S W IFT：大规模深度神经网络训练的快速故障恢复

Yuchen Zhong、Guang ming Sheng、Juncheng Liu、Jinhui Yuan 和 Chuan Wu，*IEEE* 高级会员

摘要——随着深度学习模型规模不断增大，训练所需的时间和资源也越来越多，因此容错变得愈发重要。现有最先进方法（如 CheckFreq 和 Elastic Horovod）需要在内存中备份一份模型状态（即参数和优化器状态），这对于大型模型代价高昂，并会带来不可忽视的开销。本文提出 SWIFT，这是一种面向分布式深度神经网络训练的新型恢复设计，可在不影响训练吞吐量和模型准确率的前提下，显著降低故障恢复开销。SWIFT 不会额外复制模型状态，而是解决故障导致的模型状态不一致问题，并利用数据并行中的模型状态副本进行故障恢复。当副本不可用时，我们提出一种基于日志的方法，记录中间数据，并在故障发生时重放计算以恢复丢失状态。我们将重新计算分布到多台机器上，进一步加快故障恢复。此外，我们还选择性地记录中间数据，以权衡恢复时间与中间数据的存储开销。评估结果表明，与最先进方法相比，SWIFT 显著缩短了故障恢复时间，并且在无故障执行期间实现了相当或更高的训练吞吐量，同时不会降低最终模型准确率。与最先进方法相比，SWIFT 还可使总训练时间最高加速 1.16 倍。

索引词—分布式DNN训练；故障韧性

arXiv:2302.06173v1 [cs.DC] 13 Feb 2023

1 引言

近ARGER 年来，为提升模型性能，越来越大型的深度神经网络（DNN）相继出现 [1],[2], [3]。大型深度神经网络模型训练作业通常使用大量加速器（例如 GPU），且运行时间很长 [4]。例如，在 1024 个 A100 GPU 上训练 GPT-3 模型 [3]，预计需要一个多月的时间 [5]。

在 GPU训练集群中，作业发生故障十分常见 [4]。例如，机器崩溃和网络故障偶尔会发生，或者更高优先级的作业会占用资源 [4]。在这些情况下，分布式DNN训练作业会被中断，导致深度神经网络模型状态（即模型参数和优化器状态）丢失，训练作业也随之失败。对于大型深度神经网络模型训练作业而言，故障的影响更加严重：机器数量增加必然会提高发生故障的概率；大型模型的训练需要数天至数月，因此训练过程中发生故障的可能性更高。近期的研究工作也印证了这一点 [6], [7]。

全局检查点是深度学习（DL）框架中实现容错的事实标准方法 [8], [9]。训练作业会定期保存整个模型状态的检查点。作业发生故障时，所有工作节点都会从最新的检查点重新启动。根据检查点保存的频率，这通常会导致数小时的计算时间损失 [10]。CheckFreq [10] 通过将操作分为两个阶段，实现更频繁的检查点保存：首先，将模型状态复制到 GPU 内存中，称为快照；如果 GPU 内存不足，则复制到 CPU 内存中；第二阶段则异步将快照写入磁盘。

• *YuchenZhong*、*Guangming Sheng* 和 *Chuan Wu* 就职于中国香港的香港大学。电子邮箱：*{yczhong, gmsheng, cwu}@cs.hku.hk*。

• *Juncheng Liu* 和 *Jinhui Yuan* 就职于中国北京的 *OneFlow Inc.*。电子邮箱：*{liujuncheng, yuanjinhui}@oneflow.org*。

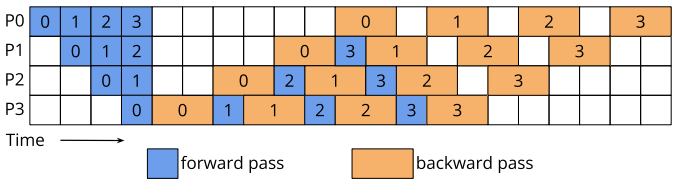
Elastic Horovod [11], 是一种弹性训练框架，采用了类似的方法，但没有第二阶段。这是因为 Elastic Horovod 假设采用分布式数据并行训练，其中每个工作节点都保存一份模型状态副本；在故障恢复期间，一个存活工作节点会将快照广播给其他工作节点，所有工作节点再从快照重新开始训练。创建快照对于 Elastic Horovod 防止状态损坏是必要的：如果某个工作节点在参数更新期间崩溃，其他工作节点就会陷入尴尬境地——部分参数已更新，而其他参数尚未更新。我们将这一问题称为崩溃一致性问题（§2.3）。然而，正如我们将在 §2.2 中所见，对于大型深度神经网络模型，这两种方法都会因快照创建的开销而减慢训练速度。

本文研究了一种更优的分布式深度神经网络训练故障韧性设计，在不影响训练吞吐量和最终模型准确率的情况下，显著降低恢复开销。我们的一个关键观察是，深度神经网络训练中用于模型状态更新的许多优化器在数学上都是可逆的。例如，随机梯度下降（SGD）仅涉及逐元素加法和标量乘法等线性算子，其逆算子十分简单。在分布式数据并行训练中发生崩溃一致性问题时，我们可以通过撤销更新 已更新的参数（§4），将存活工作节点的模型状态恢复到一致状态。因此，我们无需像 CheckFreq 和 Elastic Horovod 那样定期创建快照，从而将无故障执行期间的开销降至零（定期检查点除外）。由于该方法利用存活工作节点中的模型状态副本进行恢复，我们将这种恢复方法命名为基于复制的恢复。

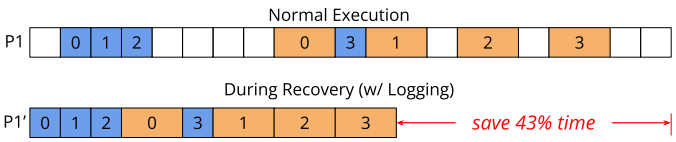
然而，即使采用数据并行，也并非总能获得副本。例如，一些先前的研究主张

仅在同一台机器上的多个 GPU 之间进行数据并行，以利用 NVLink [5], [12] 等高速服务器内互连，加速梯度同步。一旦发生机器故障，所有副本都会丢失。

随后，我们研究分布式系统中另一种基础容错方法——日志记录，这种方法已在数据处理系统中得到广泛研究 [13], [14], [15], [16]。我们为流水线并行训练引入基于日志的恢复（§5）。在流水线并行中，工作节点构成链式拓扑，并通过点对点通信，将中间激活值或梯度传递给后继或前驱工作节点。图 1a 展示了单前向单反向（1F1B）流水线调度 [17]（§2.1）。通过日志记录，每个工作节点都会在本地记录发往另一台机器上相邻工作节点的所有数据。发生故障后，故障工作节点的替代节点会获取日志数据，并重放计算以恢复丢失状态。此外，我们还将日志数据分发给存活工作节点，让它们协助恢复（§5.2）。日志记录将恢复范围从所有工作节点构成的完整计算图缩小到故障工作节点上的计算图，因此，与全局检查点保存相比，可以缩短恢复时间。图 1b 给出了一个示例。据我们所知，我们是首个将日志记录引入分布式深度学习系统以增强故障韧性的团队。



(a) 采用 1F1B 调度的四设备（P0, ..., P3）流水线并行。方块中的数字表示正在处理的微批次。白色方块表示气泡时间。



(b) 一个示例展示了日志记录如何加快故障后的恢复：其中 (a) 中的工作节点P1发生故障，并由新的工作节点P1'替代。在正常执行期间，P1 需要等待来自其他工作节点的中间激活值或梯度。在恢复 P1 的丢失状态时，P1' 可以读取已记录的数据（来自其他工作节点的激活值和梯度），并直接进行重新计算（无需像正常执行时那样经历气泡时间），从而实现比检查点恢复大幅缩短的恢复时间。

图 1： 流水线并行与日志记录示例。

然而，基于日志的恢复带来了独特的挑战。在深度神经网络训练期间，需要持续进行日志记录，其运行时开销和空间开销可能令人难以承受。一旦某部分已记录的数据丢失，就无法精确恢复原始状态。为了降低日志记录的运行时开销，我们只记录机器间通信数据，因为故障通常发生在机器上，而非机器上的单个工作节点。此外，我们异步执行日志记录，在后台存储数据。我们还利用工作节点的空闲时间（即流水线并行中的气泡时间）进行日志记录。这样一来，日志记录就不在关键路径上（§5.1）。为了将空间开销控制在可管理的水平，我们设计了一种算法，只选择一部分机器来记录中间数据（§5.3）。选择性日志记录以恢复时间换取空间占用。

我们设计并实现了 SWIFT，其中包括基于副本和基于日志的恢复，以加快故障恢复。我们的主要贡献总结如下：

▷ 我们提出了一种名为更新-撤销的新机制，用于解决故障导致的模型状态不一致问题，并在不创建额外副本的情况下，利用数据并行中的模型状态副本实现故障恢复。

▷ 我们提出采用日志记录方法，以实现流水线并行中的快速故障恢复。我们使用异步日志记录、在气泡时间内进行日志记录以及选择性日志记录，以降低运行时和空间开销。

▷ 我们基于 PyTorch 实现了 SWIFT，并展示了它在大型深度神经网络模型分布式训练中的优势。在 Wide-ResNet-50 [18], 训练中，采用基于复制的恢复时，与全局检查点、CheckFreq 和 Elastic Horovod 相比，SWIFT 分别将恢复时间缩短了 98.9%、98.1% 和 98.1%。在 BERT [1] 和 ViT [19], 训练中，采用基于日志的恢复时，与全局检查点相比，SWIFT 分别将恢复时间缩短了 57.3% 和 76.3%。利用实验中收集的轨迹数据，我们表明 SWIFT 可将总训练时间最多加速 1.16 倍与最先进方法相比。我们已在 https://github.com/jasperzhong/swift 开源 SWIFT。

2 背景与动机

2.1 分布式DNN训练

我们关注同步分布式DNN训练，即多台机器上的众多工作节点协同工作，迭代训练最新的深度神经网络模型。每次训练迭代包括一个前向计算过程（用于计算损失）和一个反向传播过程（用于计算梯度），计算出的梯度用于模型更新 [20]。与异步训练相比，同步训练能够确保更高的模型准确率，因此在大规模深度神经网络训练中广受欢迎 [5], [20], [21], [22]。

数据并行是分布式DNN训练中应用最广泛的范式 [21], [22]。输入数据被分配到各个工作节点。每个工作节点都拥有一个模型副本，并针对数据子集计算本地梯度。每次迭代中，工作节点之间都会进行梯度同步，以确保模型副本的一致性。

