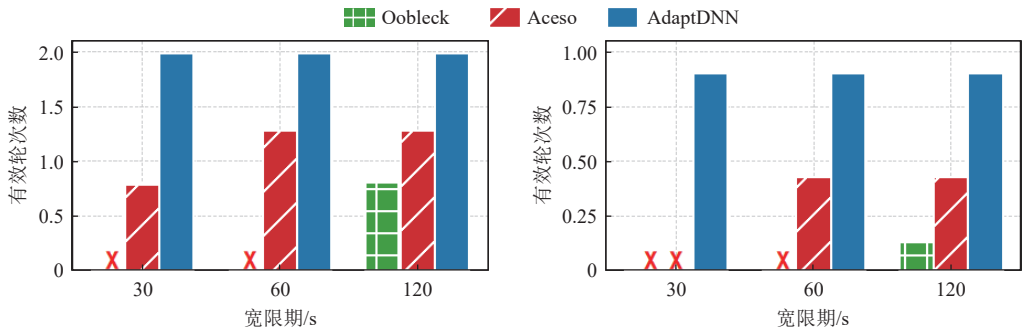


(a) GPT-3 1.3B, GPU数量从16变为15　(b) GPT-3 2.7B, GPU数量从16变为15

注：“X”表示无法完成故障恢复.

Fig. 14　Influence of bandwidth on *iters*

图 14　网络带宽对有效轮次数的影响



(a) GPT-3 1.3B, GPU数量从16变为15　(b) GPT-3 2.7B, GPU数量从16变为15

注：“X”表示无法完成故障恢复.

Fig. 15　Influence of grace period on *iters*

图 15　宽限期对有效轮次数的影响

带宽和 *MTTF* 都比较小,导致故障恢复的时间较长,因此综合考虑故障恢复开销和并行方案性能的 AdaptDNN 能在所有测试下找到最优的调整方案,从而实现了更大的有效轮次数.

7 结 论

本文对弹性实例下的深度学习分布式训练进行了深入研究,提出了在实例回收前利用宽限期快速完成故障恢复,并利用消除瓶颈的思想快速搜索模型的调整方案,从而实现了模型训练的自适应可伸缩性.实验表明我们的系统能更好地适应弹性实例场景,从而为大模型的低成本训练提供了可行且高效的解决方案.

作者贡献声明: 刘国栋提出本文的主要思路,并完成了部分实验和撰写论文;朱家祺和高梓源完成了部分实验和撰写论文;包云岗和王卅负责对论文整体思路进行指导并提出了修改意见.

参 考 文 献

[1] OpenAI. AI and compute[EB/OL]. (2018-05-16)[2024-05-30]. https://openai.com/index/ai-and-compute/

[2] Li Mu, Andersen D, Park J, et al. Scaling distributed machine learning with the parameter server[C]//Proc of the 11th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2014: 583−598

[3] Jia Zhihao, Zaharia M, Aiken A. Beyond data and model parallelism for deep neural networks[C/OL]//Proc of the 2nd Machine Learning and Systems Conf. 2019[2024-10-21]. https://proceedings.mlsys.org/paper_files/paper/2019/file/b422680f3db0986ddd7f8f126baaf0fa-Paper.pdf

[4] Narayanan D, Harlap A, Phanishayee A, et al. PipeDream: Generalized pipeline parallelism for DNN training[C]//Proc of the 27th ACM Symp on Operating Systems Principles. New York: ACM, 2019: 1−15

[5] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[C/OL]//Proc of the 34th Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2020[2024-10-21]. https://papers.nips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[6] Duan Jiangfei, Song Ziang, Miao Xupeng, et al. Parcae: Proactive, liveput-optimized DNN training on preemptible instances[C]//Proc of the 21st USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2024: 1121−1139

[7] Thorpe J, Zhao Pengzhan, Eyolfson J, et al. Bamboo: Making preemptible instances resilient for affordable training of large DNNs[C]//Proc of the 20th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2023: 497−513

[8] Li Jiamin, Xu Hong, Zhu Yibo, et al. Lyra: Elastic scheduling for deep learning clusters[C]//Proc of the 18th European Conf on Computer Systems. New York: ACM, 2023: 835−850

[9] Narayanan D, Shoeybi M, Casper J, et al. Efficient large-scale language model training on GPU clusters using megatron-LM[C]//Proc of the 34th Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2021: 1−15

[10] Mohan J, Phanishayee A, Chidambaram V. CheckFreq: Frequent, fine-grained DNN checkpointing[C]//Proc of the 19th USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2021: 203−216

[11] Athlur S, Saran N, Sivathanu M, et al. Varuna: Scalable, low-cost training of massive deep learning models[C]//Proc of the 17th European Conf on Computer Systems. New York: ACM, 2022: 472−487

[12] Jang I, Yang Zhenning, Zhang Zhen, et al. Oobleck: Resilient distributed training of large models using pipeline templates[C]//Proc of the 29th Symp on Operating Systems Principles. New York: ACM, 2023: 382−395

[13] Zheng Lianmin, Li Zhuohan, Zhang Hao, et al. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning[C]//Proc of the 16th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2022: 559−578

[14] Liu Guodong, Miao Youshan, Lin Zhiqi, et al. Aceso: Efficient parallel DNN training through iterative bottleneck alleviation[C]//Proc of the 19th European Conf on Computer Systems. New York: ACM, 2024: 163−181

[15] Dean J, Corrado G, Monga R, et al. Large scale distributed deep networks[C/OL]//Proc of the 26th Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2012[2024-10-21]. https://papers.nips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[16] Wang Minjie, Huang C, Li Jinyang. Supporting very large models using automatic dataflow graph partitioning[C]//Proc of the 14th European Conf on Computer Systems. New York: ACM, 2019: 1−17

[17] Mirhoseini A, Pham H, Le Q, et al. Device placement optimization with reinforcement learning[C]//Proc of the 34th Int Conf on Machine Learning. New York: PMLR, 2017: 2430-−2439

[18] Gao Yuanxiang, Chen Li, Li Baochun. Spotlight: Optimizing device placement for training deep neural networks[C]//Proc of the 35th Int Conf on Machine Learning. New York: PMLR, 2018: 1676−1684

[19] Xu Yuanzhong, HyoukJoong L, Chen Dehao, et al. GSPMD: General and scalable parallelization for ML computation graphs[J]. arXiv preprint, arXiv: 2105.04663

[20] Huang Yanping, Cheng Youlong, Bapna A, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism[C/OL]//Proc of the 33rd Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2019[2024-10-21]. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[21] Narayanan D, Phanishayee A, Shi K, et al. Memory-efficient pipeline-parallel DNN training[C]//Proc of the 38th Int Conf on Machine

Learning. New York: PMLR, 2021: 7937−7947

[22] Li Shigang, Torsten H. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines[C]//Proc of the 34th Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2021: 1−14

[23] Fan Shiqing, Rong Yi, Meng Chen, et al. DAPPLE: A pipelined data parallel approach for training large models[C]//Proc of the 26th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2021: 431−445

[24] PyTorch. Torch Distributed Elastic[EB/OL]. [2024-05-30]. https://pytorch.org/docs/stable/distributed.elastic.html

[25] The Horovod Authors. Elastic Hovorod[EB/OL]. [2024-05-30]. https://horovod.readthedocs.io/en/stable/elastic_include.html

[26] Wu Yidi, Ma Kaihao, Yan Xiao, et al. Elastic deep learning in multi-tenant GPU clusters[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 33(1): 144−158

[27] Xiao Wencong, Bhardwaj R, Ramjee R, et al. Gandiva: Introspective cluster scheduling for deep learning[C]//Proc of the 13th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2018: 595−610

[28] Qiao A, Choe S, Subramanya S, et al. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning[C]//Proc of the 15th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2021: 1−18

[29] Gu Diandian, Zhao Yihao, Zhong Yinmin, et al. ElasticFlow: An elastic serverless training platform for distributed deep learning[C]//Proc of the 28th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2023: 266−280