算子并行是一种通过沿非批次轴将深度神经网络模型中的算子拆分到多个工作节点上，从而处理大型深度神经网络的解决方案 [23]。需要通过通信从其他工作节点获取输入数据 [2]。

流水线并行会将一个小批次拆分成更小的微批次，并将它们流水线式地传送到部署在不同工作节点上的深度神经网络模型阶段，以便各工作节点能够同时处理不同的微批次 [12], [17], [24], [25]。在承载相邻阶段的工作节点之间进行点对点通信，以传输中间激活值。GPipe [24] 和一前向一反向（1F1B） [25] 等同步流水线并行调度方式会在每次迭代中刷新流水线，即工作节点会等待该迭代中所有正在执行的微批次完成后，才进入下一次迭代

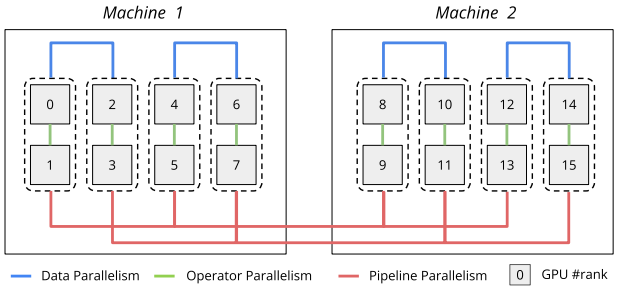


图 2： Megatron-LM 中经过手动优化的三维并行方案，在两台机器上使用 16 个 GPU。深度神经网络模型被拆分为四个流水线并行阶段，每个阶段被划分到两个 GPU 上以实现算子并行，并且每个阶段都有一个副本。一个阶段的副本位于同一台机器上。

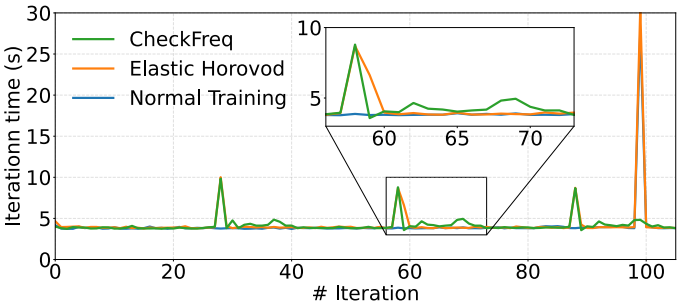


图 3： 无故障执行期间 Wide-ResNet-50 的训练吞吐量。

。尽管模型准确率更高，流水线刷新会导致流水线执行中工作器空闲（即气泡） [5],[12], [24], [25]。对于 GPipe 和 1F1B，气泡时间比为 $(p-1)/(m+p-1)$，其中 p 是阶段数，m 是微批次数 [5]。对于图 1a 所示的示例，气泡时间比为 $3/7$。本文采用 1F1B [25]，因为其气泡时间比相同，但峰值内存使用量低于 GPipe [24]。请注意，我们的方法并不局限于 1F1B。

近期研究将三种并行范式结合起来，称为三维并行 [2], [5], [12], [26], [27]。图 2 展示了 Megatron-LM [2], [5],这一用于 Transformer语言模型的先进训练系统中的手动优化并行方案。尽管该示例使用了数据并行，但副本位于同一台机器上。如果一台机器发生故障，我们将丢失该故障机器上的模型状态。

2.2 大型模型的快照创建问题

CheckFreq [10] 和 Elastic Horovod [11],等最先进的容错方法依赖于快照 操作。每次迭代更新模型状态后，都会在GPU内存中捕获一份模型状态副本（称为快照）；如果 GPU 无法容纳，则将其复制到 CPU内存中。快照操作可以与下一次迭代的前向和反向传播重叠进行。在快照操作完成之前，更新操作的下一次迭代不会开始，从而导致检查点停顿。然而，近年来，深度神经网络模型的规模已从数百万个参数增长到数十亿个参数，变得过大，无法装入单个 GPU [28]。它

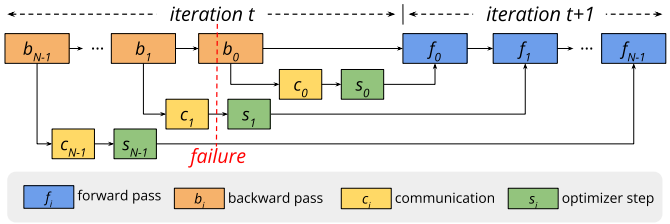


图 4： 逐层无等待更新中的崩溃一致性问题。数字表示层索引。箭头表示依赖关系。红色虚线表示故障。

完整快照越来越难以装入单个 GPU。在这种情况下，快照会被复制到 CPU。

我们通过实验发现，对于大型模型，将快照创建到CPU内存的开销很大，会降低训练吞吐量。我们使用两台机器上的 8 个 32GB V100 GPU，通过数据并行训练一个放大的 Wide-ResNet-50 [18] 模型，其模型状态大小为 9.8GB。快照操作需要将模型状态复制到 CPU。模型设置、训练设置，以及 CheckFreq 和 Elastic Horovod 的设置详见 §7。训练期间，GPU内存占用达到 30.4 GB，无法容纳快照。图 3 显示，在创建快照时（迭代次数 30、60 和 90），使用 CheckFreq 和 Elastic Horovod 的迭代时间显著增加。创建快照后，CheckFreq 的迭代时间仍较长，这表明将快照写入磁盘也会影响正常训练。但全局检查点会导致较大的开销（迭代次数 100），因为它是同步的。有趣的是，使用 CheckFreq 时，检查点停顿确实微乎其微，仅为 0.2 毫秒。这项实验表明，快照操作仍会带来不可忽视的运行时开销，并拖慢训练过程。

2.3 崩溃一致性问题

如图 4 所示，大多数深度学习框架 [8], [9], [26], [29] 采用无等待模型更新。某个深度神经网络层的模型状态一旦获得该层的梯度，就可以立即更新。如果工作节点在更新过程中崩溃，其他工作节点就会处于不一致状态：部分层已更新，而其他层尚未更新。这个问题不仅存在于数据并行中，也存在于流水线并行中，后者会将深度神经网络模型分散到多个工作节点上。由于计算图中的依赖关系，模型状态会在不同时间更新；工作节点托管的那部分模型状态更新后，便会进入下一次训练迭代。这种不一致状态可能导致最终训练所得模型的准确率下降 [6], [30]。

Elastic Horovod 通过快照操作解决了这个问题。然而，对于大型模型而言，这种方法成本可能很高（§2.2）。另一种变通方法是在更新工作节点上的模型状态之前，等待所有层的梯度都准备就绪（即在模型更新前添加一个屏障）。采用这种方法，即使某个工作节点在更新过程中发生故障，其他工作节点仍可完成模型更新并保持一致。不过，这种更新方法需要更多等待时间。在§4中，我们提出了一种更好的解决方案，无需创建快照或增加等待时间，即可解决崩溃一致性问题。

2.4 基于日志的故障恢复

日志记录方法会在运行时记录作业的应用程序数据，并在故障恢复期间精确重放计算。记录的数据主要有两种类型。Spark [13] 和 Ray [14],[31] 记录血统信息，即计算图。其他系统则记录（或仅缓冲）原始中间数据 [15],[16]。例如，在流处理系统的上游备份中 [15], , 上游机器会保留输出队列中的数据，同时下游机器对其进行处理。

在深度神经网络训练中，计算图通常是固定的，图中单个算子的执行时间通常为毫秒级。记录细粒度算子的血统信息会带来显著开销，但收益不大，因为算子很少发生故障 [8]。因此，我们考虑记录中间数据。并行范式决定了通信算子，进而决定了需要记录的数据。数据并行和算子并行使用集合通信，例如全归约和全收集 [27]。集合通信具有复杂的数据依赖（例如多对多），因此增加了日志记录的难度。流水线并行采用点对点通信，这使日志记录更加简单。此外，对于 Transformer 模型，流水线并行的通信数据量远小于数据并行和算子并行；而目前主流的大型模型通常都基于 Transformer [3], [5]。因此，我们主张为流水线并行记录日志。

3 SWIFT 概述

对于给定的分布式DNN训练作业及其特定的并行配置，SWIFT 会在训练作业中引入故障弹性机制，旨在缩短故障恢复时间，同时不影响训练吞吐量和最终模型准确率。分布式DNN训练作业横跨由物理机器组成的集群，每台机器上运行一个或多个工作节点。本文始终关注故障停止模型 [32]，因为它在实际集群中更为常见 [4]。集群中的机器可能在训练过程中崩溃，导致其承载的工作节点的易失性模型状态丢失，即主要存储在 GPU 上的参数和优化器状态。

SWIFT 会在训练作业启动前确定容错策略。如果存在冗余（即模型状态至少在另一台机器上有一个副本），它总会加以利用，因为基于复制的恢复具有较低的运行时和恢复开销。如果副本不可用、采用了流水线并行，并且日志记录值得进行（§5.4），则使用基于日志记录的恢复。如果以上条件均不满足，则仅使用全局检查点。无论如何，都会定期执行全局检查点，以确保系统在发生灾难性故障（例如所有副本或日志数据均丢失）时仍能恢复到正确状态。基于复制的恢复和基于日志记录的恢复可以结合使用，因为大型深度神经网络训练中常见混合并行，例如模型的某些部分采用数据并行，而其他部分采用流水线并行 [27]。

工作节点通过捕获与故障

表 1： 五种典型优化器中使用的算子。EW = 逐元素；可逆 =可逆。

算子		SGD	Adam [33]	AdamW [34]	LAMB [22]	AMSGrad [35]
Inv.	逐元素加法	✓	✓	✓	✓	✓
	标量乘法	✓	✓	✓	✓	✓
	逐元素乘法		✓	✓	✓	✓
	逐元素平方根		✓	✓	✓	✓
	EW 除法		✓	✓	✓	✓
Not Inv.	EW 最大值					✓
	sum				✓	

算法 1 带动量的 SGD

输入： 学习率序列 $\{\eta_t\}_{t=1}^{T}$；权重衰减 $\lambda > 0$; 动量参数 $0 \leq \mu \leq 1$；阻尼用于动量 $0 \leq \tau \leq 1$。
初始化： $x_1 \in \mathbb{R}^d$； $m_0 = 0 \in \mathbb{R}^d$。
对于 $i = 1$ **到** T，**执行**
  $g_t = \nabla f(x_t)$
  $m_t = \mu m_{t-1} + (1-\tau)(g_t + \lambda x_t)$
  $x_{t+1} = x_t - \eta_t m_t$
</style}结束
循环

机器通信时发生的通信错误，可以检测到机器故障。检测到故障后，会将一台替代机器加入训练作业。存活工作节点停止训练，并启动故障恢复流程。存活工作节点首先使用更新-撤销方法（§4）解决模型状态不一致问题（§2.3）。随后恢复故障工作节点的替代节点。对于基于复制的恢复，持有副本的某个存活工作节点会将模型状态广播给替代工作节点（替代工作节点与该存活工作节点进行数据并行）。对于基于日志记录的恢复，替代工作节点会加载最新的检查点，然后根据已记录的数据重新计算丢失的迭代次数，直至恢复到故障发生前的迭代次数。我们还在附录 B 中讨论了多重故障和级联故障。

对于图 2 中的示例，由于副本不可用，且两台机器之间采用了流水线并行，因此可以使用基于日志的恢复。我们记录训练期间机器间通信的数据（§5.1）：GPU 3 和 7 在前向传播中记录中间激活值，而 GPU11 和 15 在反向传播中记录梯度。

4 更新-撤销

我们提出通过撤销更新来解决崩溃一致性问题（§2.3）。我们的想法很简单：如果在模型更新期间发生故障，导致工作节点上的某些参数已更新，而另一些尚未更新，那么存活工作节点将对已更新的参数撤销 更新。除了模型参数，动量等优化器状态也需要恢复。最后，所有工作节点都会恢复到一致的模型状态版本。

我们发现，优化器的许多更新算子在数学上都是可逆的，也就是说，对于算子 f，存在一个逆算子 f^{-1}，可以撤销 f 所执行的操作。例如，逐元素加法和标量乘法等线性算子都是可逆的 [36]。表 1

算法 2 撤销带动量的 SGD

1: **输入：** 学习率 η_t；权重衰减 $\lambda > 0$；动量参数 $0 \leq \mu \leq 1$；动量阻尼 $0 \leq \tau \leq 1$； $x_{t+1} \in \mathbb{R}^d$； $g_t \in \mathbb{R}^d$； $m_t \in \mathbb{R}^d$。2: $x_t = x_{t+1} + \eta_t m_t$3: $m_{t-1} = (m_t - (1-\tau)(g_t + \lambda x_t))/\mu$

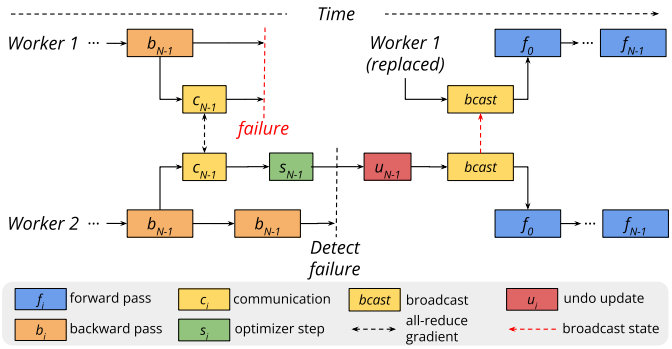


图 5： 采用数据并行和撤销操作的基于复制的恢复。

总结了五种代表性优化器中使用的算子。算法 2 展示了如何撤销带动量的 SGD 的更新（算法 1）（即从 x_{t+1}恢复到 x_t，以及从 m_t 恢复到 m_{t-1}）。更多示例见附录 A。如果优化器仅包含线性算子，那么撤销操作很简单。然而，有些优化器涉及非线性算子，例如，LAMB 优化器 [22]使用参数的 L2 范数缩放梯度。对于 LAMB 优化器，我们还可以保存 L2 范数（一个标量），并据此恢复先前的模型状态。对于 AMSGrad [35]，更新撤销并不适用。尽管撤销算法在数学上是正确的，但由于浮点误差，恢复后的状态可能与原始状态略有不同 [37]。我们的实验表明，这种微小误差不会影响训练后模型的准确率（§7.2）。

图 5 展示了撤销更新如何帮助实现基于复制的故障恢复。两个工作节点使用同步数据并行来训练一个深度神经网络模型。在训练迭代 t 中，工作节点 1 在反向传播期间发生故障，丢失了所有易失状态。此时，工作节点 2 已经更新了层 $N-1$ 的参数，但尚未更新其他层的参数。随后，工作节点 2 撤销层 $N-1$ 的更新，以确保其状态的一致性。当工作节点 1 的替代节点加入系统后，工作节点 2 将其状态发送给工作节点 1，然后两者从迭代 t 继续训练。

撤销更新不需要额外的 GPU 内存，只需缓存最新的梯度 g_t；这是主流深度学习框架 [8], [9], [29] 中的常见做法。在下一次反向传播中计算出下一次迭代所需的梯度 g_{t+1} 时，采用同步训练时，所有工作节点的参数都已通过 g_t 更新，因此 g_{t+1} 可以安全地在内存中覆盖 g_t。只需保留一个版本的梯度。

5 基于日志的故障恢复

5.1 基本机制

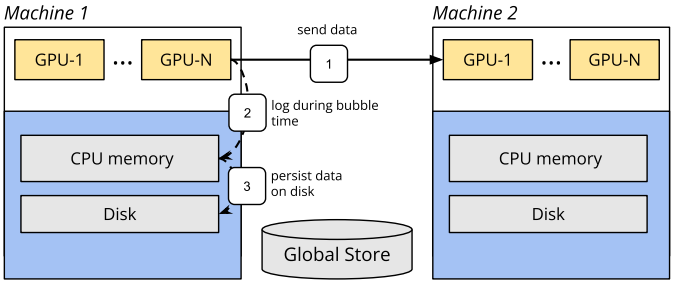
要记录哪些数据。对于多 GPU 机器，GPU 单独发生故障的情况很少见，而机器崩溃则更为常见 [4]。因此，我们不记录机器内部的 GPU 间通信，只记录机器间通信，从而大幅降低日志记录开销。需要记录的数据包括前向传播中的中间激活值和反向传播中的梯度。除了保存原始张量，我们还需要记录一些元数据，包括发送方、接收方和时间戳（其中包含当前训练迭代以及正在训练的当前微批次的标识符）。时间戳用于确定恢复期间重放数据的顺序。

如何记录数据。我们采用的日志记录方法与上游备份（§2.4）类似，即由发送方而非接收方记录消息。这样一来，故障恢复所需的中间数据不会丢失，而是在发生故障时仍保留在发送方机器上。发送方无需在发送消息前存储消息，而可以在后台异步记录。系统为每个工作节点设置一个队列。训练期间，工作节点持续将输出张量推入队列，同时另一个后台线程持续从队列中读取张量并进行日志记录。检测到训练作业发生故障时，每个工作节点都会清空队列中尚未完成的日志记录任务。这种异步日志记录 显著减轻了对训练吞吐量的影响。

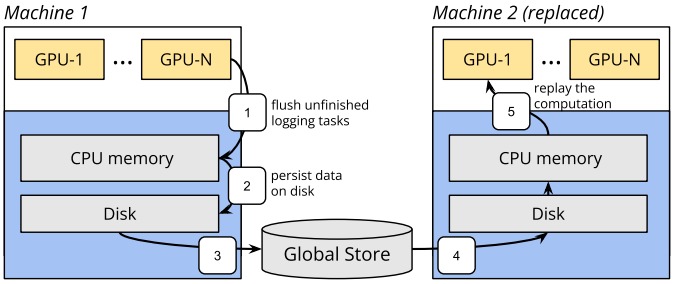
与异步检查点保存（如 CheckFreq）类似，尽管它仍会产生显著开销（§2.2），我们还需要进一步降低异步日志记录的开销。在同步流水线并行训练中，存在许多工作节点处于空闲状态的气泡时间，这正是进行日志记录的理想时机（§2.1）。采用在气泡时间进行日志记录时，输出张量不会在生成后立即记录，而是会等到出现气泡时再记录。等待时间不会超过一次训练迭代，因为每次迭代中总会出现气泡。日志数据传输到 CPU 后，GPU 上的数据即可安全移除。因此，在气泡时间进行日志记录不会导致 GPU 上的数据大量累积。这样一来，日志记录便不在关键路径上。

图 6a 展示了在无故障运行期间如何进行日志记录。工作节点（GPU）将张量发送给下游工作节点后（步骤 1），数据会在 GPU 上保留一小段时间。在接下来的气泡时间内，张量会被异步复制到 CPU内存（步骤 2）。后台线程将 CPU内存中的日志数据写入磁盘，并保存到文件中（步骤 3）。

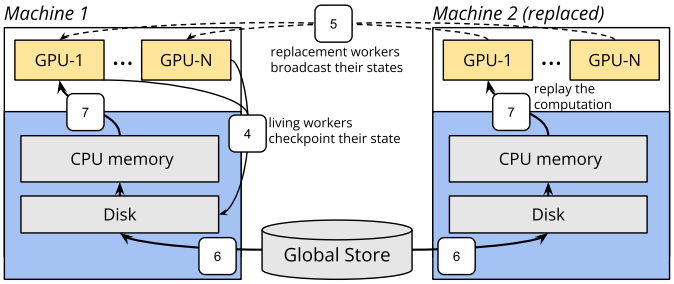
恢复。图 6b 展示了如何利用日志数据实现故障恢复。假设机器2发生故障。仍在运行的上游工作节点（本例中为机器 1）将未记录的数据复制到 CPU内存中（步骤1），然后将其写入磁盘（步骤2）。随后，上游工作节点将其日志文件上传到全局存储（步骤3），例如 HDFS [38]。机器2的替代节点从全局存储下载所需的日志文件到本地磁盘（步骤4），加载最新检查点，并按照时间戳的确切顺序，从日志文件中重放先前接收的张量（步骤5）。步骤3、4和5可以以流水线方式执行



(a) 运行时的日志记录。



(b) 恢复期间的日志记录。



(c) 采用并行恢复时的日志记录。步骤 1 至 3 与图 6b 中的相同，故省略。

图 6： 日志记录机制。

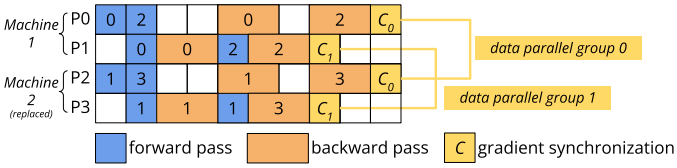


图 7： 并行恢复。微批次数量为 4（$m = 4$），每个数据并行恢复组使用 2 个工作节点（$d = 2$）。每台机器有两个工作节点。假设 machine2 崩溃并被替换。我们将图 1a 中的流水线分解为数据并行的两阶段子流水线。