[30] Wang Zhuang, Jia Zhen, Zheng Shuai, et al. Gemini: Fast failure recovery in distributed training with in-memory checkpoints[C]//Proc of the 29th Symp on Operating Systems Principles. New York: ACM, 2023: 364−381

[31] He Tao, Li Xue, Wang Zhibin, et al. Unicron: Economizing self-healing LLM training at scale[J]. arXiv preprint, arXiv: 2401.00134, 2023

[32] Jiang Ziheng, Lin Haibin, Zhong Yinmin, et al. MegaScale: Scaling large language model training to more than 10 000 GPUs[C]//Proc of the 21st USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2024: 745−760

[33] Wang Yuxin, Kang Xueze, Shi Shaohuai, et al. Reliable and efficient in-memory fault tolerance of large language model pretraining[EB/OL]. (2024-08-19)[2024-09-06]. https://arxiv.org/abs/2310.12670

[34] Lian Xinyu, Jacobs S, Kurilenko L, et al. Universal checkpointing: Efficient and flexible checkpointing for large scale distributed training[J]. arXiv preprint, arXiv: 2406.18820, 2024

[35] Wan Borui, Han Mingji, Sheng Yiyao, et al. ByteCheckpoint: A unified checkpointing system for LLM development[J]. arXiv preprint, arXiv: 2407.20143, 2024

[36] Gandhi S, Zhao M, Skiadopoulos A, et al. SlipStream: Adapting pipelines for distributed training of large DNNs amid failures[J]. arXiv preprint, arXiv: 2405.14009, 2024

[37] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C/OL]//Proc of the 31st Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2017[2024-10-21]. https://user.phil.hhu.de/~cwurm/wp-content/uploads/2020/01/7181-attention-is-all-you-need.pdf

[38] Navarro A, Asenjo R, Tabik S, et al. Analytical modeling of pipeline parallelism[C]//Proc of the 18th Int Conf on Parallel Architectures and Compilation Techniques. Piscataway, NJ: IEEE, 2009: 291−290

[39] Perez T, Chen Wei, Ji R, et al. PETS: Bottleneck-aware spark tuning with parameter ensembles[C]//Proc of the 27th Int Conf on Computer Communication and Networks. Piscataway, NJ: IEEE, 2018: 1−9

[40] Kahveci B, Gedik B. Joker: Elastic stream processing with organic adaptation[J]. Journal of Parallel and Distributed Computing, 2020, 137: 205-223

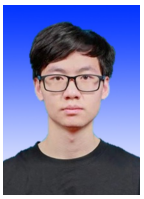
[41] NVIDIA. ConnectX-5: Advanced offload capabilities for the most demanding applications[EB/OL]. [2024-05-30]. https://www.nvidia.com/ en-us/networking/ethernet/connectx-5/

Liu Guodong, born in 1997. PhD candidate. His main research interests include distributed systems and machine learning systems.

刘国栋, 1997 年生. 博士研究生. 主要研究方向为分布式系统、机器学习系统.

Zhu Jiaqi, born in 1999. Master candidate. His main research interests include operating systems and machine learning systems. (zhujiaqi22s@ict.ac.cn)

朱家祺, 1999 年生. 硕士研究生. 主要研究方向为操作系统、机器学习系统.



Gao Ziyuan, born in 2001. Master candidate. His main research interests include machine learning system, virtualization, and cloud computing. (gaoziyuan23s@ict.ac.cn)

高梓源, 2001 年生. 硕士研究生. 主要研究方向为机器学习系统、虚拟化、云计算.

Bao Yungang, born in 1980. PhD, professor. Member of CCF. His main research interests include data center architecture, agile design methodology of processor chips, and ecosystem of open-source processor chips. (baoyg@ict.ac.cn)

包云岗, 1980 年生. 博士, 研究员. CCF 会员. 主要研究方向为数据中心体系结构、处理器芯片敏捷设计方法论、开源芯片生态.

Wang Sa, born in 1986. PhD, associate professor. Member of CCF. His main research interests include cloud computing, operating systems, and system modeling and performance analysis.

王　卅, 1986 年生. 博士, 副研究员. CCF 会员. 主要研究方向为云计算、操作系统、系统建模与性能分析.