不同的输出。我们在第 6 节中详细介绍了如何实现确定性。

5.2 并行恢复

我们利用存活工作节点协助恢复故障工作节点。由于所有微批次的中间结果都已记录，我们可以基于已记录的数据执行数据并行训练，以加快丢失状态的重新计算。具体而言，所有工作节点（包括替代工作节点和存活工作节点）都会获取日志文件，并各自持有故障机器上的计算图副本。每个工作节点从日志文件中读取不同微批次的日志数据，并将其作为输入重新计算梯度；随后与计算其他微批次的工作节点同步梯度，再执行模型更新。这样，多个工作节点可以并行重新计算故障机器上的微批次，从而加快恢复，同时确保其逻辑等价于由每个替代工作节点依次执行这些微批次。如果一个批次被划分为 m 个微批次，并且我们使用 d 个工作节点进行并行恢复，则每个工作节点将分配到 m/d 个微批次进行重新计算。

通过将日志文件分割成多个较小的文件。如有必要，存活工作节点将撤销该更新（§4）（图中未显示）。

与纯全局检查点机制相比，最显著的区别在于，存活工作节点无需加载检查点并回滚其训练进度来进行恢复，只有替代机器上重新启动的工作节点需要这样做。恢复范围仅限于故障机器上的局部计算图，而不是整个计算图，从而加快了恢复速度。

垃圾回收。 执行全局检查点保存后，所有较早的日志文件都会失效，并且由于系统此时可以直接加载最新的检查点，因此会进行垃圾回收。尽管随着迭代次数增加，日志大小也会增加，但由于定期执行全局检查点保存，其大小存在上限。因此，全局检查点保存的频率决定了日志大小的上限。我们将在§5.3中进一步讨论存储开销。

一致性。 我们使用时间戳，以与故障发生前执行相同的顺序重放计算，并使用与故障发生前计算相同的输入。请注意，日志记录要求计算具有确定性 （即相同的输入会产生相同的输出）。否则，在使用已记录的数据重新计算时，我们会得到

图7给出了一个示例，其中两台机器运行四阶段训练流水线。机器2（承载阶段2和阶段3）发生故障，而机器1（在正常训练中承载阶段0和阶段1）协助替代机器进行恢复计算。机器1上的工作节点P0与替代机器上的工作节点P2以数据并行方式重新计算阶段2模型，分别使用两个微批次（分别为0、2和1、3）；工作节点P1和工作节点P3重新计算阶段3模型，分别使用微批次0、2和1、3。需要注意的是，采用并行恢复时，梯度同步需要额外耗时。

并行恢复流程如图6c所示。与图 6b类似，在存活工作节点上，首先将未提交日志数据刷新并上传至全局存储（步骤1至3，图中未示出）。随后，存活工作节点保存其状态的检查点（步骤4）。替代工作节点从最新检查点加载其模型状态，并将其状态广播给存活工作节点（步骤5）。与此同时，所有工作节点从全局存储下载日志文件（步骤6），并选择相应微批次的日志数据进行重新计算（步骤7）。恢复完成后，存活工作节点加载其检查点，以恢复原始模型参数和优化器状态（图中未示出）。

5.3 选择性日志记录

记录所有跨机器消息可能会占用大量存储空间。接下来，我们研究采用选择性日志记录时存储空间与恢复时间之间的权衡。我们的想法是对机器进行分组，记录组间通信而不记录组内通信。可以将原始方法视为一种特殊情况，即每台机器构成一个组。这样，如果某个组中的一台机器发生故障，由于我们不记录组内通信，整个机器组上的训练都需要从最新检查点回滚。因此，恢复时间会更长。也就是说，选择性日志记录以恢复时间换取空间开销。一种简单的分组策略是使每个组中的机器数量保持均衡。然而，由于流水线并行中模型划分通常不均衡， [12], 这种分组策略通常并非最优。给定日志数据的存储容量限制，如何对机器进行分组以最小化故障恢复时间？假设我们有 N 台机器，最初创建N 个组。我们对每个组的平均每次迭代计算时间 $R(G_i)$ 进行分析。对于每一对相邻组 G_i 和 G_{i+1} （即在流水线中承载相邻工作节点的组），我们获得它们之间每次迭代的传输数据量$M(G_i, G_{i+1})$。然后，在给定存储容量限制 M_{max}、网络带宽 B （假设带宽均质）和检查点间隔 T（迭代次数）的情况下，我们旨在寻找一种组配置 $\mathcal{G} = \{G_1, \ldots, G_k\}$，使总体恢复时间 R最小：

$$\min_{\mathcal{G}} R(\mathcal{G}) \quad \text{s.t. } M(\mathcal{G}) \leq M_{\text{max}},$$

其中， $M(\mathcal{G})$ 表示日志数据所需的总存储空间。如§5.1 所述，它由全局检查点频率决定：

$$M(\mathcal{G}) = T \cdot \sum_{G_i, G_{i+1} \in \mathcal{G}} M(G_i, G_{i+1}).$$

假设我们合并两个相邻组 G_i 和 G_{i+1}，则合并后组的恢复时间为：

$$R(G_i, G_{i+1}) = R(G_i) + R(G_{i+1}) + M(G_i, G_{i+1})/B,$$

其中， $M(G_i, G_{i+1})/B$ 表示两个相邻组之间的点对点通信时间。为简化起见，我们忽略气泡时间，并推导总恢复时间 R 和总空间开销 M的变化：

$$\begin{aligned}\Delta R = R(G_i, G_{i+1}) \cdot \frac{|G_i| + |G_{i+1}|}{N} - R(G_i) \cdot \frac{|G_i|}{N} \\ - R(G_{i+1}) \cdot \frac{|G_{i+1}|}{N}, \\ \Delta M = M(G_i, G_{i+1}) \cdot T,\end{aligned}$$

其中， $|G_i|$ 是 G_i中的机器数量。假设每台机器的故障概率相同，据此计算 ΔR。注意， ΔR 始终为正。合并 G_i 和 G_{i+1}时，我们将单位存储空间减少量对应的恢复时间增量 ΔR 最小化，即最小化 $\Delta R/\Delta M$。

为了确定机器的分组，我们反复合并 $\Delta R/\Delta M$最小的两个相邻组，直到总空间占用小于 M_{max}。注意，该过程最多执行 $N-1$ 次迭代；此时所有机器将组成一个组，且不再进行日志记录。因此，时间复杂度最多为 $O(N^2)$。

如果采用并行恢复，则一个组 G_i 的恢复最多由 $\lfloor N/|G_i| \rfloor$ 个数据并行组并行执行。为简单起见，我们假设其能够实现与数据并行的线性可扩展性。因此，我们将计算中的 $R(G_i)$ 除以 $\lfloor N/|G_i| \rfloor$。

5.4 用例

并非所有情况都适合日志记录。例如，当日志大小远超模型大小时，最好为模型创建检查点。通常，基于 CNN 的模型的中间激活值会非常庞大，不适合进行日志记录（甚至不适合流水线并行） [39]。我们可以计算每次迭代的日志大小。对于基于 Transformer 的模型，一个微批次中的中间激活值/梯度大小为 micro_batch_size ×hidden_size ×sequence_length [28]。此外，我们可以根据流水线调度（§2.1）计算气泡时间比。给定迭代时间和 PCIe 带宽，我们可以确定日志数据能否在气泡时间内从 GPU 传输到 CPU。如果不能，那么进行日志记录就不值得。

6 实现

我们使用 2.6k 行 Python 代码，在 PyTorch 1.9.0 [9]和 NCCL 2.7.6 [40], 上实现了 SWIFT。我们还为 PyTorch 和 NCCL 添加了约 400 行 C++代码。

故障检测。 我们在每个工作节点上启动一个后台线程，使用 NCCL 的 ncclCommGetAsyncError() 函数持续轮询是否发生了通信故障。如果发生故障，该工作节点首先在全局键值存储中将故障标志设为 true，然后中止自己的 NCCL 通信器。全局键值存储与主控机器（rank 0）位于同一位置。其他工作节点的后台线程也会轮询全局键值存储中的该标志；如果某个工作节点发现标志已设为 true，就会中止自己的 NCCL 通信器。

更新-撤销。 在数据并行中，我们会在每个张量的全归约操作之后插入一个 CUDA 事件，并在更新梯度对应的参数之前查询该事件是否已完成。如果尚未完成，就等待全归约操作完成。完成后，将启动对应参数的 CUDA 内核，以更新参数和优化器状态，并将该参数标记为已更新。请注意，即使此时发生故障，我们也必须让这些内核执行完毕。发生故障后，存活工作节点会撤销已标记为已更新的参数更新。

在流水线并行中，由于计算依赖关系，不同阶段的模型参数会在不同时间点进行更新。因此，故障发生后，存活工作节点需要交换当前迭代编号，以确定故障前共识迭代编号。迭代编号大于故障前共识迭代编号的工作节点需要撤销更新。

日志记录。我们使用专用的 CUDA 流，将日志数据从 GPU 复制到 CPU，以进行异步日志记录。复制操作后，我们插入一个 CUDA 事件，以检查复制是否完成。主线程在气泡时间启动异步复制操作后，会将 CUDA 事件及相应的张量和元数据发送到队列。后台线程持续从队列中读取项目，并检查异步复制是否完成

表 2： 基准模型。数据并行 = 数据并行。流水线并行 =流水线并行。

模型	数据集	批次 size	#参数量（十亿）	并行方式
Wide-ResNet-50	ImageNet [41]	256	1.23	DP
ViT-128/32	ImageNet	4096	1.64	PP
BERT-128	维基百科 [1]	512	1.11	PP

方法是检查 CUDA 事件的状态。如果复制已完成，该线程就会将数据保存到文件中。对于 §5.1 中所述的全局存储，我们支持 HDFS [38]和 Amazon S3。

日志记录中的确定性。 深度神经网络训练中的非确定性可能来自随机数种子和算法本身。例如，在深度神经网络训练中，cuDNN 中的某些卷积算法具有非确定性[1]。我们将 torch.backend.cudnn.deterministic= 设置为 True，以解决此问题。此外，对于卷积运算，PyTorch 也会在首次运行时对多种算法进行基准测试，选出最快的算法，并缓存这一选择，以便之后直接使用同一种算法。不过，对于相同输入，不同确定性算法的计算结果仍存在细微差异。为确保工作节点在故障恢复后选择与故障前相同的卷积算法，我们会保存先前的基准测试结果，以供故障恢复使用。

用法。 我们为用户提供了易于使用的接口。用户只需提供一个用户定义函数（UDF），用于训练一个迭代，并指定容错和训练配置。这样，用户在训练模型时即可获得容错支持；发生故障后，系统也能自动执行恢复，无需用户介入。

7 评估

测试平台。我们在 16 台 DGX-2 机器上进行实验，每台机器配备八块 32 GB Tesla V100 GPU（采用 NVLink 互连）、160 个 CPU、1.5 TB 内存和 3.6 TB NVMe SSD 磁盘。这些机器通过 40 Gbps 以太网连接。我们在这些机器上搭建了一个 HDFS 集群作为全局存储。

基准模型。 我们在表 2 所示的设置下，评估 SWIFT 在训练大型图像分类模型和具有数十亿参数的语言模型时的表现。我们对相应论文中的原始模型进行扩展：对于 Wide-ResNet-50 [18],，我们将基础通道大小从 64 增加到 320；对于 BERT- Large [1] 和 ViT-Large/32 [19],，我们将 Transformer 层的数量从 24 增加到 128，保持隐藏层维度不变，并分别将扩展后的模型称为 BERT-128 和 ViT-128/32。我们使用数据并行，在两台机器上训练 Wide-ResNet-50，每台机器使用四个 GPU。为了训练 ViT-128/32 或 BERT-128（最大序列长度为 128），我们在全部 16 台机器上使用 128 阶段流水线，其中每个 Transformer 层占用一个 GPU。对于 Wide-ResNet-50 和 ViT-128/32，我们使用带动量的 SGD；对于 BERT-128 [1], [19], [19]，我们使用 Adam。我们选择能够最大化性能的微批次数量，ViT-128/32 和 BERT-128 分别使用 16 和 4。SWIFT 对 Wide-ResNet-50 应用基于复制的恢复，对 ViT-128/32 和 BERT-128 应用基于日志的恢复（默认情况下，在选择性日志记录中分别使用 16 个机器组和 8 个机器组）。我们对每个实验运行 200 次迭代，在第 100 次迭代开始时执行一次全局检查点，并在第 150 次迭代开始时终止一台机器（rank 1）。

基线。我们将 SWIFT 与全局检查点保存（PyTorch 中的默认方式）、CheckFreq [10] 和ElasticHorovod [11]进行比较。我们使用 CheckFreq 的开源代码 [42]，并将 Elastic Horovod 的快照实现替换为 CheckFreq 的实现，因为前者未实现将快照保存到 CPU。我们根据 CheckFreq 提出的算法，并采用与 CheckFreq 实验相同的可接受检查点开销（3.5%），计算出最佳快照频率（每 30 次迭代一次）。对于基于日志的恢复，我们仅与全局检查点保存进行比较，因为其检查点开销已经很低（流水线并行训练中的检查点保存采用流水线方式），且 CheckFreq 的性能也会相近。Elastic Horovod 不适用，因为它仅支持数据并行。我们还引入了一种同步日志记录方法（在发送张量前调用 torch.save()）作为基线，以评估异步日志记录以及在气泡时间内记录日志的效果（§5.1）。

指标。 我们评估无故障执行期间的训练吞吐量和迭代时间、恢复期间的吞吐量，以及恢复时间。训练吞吐量按所有工作节点在每次训练迭代中处理的图像数（或词元数）计算。初始化时间是指从工作节点检测到故障，到故障工作节点的替代节点加入训练作业所经历的时间。恢复时间是指从替代工作节点加入训练作业，到恢复至故障发生前的迭代次数所经历的时间。

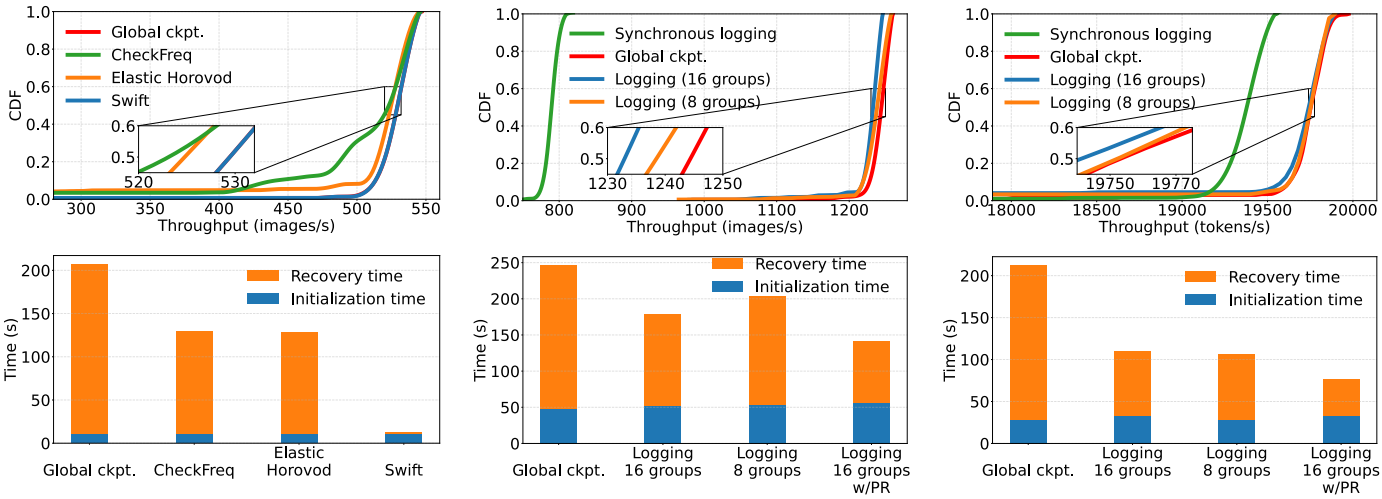
7.1 宏基准测试

基于复制的恢复。 图 8a 显示，在无故障训练期间，SWIFT 的基于复制的恢复所产生的运行时开销低于最先进的方法。与正常训练（不使用任何检查点或快照）相比，CheckFreq 和 Elastic Horovod 的训练吞吐量有所下降。

图 8a 还展示了在第 150 次迭代发生机器故障时的恢复时间。全局检查点需要较长的恢复时间，因为所有工作节点都必须加载检查点并重新计算丢失的迭代（本实验中为 50 次迭代）。CheckFreq 和 Elastic Horovod 虽然会频繁创建快照，但仍需重新计算 30 次迭代（上一个快照是在第 120 次迭代时创建的）。使用 SWIFT 的基于复制的恢复时，存活工作节点通过撤销更新来解决不一致问题，然后将副本广播给替代工作节点。与全局检查点、CheckFreq 和 Elastic Horovod 相比，该方法分别将恢复时间缩短了 98.9%、98.1% 和 98.1%。

基于日志的恢复。 图 8b 和图 8c 显示，在 ViT-128/32 的无故障训练期间，SWIFT 的日志记录速度略慢于全局检查点保存；而在 BERT-128 上，其吞吐量与全局检查点保存相近。ViT-128/32 略有延迟，是因为训练时使用了较大的批次（4096），并且

1. cuDNN 可复现性：https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#reproducibility



(a) Wide-ResNet-50 (b) ViT-128/32 (c) BERT-128

图 8： 无故障训练的吞吐量（上）和恢复时间（下）。PR 表示并行恢复。

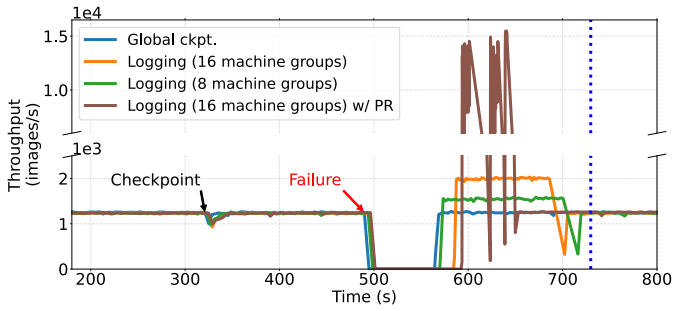


图 9： 故障恢复期间 ViT-128/32 的训练吞吐量。蓝色虚线表示采用全局检查点后恢复完成的时间。

表3： 每次迭代日志记录造成的空间开销。

模型	#机器数组	日志记录总量大小（GB）	平均消耗带宽（GB/s）
ViT-128/32	16	24.66	0.23
	8	11.51	0.11
BERT-128	16	8.05	0.075
	8	3.76	0.035

日志数据量相对较大。同步日志记录会显著降低训练吞吐量，尤其是在训练 ViT-128/32 时，因为其日志记录的数据量高于 BERT-128。SWIFT 的异步日志记录以及在气泡时间内进行的日志记录（§5.1）将日志记录移出关键路径，因此其吞吐量与全局检查点保存相近。

图 8b 和图 8c 显示，采用日志记录的恢复时间明显短于采用全局检查点的恢复时间。在 16 个机器组的情况下，ViT-128/32 和 BERT-128 的恢复时间分别减少了 36.0% 和 58.5%。这是因为只需恢复故障机器上的 8 阶段子流水线，而采用全局检查点时则需要重新运行整个 128 阶段流水线。需要注意的是，日志记录所需的初始化时间略长，因为它需要执行额外的初始化操作，例如创建 CUDA 流和日志记录线程

机器组大小。图 8b 和图 8c 还展示了不同机器组大小对基于日志的恢复的训练吞吐量和恢复时间的影响。在图 8b 中，使用 8 个机器组时，吞吐量与全局检查点保存相近，这是因为其日志数据少于使用 16 个机器组时的数据（§5.3）。在图 9和图 8b 中，我们观察到，使用 8 个机器组进行日志记录需要更长的恢复时间，因为要在两台机器上恢复一个 16 阶段子流水线，而使用 16 个机器组时只需恢复一个 8 阶段子流水线。表3展示了不同模型和不同机器组数量下，每次迭代的日志总大小，以及日志记录在气泡时间内占用的平均带宽。这体现了选择性日志记录在恢复时间与空间开销之间的权衡（§5.3）。

并行恢复。对于图 9、图 8b 和图 8c 中采用并行恢复（§5.2）的日志记录场景，我们使用 16 个工作节点（GPU）并发地为一个故障工作节点执行恢复计算。我们发现，与全局检查点保存相比，并行恢复显著提高了 ViT-128/32 的训练吞吐量，从 12.5 倍提升至 15 倍（BERT-128 的结果类似），这是因为恢复时间缩短了（ViT-128/32 和 BERT-128 分别缩短了 57.3% 和 76.3%）。并行恢复下的吞吐量波动是因为并行恢复速度很快，导致文件传输成为瓶颈；也就是说，重放完较早的日志文件时，新的日志文件尚未从 HDFS 下载完成。

时空权衡。图10进一步评估了采用选择性日志记录时恢复时间与空间开销之间的权衡。在给定最大存储容量的情况下，我们使用 §5.3 中的算法来决定如何对机器分组。对于这两种深度神经网络模型，随着我们降低空间阈值，恢复时间会变长。在实际应用中，我们可以利用图中曲线找出合适的权衡方案。分组配置见附录 C。

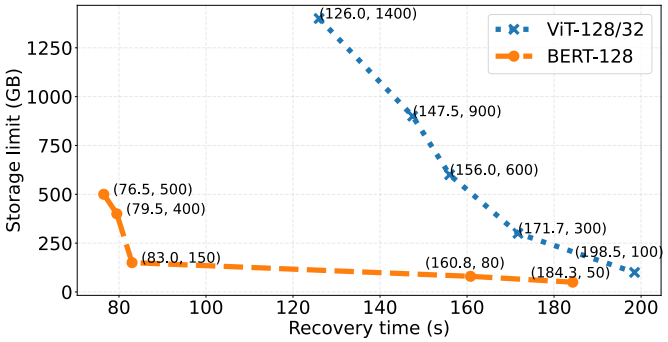
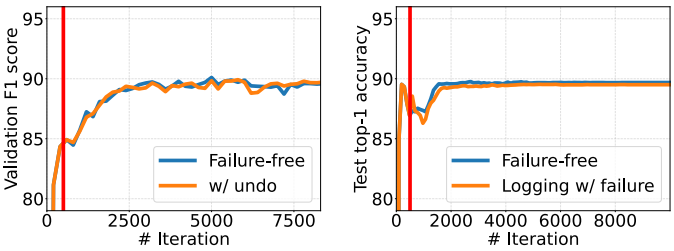


图 10： 恢复时间与存储空间上限之间的权衡。标记：（恢复时间，单位为秒；存储上限，单位为吉字节）。



(a) BERT-Large (b) ViT-Base/32

图 11： 端到端训练。红线表示在第 500 次迭代时发生故障。

7.2 端到端训练

接下来，我们运行端到端训练，以验证 SWIFT 不会影响训练后模型的准确率。在图 11a 中，我们使用流水线并行，在两台机器上的 8 个 GPU 上，利用 Adam 优化器对 SQuAD-v1.1 数据集 [43]，上的 BERT-Large [1] 进行微调。我们在此实验中禁用了日志记录，但检查了更新-撤销（§4）可能产生的影响。我们在第 500 次迭代结束时终止一台机器，故意在第 500 次迭代额外执行一次更新，然后撤销该更新。我们观察到，更新-撤销不会影响最终的微调准确率。

在图 11b 中，我们在 CIFAR-100 数据集上使用带动量的 SGD 对 ViT-Base/32 [19] 进行微调，共迭代 10000 次，并在三台机器上使用 12 个 GPU 进行流水线并行。我们在第 500 次迭代结束时关闭了位于流水线中间阶段的机器（即机器1，其工作节点的秩为 4 至 7）。我们既未将机器分组以进行日志记录，也未启用并行恢复。我们发现，与无故障训练相比，基于日志记录的故障恢复不会降低准确率。

7.3 仿真研究

我们进一步通过模拟研究 SWIFT 对端到端训练时间的影响。模拟设置见表 4（其他设置与第 7.1 节中的实验设置相同）。我们根据实验测得的迭代时间和训练迭代总次数，计算无故障时预期的端到端训练时间。对于 Wide-ResNet-50 和 ViT-128/32，我们按照常见的机器学习实践，假设每个训练轮次结束时保存一个检查点 {v1}

表 4： 仿真研究中的训练工作负载。

模型	迭代次数总计 迭代次数	检查点 时间间隔	端到端训练 无故障时间
Wide-ResNet-50	450,360	5,004	479.4小时
ViT-128/32	93,600	312	85.6小时
BERT-128	500,000	5,000	461.1小时

表5： 发生故障时的模拟端到端训练时间。

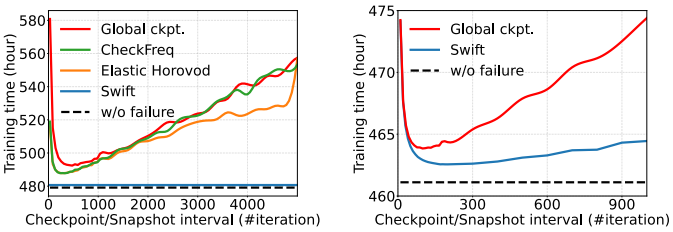
模型	故障次数	全局检查点	SWIFT	加速比
Wide-ResNet-50	28	557.4小时	480.7小时	1.16倍
ViT-128/32	5	86.4小时	86.0小时	1.01倍
BERT-128	27	524.2小时	476.1小时	1.10倍

。对于 BERT-128，我们假设每 5000 次迭代进行一次检查点保存，这相当于其训练迭代总次数的 1%。随后，我们假设故障间隔中位时间为 17 小时（遵循 [6]），并在训练期间均匀随机地注入故障。每次模拟重复十次，并报告平均结果。

端到端训练时间。如表5所示，与全局检查点相比，SWIFT 能显著缩短长时间运行作业的端到端训练时间。具体而言，SWIFT 可将 ImageNet 上训练 Wide-ResNet-50 和在维基百科数据集上预训练 BERT-128 的端到端训练速度分别提升 1.16 倍和 1.10 倍，相当于节省 77 小时和 48 小时的训练时间。对于在 ImageNet 上训练 ViT-128/32 这类运行时间较短的作业，发生的故障较少，因此从快速故障恢复中获益较少。我们还比较了使用 Elastic Horovod 和 CheckFreq 训练 Wide-ResNet-50 的端到端训练时间。在模拟中，我们使用第 7.1 节收集的数据来考虑快照开销，并采用与第 7.1 节相同的快照频率。使用 CheckFreq 进行端到端训练需要 518.9 小时，使用 Elastic Horovod 则需要 515.9 小时。SWIFT 的速度分别是 CheckFreq 和 Elastic Horovod 的 1.08 倍和 1.07 倍。

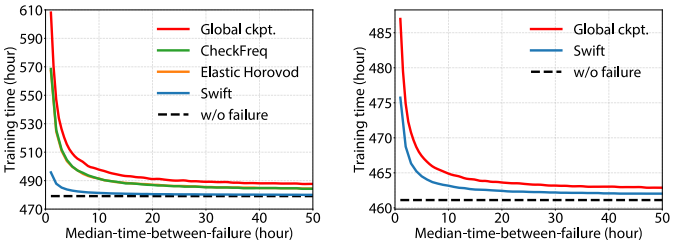
检查点频率的影响。我们改变检查点/快照频率，以研究其对端到端训练时间的影响。对于 SWIFT 中基于复制的恢复，我们保持检查点频率不变，因为它不需要频繁保存检查点。如图12所示，在所有情况下，SWIFT 的训练时间都短于其他方法。根据曲线，可以为每种方法找到最佳检查点频率，从而实现最短训练时间。比较各方法的最佳情况，对于 Wide-ResNet-50，SWIFT 相比全局检查点保存、CheckFreq 和 Elastic Horovod 分别节省了 11.8 小时、7.1 小时和 7.2 小时；对于 BERT-128，SWIFT 相比全局检查点保存节省了 1.3 小时——由于 BERT-128 的检查点保存开销极小（0.93 秒），改进幅度有限。

故障频率的影响。我们进一步调整故障间隔中位数，以研究其对端到端训练时间的影响，同时将检查点/快照频率固定为图12给出的最佳频率。图13显示，故障越频繁，SWIFT 的加速比越高；当故障不频繁时，SWIFT 的



(a) Wide-ResNet-50 (b) BERT-128

图 12： 检查点频率的影响。



(a) Wide-ResNet-50 (b) BERT-128

图 13： 故障频率的影响。

训练时间也比所有其他方法都短。

8 相关工作

弹性训练。 大多数深度学习作业使用静态作业配置（例如工作节点数量）。在弹性训练中，工作节点可以加入或离开。作业可以扩展规模，以利用临时闲置资源（例如云计算中的竞价实例），也可以缩减规模，为高优先级作业预留资源。然而，大多数弹性训练工作仍然依赖检查点重启方法来避免崩溃一致性问题（§2.3）。SWIFT 可以使用更新-撤销（§4）解决不一致问题，从而使弹性训练受益（例如，在新工作节点加入时广播工作节点的状态）。

深度学习系统中的检查点保存。Check-N-Run [7] 提出了一种专为训练深度学习推荐模型设计的增量式检查点保存方法，它利用了每次迭代只更新推荐模型一部分参数这一特点。该方法与我们的工作互为补充，因为 SWIFT 并不限于推荐模型。推荐模型中的 MLP 层通常采用数据并行方式训练 [6],，可以受益于基于复制的恢复。Orpheus [48] 也提出了增量式检查点保存方法，但它存储的是梯度的充分向量，其大小远小于梯度本身。恢复时，系统利用存储的充分向量重建梯度，并将其应用于检查点，以恢复模型状态。我们可以将自己的日志记录方法视为对他们方法的扩展。我们考虑的是机器上计算图的充分向量（例如，流水线并行中的中间激活值/梯度），而不是单个算子的充分向量。我们由上游机器异步记录数据，而他们为了保证一致性，需要采用开销高昂的同步日志记录。近期的研究也提出了部分恢复方法：只加载故障机器的检查点，然后继续进行训练 [6], [30]。部分恢复避免了全局回滚，但由于工作节点之间的模型状态不一致，会造成准确率损失 [6]。相比之下，SWIFT 在缩短恢复时间的同时，不会降低最终模型准确率。

大规模深度神经网络训练。 除了并行化之外，大规模深度神经网络训练的其他互补技术还包括内存优化 [28]和混合精度训练 [49]。SWIFT 可以与其中许多技术结合使用。例如，我们可以将基于复制的恢复与完全分片数据并行（FSDP）结合，这是一种流行的内存优化技术，可将模型状态分片存储在数据并行工作节点之间 [28]。我们可以为分片模型状态的每个部分维护两份副本，以实现故障韧性。此外，混合精度训练可以通过对中间数据使用较低精度来减少日志大小 [49]。

9 结论

本文提出了 SWIFT，这是一种可加速分布式DNN训练故障恢复的新型设计。SWIFT 利用数据并行训练中的冗余来恢复故障，并通过撤销更新解决崩溃一致性问题。SWIFT 倡导在流水线并行训练中进行日志记录，在运行时记录机器间中间数据，并将需要重新执行的计算图限制为故障工作节点上的计算图。我们还设计了并行恢复以进一步加快恢复，并通过选择性日志记录探索恢复时间与空间开销之间的权衡。与最先进的方法相比，大量评估结果表明，SWIFT 能显著加快故障恢复，同时不影响训练吞吐量和模型准确率。

参考文献

[1] J. Devlin、M.-W. Chang、K. Lee 和 K. Toutanova，《BERT：面向语言理解的深度双向 Transformer 预训练》，载于 北美计算语言学协会：人类语言技术分会论文集，2019年。[2] M. Shoeybi、M. Patwary、R. Puri、P. LeGresley、J. Casper 和 B. Catanzaro，《Megatron-LM：使用模型并行训练数十亿参数语言模型》，*arXiv* 预印本，2019年。[3] T. Brown、B. Mann、N. Ryder、M. Subbiah、J. D. Kaplan、P. Dhariwal、A. Neelakantan、P. Shyam、G. Sastry、A. Askell、S. Agarwal、A. Herbert-Voss、G. Krueger、T. Henighan、R. Child、A. Ramesh、D. Ziegler、J. Wu、C. Winter、C. Hesse、M. Chen、E. Sigler、M. Litwin、S. Gray、B. Chess、J. Clark、C. Berner、S. McCandlish、A. Radford、I. Sutskever 和 D. Amodei，《语言模型是少样本学习者》，载于 神经信息处理系统进展论文集，2020年。[4] M. Jeon、S. Venkataraman、A. Phanishayee、J. Qian、W. Xiao 和 F. Yang，《面向深度神经网络训练工作负载的大规模多租户 GPU 集群分析》，载于 *2019*年 *USENIX* 年度技术会议论文集，2019年。[5] D. Narayanan、M. Shoeybi、J. Casper、P. LeGresley、M. Patwary、V. Korthikanti、D. Vainbrand、P. Kashinkunti、J. Bernauer、B. Catanzaro、A. Phanishayee 和 M. Zaharia，《使用 Megatron-LM 在 GPU 集群上高效训练大规模语言模型》，载于 高性能计算、网络、存储与分析国际会议论文集，2021年。[6] K. Maeng、S. Bharuka、I. Gao、M. Jeffrey、V. Saraph、B.-Y. Su、C. Trippel、J. Yang、M. Rabbat、B. Lucia 和 C.-J. Wu，《理解并改进深度学习推荐的容错训练：部分恢复》，载于 第*4*届机器学习与系统会议论文集，2021年。

[7] A. Eisenman、K. K. Matam、S. Ingram、D. Mudigere、R. Krishnamoorthi、K. Nair、M. Smelyanskiy 和 M. Annavaram，《Check-N-Run：用于训练深度学习推荐模型的检查点系统》，载于 第*19*届 *USENIX* 网络系统设计与实现研讨会论文集，2022年。 [8] M. Abadi、P. Barham、J. Chen、Z. Chen、A. Davis、J. Dean、M. Devin、S. Ghemawat、G. Irving、M. Isard、M. Kudlur、J. Levenberg、R. Monga、S. Moore、D. G. Murray、B. Steiner、P. Tucker、V. Vasudevan、P. Warden、M. Wicke、Y. Yu 和 X. Zheng，《TensorFlow：一个大规模机器学习系统》，载于 第*12*届 *USENIX* 操作系统设计与实现研讨会论文集，2016年。[9] A. Paszke、S. Gross、F. Massa、A. Lerer、J. Bradbury、G. Chanan、T. Killeen、Z. Lin、N. Gimelshein、L. Antiga、A. Desmaison、A. Kopf、E. Yang、Z. DeVito、M. Raison、A. Tejani、S. Chilamkurthy、B. Steiner、L. Fang、J. Bai 和 S. Chintala，《PyTorch：一款命令式风格的高性能深度学习库》，载于 神经信息处理系统进展论文集，2019年。 [10] J. Mohan、A. Phanishayee 和 V. Chidambaram，《CheckFreq：频繁、细粒度的深度神经网络检查点保存》，载于 第*19*届 *USENIX* 文件与存储技术会议论文集，2021年。 [11] T. H. Authors，《Elastic Horovod》，2022年，https://horovod.readthedocs.io/en/stable/elastic_include.html。 [12] S. Fan、Y. Rong、C. Meng、Z. Cao、S. Wang、Z. Zheng、C. Wu、G. Long、J. Yang、L. Xia、L. Diao、X. Liu 和 W. Lin，《Dapple：一种用于训练大型模型的流水线数据并行方法》，载于第*26*届 *ACM SIGPLAN* 并行编程原理与实践研讨会论文集，2021年。 [13] M. Zaharia、M. Chowdhury、T. Das、A. Dave、J. Ma、M. McCauly、M. J. Franklin、S. Shenker 和 I. Stoica，《弹性分布式数据集：一种用于内存集群计算的容错抽象》，载于 第*9*届 *USENIX* 网络系统设计与实现研讨会论文集，2012年。 [14] S. Wang、J. Liagouris、R. Nishihara、P. Moritz、U. Misra、A. Tumanov 和 I. Stoica，《Lineage Stash：关键路径之外的容错》，载于 第*27*届 *ACM* 操作系统原理研讨会论文集，2019年。 [15] J.-H. Hwang、M. Balazinska、A. Rasin、U. Cetintemel、M. Stonebraker 和 S. Zdonik，《分布式流处理的高可用性算法》，载于 国际数据工程会议论文集，2005年。[16] Y. Shen 、 G. Chen 、 H. Jagadish 、 W. Lu 、 B. C. Ooi 和 B. M. Tudor，《分布式图处理系统中的快速故障恢复》，载于 超大型数据库基金会论文集，2014年。 [17] D. Narayanan、A. Harlap、A. Phanishayee、V. Seshadri、N. R. Devanur、G. R. Ganger、P. B. Gibbons 和 M. Zaharia，《PipeDream：面向深度神经网络训练的通用化流水线并行》，载于 第*27*届 *ACM* 操作系统原理研讨会论文集，2019年。 [18] S. Zagoruyko 和 N. Komodakis，《宽残差网络》，*arXiv* 预印本，2016年。[19] A. Dosovitskiy、L. Beyer、A. Kolesnikov、D. Weissenborn、X. Zhai、T. Unterthiner、M. Dehghani、M. Minderer、G. Heigold、S. Gelly、J. Uszkoreit 和 N. Houlsby，《一张图像胜过 16×16 个词：用于大规模图像识别的 Transformer》，载于 国际学习表征会议论文集，2020年。[20] J. Chen 、 X. Pan 、 R. Monga 、 S. Bengio 和 R. Jozefowicz，《重新审视分布式同步随机梯度下降》，*arXiv* 预印本，2016年。 [21] P. Goyal、P. Dollár、R. Girshick、P. Noordhuis、L. Wesolowski、A. Kyrola、A. Tulloch、Y. Jia 和 K. He，《准确的大批次随机梯度下降：1小时内训练 ImageNet》，*arXiv* 预印本，2017年。 [22] Y. You、J. Li、S. Reddi、J. Hseu、S. Kumar、S. Bhojanapalli、X. Song、J. Demmel、K. Keutzer 和 C.-J. Hsieh，《深度学习的大批量优化：76 分钟内训练 BERT 》，载于 国际学习表征会议论文集，2020年。 [23] Z. Jia、M. Zaharia 和 A. Aiken，《超越深度神经网络的数据并行与模型并行》，载于 第*2*届机器学习与系统会议论文集，2019年。 [24] Y. Huang、Y. Cheng、A. Bapna、O. Firat、D. Chen、M. Chen、H. Lee、J. Ngiam、Q. V. Le、Y. Wu 和 z. Chen，《GPipe：使用流水线并行高效训练巨型神经网络》，载于 神经信息处理系统进展论文集，2019年。 [25] D. Narayanan、A. Phanishayee、K. Shi、X. Chen 和 M. Zaharia，《内存高效的流水线并行深度神经网络训练》，载于 国际机器学习会议论文集，2021年。 [26] J. Yuan、X. Li、C. Cheng、J. Liu、R. Guo、S. Cai、C. Yao、F. Yang、X. Yi、C. Wu、H. Zhang 和 J. Zhao，《OneFlow：从头重新设计分布式深度学习框架》，*arXiv* 预印本，2021年。

[27] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, 和 I. Stoica，《Alpa：面向分布式深度学习的跨算子与算子内并行自动化》，载于第16届 USENIX 操作系统设计与实现研讨会论文集，2022年。[28] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, 和 Y. He，《ZeRO-Infinity：突破极大规模深度学习的 GPU 内存瓶颈》，载于高性能计算、网络、存储与分析国际会议论文集，2021年。[29] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, 和 Z. Zhang，《MXNet：面向异构分布式系统的灵活高效机器学习库》，arXiv 预印本，2015年。[30] A. Qiao, B. Aragam, B. Zhang, 和 E. Xing，《迭代收敛机器学习中的容错》，载于国际机器学习会议，2019年。[31] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, 和 I. Stoica，《Ray：面向新兴 AI 应用的分布式框架》，载于第13届 USENIX 操作系统设计与实现研讨会论文集，2018年。[32] R. D. Schlichting 和 F. B. Schneider，《失效停止处理器：设计容错计算系统的一种方法》，《ACM计算机系统汇刊》，1983年。[33] D. P. Kingma 和 J. Ba，《Adam：一种随机优化方法》，载于国际学习表征会议论文集，2015年。[34] I. Loshchilov 和 F. Hutter，《解耦权重衰减正则化》，载于国际学习表征会议论文集，2019年。[35] S. J. Reddi, S. Kale, 和 S. Kumar，《论 Adam 及其后续方法的收敛性》，载于国际学习表征会议论文集，2018年。[36] R. Harte，《有界线性算子的可逆性与奇异性》。Courier Dover Publications，2016年。[37] A. N. Gomez, M. Ren, R. Urtasun, 和 R. B. Grosse，《可逆残差网络：无需存储激活值的反向传播》，载于国际神经信息处理系统会议论文集，2017年。[38] K. Shvachko, H. Kuang, S. Radia, 和 R. Chansler，《Hadoop 分布式文件系统》，载于 IEEE 第26届大容量存储系统与技术研讨会论文集，2010年。[39] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, 和 S. W. Keckler，《vDNN：用于可扩展、节省内存的神经网络设计的虚拟化深度神经网络》，载于 IEEE/ACM 第49届年度微体系结构国际研讨会论文集，2016年。[40] J. Sylvain，《使用 NCCL 2 优化 GPU 间集合通信操作》，2017年，https://developer.nvidia.com/nccl。[41] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, 和 L. Fei-Fei，《ImageNet 大规模视觉识别挑战赛》，载于《国际计算机视觉杂志》论文集，2015年。[42] 《GitHub 上的 Checkfreq 代码》，2022年，https://github.com/msr-fiddle/ CheckFreq。[43] P. Rajpurkar, J. Zhang, K. Lopyrev, 和 P. Liang，《Squad：100,000+用于文本机器理解的问题》，载于2016年自然语言处理经验方法会议论文集，2016年。[44] K. He, X. Zhang, S. Ren, 和 J. Sun，《深度残差网络中的恒等映射》，载于欧洲计算机视觉会议论文集，2016年。[45] L. Mai, G. Li, M. Wagenländer, K. Fertakis, A.-O. Brabete, 和 P. Pietzuch，《Kungfu：使分布式机器学习训练具备自适应能力》，载于第14届 USENIX 操作系统设计与实现研讨会论文集，2020年。[46] Y. Peng, Y. Bao, Y. Chen, C. Wu, 和 C. Guo，《Optimus：面向深度学习集群的高效动态资源调度器》，载于第13届欧洲计算机系统会议论文集，2018年。[47] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, 和 E. P. Xing，《Pollux：面向高效吞吐量优化深度学习的协同自适应集群调度》，载于第15届 USENIX 操作系统设计与实现研讨会论文集，2021年。[48] P. Xie, J. K. Kim, Q. Ho, Y. Yu, 和 E. Xing，《Orpheus：通过系统与算法协同实现高效分布式机器学习——

设计》，载于*ACM*云计算研讨会论文集，2018年。[49] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, 和 H. Wu, 《混合精度训练》，载于国际学习表征会议论文集，2018年。

附录 A 更新-撤销算法

算法 3 随机梯度下降

1: **输入：** 学习率序列 $\{\eta_t\}_{t=1}^{T}$；权重衰减 λ >0。2: **初始化：** $x_1 \in \mathbb{R}^d$。3: **对于** $i = 1$ **到** T，**执行**4: $g_t = \nabla f(x_t)$5: $x_{t+1} = x_t - \eta_t(g_t + \lambda x_t)$6: **结束循环**

算法 4 撤销随机梯度下降

1: **输入：** 学习率序列 $\{\eta_t\}_{t=1}^{T}$ $\lambda >$; 0 $x_{t+1} \in \mathbb{R}^d$ $g_t \in \mathbb{R}^d$ 权重衰减; ; 。2: $x_t = (x_{t+1} + \eta_t g_t)/(1 - \eta\lambda)$

算法 5 Adam [33]

1：输入：学习率序列 $\{\eta_t\}_{t=1}^{T}$**；权重衰减** λ **>0；动量估计的指数衰减率** $0 \leq \beta_1, \beta_2 < 1$**；** $\epsilon = 10^{-8}$。**2：初始化：** $x_1 \in \mathbb{R}^d$**；** $m_0 = 0 \in \mathbb{R}^d$**；** $v_0 = 0 \in \mathbb{R}^d$。**3：对于** $i = 1$**到** T **执行 4：** $g_t = \nabla f(x_t)$ **5：** $g'_t = g_t + \lambda \cdot x_t$**6：** $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g'_t$**7：** $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g'^2_t$**8：** $\widehat{m}_t = m_t/(1 - \beta_1^t)$ **9：** $\widehat{v}_t = v_t/(1 - \beta_2^t)$ **10：** $x_t = x_{t-1} - \eta_t \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ **11：结束循环**

—

算法 6 撤销Adam

1: **输入：** 学习率序列 $\{\eta_t\}_{t=1}^{T}$；权重衰减 λ > 0 $0 \leq$; β_1 $\beta_2 < 1$ $\epsilon = 10^{-8}$ $x_{t+1} \in \mathbb{R}^d$ $g_t \in \mathbb{R}^d$ $m_t \in \mathbb{R}^d$ 动量估计的指数衰减率，；；；；；；$v_t \in \mathbb{R}^d$。2: $\widehat{m}_t = m_t/(1 - \beta_1^t)$3: $\widehat{v}_t = v_t/(1 - \beta_2^t)$4: $x_t = x_{t+1} + \eta_t \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$5: $g'_t = g_t + \lambda \cdot x_t$6: $m_{t-1} = (m_t - (1 - \beta_1) \cdot g'_t)/\beta_1$7: $v_{t-1} = (v_t - (1 - \beta_2) \cdot g'^2_t)/\beta_2$

算法 7 AdamW [34]

1：输入：学习率序列 $\{\eta_t\}_{t=1}^{T}$；权重衰减 λ >0；动量估计的指数衰减率 $0 \leq \beta_1$，$\beta_2 < 1$；$\epsilon = 10^{-8}$。2：初始化：$x_1 \in \mathbb{R}^d$；$m_0 = 0 \in \mathbb{R}^d$；$v_0 = 0 \in \mathbb{R}^d$。3：对于 $i = 1$到 T 执行 4：$g_t = \nabla f(x_t)$ 5：$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$6：$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$7：$\widehat{m}_t = m_t/(1 - \beta_1^t)$ 8：$\widehat{v}_t = v_t/(1 - \beta_2^t)$ 9：$x_t = x_{t-1} - \eta_t \cdot (\alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon) + \lambda \cdot x_{t-1})$

—

10: **结束循环**

算法 8 撤销 AdamW

1: **输入：** 学习率序列 $\{\eta_t\}_{t=1}^{T}$；权重衰减 λ > 0 $0 \leq$; β_1 $\beta_2 < 1$ $\epsilon = 10^{-8}$ $x_1 \in \mathbb{R}^d$ $m_0 = 0 \in \mathbb{R}^d$ $v_0 = 0 \in$ 动量估计的指数衰减率，；；；；；$\mathbb{R}^d$。2: $\widehat{m}_t = m_t/(1 - \beta_1^t)$3: $\widehat{v}_t = v_t/(1 - \beta_2^t)$4: $x_{t-1} = (x_t + \eta_t \cdot (\alpha \cdot \widehat{m}_t/\sqrt{\widehat{v}_t} + \epsilon))/(1 - \eta_t \cdot \lambda)$5: $m_{t-1} = (m_t - (1 - \beta_1) \cdot g_t)/\beta_1$6: $v_{t-1} = (v_t - (1 - \beta_2) \cdot g_t^2)/\beta_2$

附录 B 支持多重故障和级联故障

SWIFT 能够容忍多个同时发生的故障。即使发生多个故障，只要仍有丢失模型状态的副本，就可以使用基于复制的恢复。对于基于日志的恢复，如果故障机器上的故障工作节点构成流水线中的连续部分，则可以联合恢复这些工作节点；如果它们分布在流水线的不同部分，则可以分别恢复这些部分。

SWIFT 也能容忍级联故障。如果在从一个故障中恢复时又发生另一个故障，只要丢失状态仍有备份，就可以使用基于复制的恢复。对于基于日志的恢复，在恢复一台故障机器期间，如果另一台承载工作节点的机器发生崩溃，且这些工作节点在流水线中与正在恢复的机器上的工作节点相连，则会中止当前恢复过程，并联合恢复这两台机器；如果新崩溃的机器未承载与正在恢复的机器上的工作节点相连的工作节点，则不会影响当前恢复过程，新故障可以独立恢复。

附录 C 分组详情

我们使用第 7.1 节中描述的模型和设置。我们对每个阶段的 5 次迭代的计算时间进行分析，并取平均值。阶段之间的通信量可以根据模型配置和批次大小直接计算得出。

表 6： BERT-128 的分组结果。结果中的数字表示机器排名。

存储上限（字节）	结果
5.00E+11	[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]]
4.00E+11	[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11, 12], [13, 14], [15]]
3.50E+11	[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9, 10], [11, 12], [13, 14], [15]]
3.00E+11	[[0], [1], [2], [3], [4], [5], [6], [7, 8], [9, 10], [11, 12], [13, 14, 15]]
2.50E+11	[[0], [1], [2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14, 15]]
2.20E+11	[[0], [1, 2], [3, 4], [5, 6], [7, 8], [9, 10, 11, 12], [13, 14, 15]]
1.50E+11	[[0, 1, 2], [3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15]]
1.00E+11	[[0, 1, 2], [3, 4, 5, 6, 7, 8], [9, 10, 11, 12, 13, 14, 15]]
8.00E+10	[[0, 1, 2, 3, 4, 5, 6, 7, 8], [9, 10, 11, 12, 13, 14, 15]]
5.00E+10	[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]

表7: ViT-128/32 的分组结果。结果中的数字表示机器排名 .

存储上限（字节）	结果
1.40E+12	[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]]
1.20E+12	[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11, 12], [13, 14], [15]]
1.10E+12	[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11, 12], [13, 14, 15]]
1.00E+12	[[0], [1], [2], [3], [4], [5], [6], [7, 8], [9, 10], [11, 12], [13, 14, 15]]
9.00E+11	[[0], [1], [2], [3], [4], [5], [6, 7], [8, 9], [10, 11, 12], [13, 14, 15]]
8.00E+11	[[0], [1], [2], [3], [4, 5], [6, 7], [8, 9], [10, 11, 12], [13, 14, 15]]
7.00E+11	[[0], [1], [2], [3], [4, 5], [6, 7], [8, 9], [10, 11, 12, 13, 14, 15]]
6.00E+11	[[0], [1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11, 12, 13, 14, 15]]
5.00E+11	[[0, 1], [2, 3], [4, 5], [6, 7, 8, 9], [10, 11, 12, 13, 14, 15]]
4.00E+11	[[0, 1], [2, 3, 4, 5], [6, 7, 8, 9], [10, 11, 12, 13, 14, 15]]
3.00E+11	[[0, 1], [2, 3, 4, 5], [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
2.00E+11	[[0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
1.00E+11	[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]