计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
62(6): 1380−1395, 2025

Resilio: 一种大模型弹性训练容错系统

李 焱[1] 杨偲乐[1] 刘成春[1] 王林梅[1] 田瑶琳[1] 张信航[1] 朱 昱[2] 李菡蒲[1] 孙 磊[1] 颜深根[2] 肖利民[1] 张伟丰[1]

[1](联想研究院 北京 100094)

[2](纳米集成电路与系统实验室(清华大学) 北京 100084)

(liyan665@gmail.com)

Resilio: An Elastic Fault-tolerant Training System for Large Language Models

Li Yan[1], Yang Sile[1], Liu Chengchun[1], Wang Linmei[1], Tian Yaolin[1], Zhang Xinhang[1], Zhu Yu[2], Li Chunpu[1], Sun Lei[1], Yan Shengen[2], Xiao Limin[1], and Zhang Weifeng[1]

[1](*Lenovo Research*, *Beijing* 100094)

[2](*Nanoscale Integrated Circuits and Systems Lab* (*Tsinghua University*), *Beijing* 100084)

Abstract Large language models with hundreds of billions of parameters are driving rapid technology innovations and business model transformations in artificial intelligence and heterogeneous computing. However, training such models requires prolonged occupation of extensive hardware resources, thus often incurring diverse high-frequency software/hardware failures. These failures not only are challenging to diagnose, but also lead to much longer training time due to unwanted computation waste and slow training convergence. Resilio, an elastic fault-tolerant system for training large language models is proposed, to provide an efficient automated fault recovery mechanism. It is designed to target multiple typical failure scenarios during training processes, such as network interruptions, node crashes, and process failures. Leveraging the characteristics of the parallel model training strategies and underlying hierarchical storage architectures, Resilio implements multi-layer optimizations on checkpoint read/write operations and Just-In-Time (JIT) recovery mechanisms. For models with 100 billion scale parameters, Resilio reduces the end-to-end recovery time under 10 minutes, while reducing the re-started computation after interruptions to the cost of a single training iteration. Upon variations of the computation resources, Resilio can quickly identify the cluster configurations to enable optimal parallel training strategies. Combined with the fault-tolerant scheduling capability, the system ensures adaptive and elastic resource allocations to greatly improve training efficiency and boost GPU utilization across large-scale computing clusters.

Key words large-scale model training; deep learning; fault tolerance; failure detection; elastic training; automatic parallelization

摘 要 具备千亿级参数的大型语言模型正在引领当今人工智能与异构计算的技术革新及商业模式的深刻转变. 然而, 大模型训练任务需要长时间占用大量的硬件资源, 软硬件故障发生的频率高且类型较多, 并且故障原因难定位导致训练中断时间较长. 针对大模型训练过程中面临的网络中断、节点宕机、进程崩溃等多种典型故障, 提出一种大模型弹性容错系统 Resilio 来提供高效自动的恢复机制. 基于模型训练的并行策略与硬件的存储层次特点, Resilio 通过多层次优化检查点读写操作和即时检查点保存机制, 对于

收稿日期: 2025-03-01; **修回日期**: 2025-04-10

基金项目: 国家重点研发计划项目(2024YFB4505703)

This work was supported by the National Key Research and Development Program of China (2024YFB4505703).

通信作者: 肖利民(xiaolm@lenovo.com)

千亿规模参数模型，可以将端到端故障恢复时间缩短至 10 min 以内，模型中断后的重新训练时间缩短至单次训练迭代时间. 当集群资源弹性变化时，Resilio 能够快速准确地获取大模型训练最优并行策略配置，与容错调度组件共同确保系统的自适应能力，弹性调度训练资源用以提升作业的训练效率和集群 GPU 资源利用率.

关键词 大模型训练；深度学习；容错；故障检测；弹性训练；自动并行

中图法分类号 TP319

DOI: 10.7544/issn1000-1239.202550147 **CSTR:** 32373.14.issn1000-1239.202550147

近年来，深度学习模型在自然语言处理、计算机视觉和推荐系统等领域取得了显著进展，以 Transformer 结构[1] 为基础的模型参数量已达到数百亿甚至上千亿，训练这些模型需要数千个 GPU 并行工作数周甚至数月. 在这种大规模分布式训练环境中，硬件故障和系统中断的概率显著增加，而网络基础设施的带宽限制和容错能力不足进一步加剧了训练的不稳定性[2]. Meta 使用千卡 A100 训练 OPT-175B 模型耗费了 2 个月，共发生了 105 次重启，最长健康持续训练时间为 2.8 天[3]. 使用万卡集群训练 Llama3-70B 过程中共发生 419 次任务中断，其中 GPU 故障占比 58.7%[4].

在训练过程中一般周期性的保存模型训练的状态，简称检查点（checkpoint, CKPT），CKPT 除了包含模型参数（权重和偏置），还涵括优化器状态（动量和学习率等）、训练的迭代次数和数据迭代标签等信息，在大规模分布式训练中，CKPT 的写入和读取操作往往成为训练过程的瓶颈. 对于 GPT3-175B 大模型[5]，CKPT 容量达到了惊人的 2.4 TB，此外，高频的 CKPT 操作会加剧训练任务的阻塞时间和带宽占用，低频 CKPT 会导致重训练耗时较高. 当集群资源弹性变化时，即出现资源不足或者资源扩增时，如何维持现有的训练状态，容错训练系统需要具备动态调整资源分配、并行策略的能力.

软硬件故障的多样性、随机性以及硬件资源的弹性变化，对如何构建一套高效准确的自动化故障恢复系统带来了巨大的挑战，主要包括 3 个方面：

1）准确的故障感知和有效的作业恢复成为巨大挑战. 故障涉及海量硬件器件和软件栈，故障暴露出来的日志和指标数据庞杂，如何从中检测到故障的发生，需要在开销与准确性间取得良好的平衡. 作业恢复机制需要有效地规避故障源，避免无效的恢复行为加剧作业中断时长，同时尽可能缩短恢复阶段的耗时.

2）尽管 CKPT 技术对容错至关重要，但面临两大挑战. 首先，I/O 性能瓶颈导致大规模 CKPT 写入速度远低于硬件上限，延长了保存时间. 其次，固定频率的 CKPT 保存策略在故障发生时会导致大量训练进度丢失，尤其在大规模分布式训练中，这可能浪费数小时到数天的训练时间，显著降低资源利用率. 这些挑战凸显了优化 CKPT 机制的重要性.

3）现有的弹性训练系统通常仅支持固定资源配置，缺乏复杂场景下的自动恢复机制. 在采用多种并行策略的大型模型中，初始资源配置确定后，难以灵活调整资源数量与分布. 尤其在节点失效或需临时缩减资源时，现有弹性系统通常无法自动处理任务和资源重分配. 这导致动态环境中资源管理效率低下和故障恢复能力不足，影响系统的容错能力和资源利用效率.

现有的深度学习框架 PyTorch[6]，TensorFlow[7]，DeepSpeed[8] 和 Megatron-LM[9] 等提供了基本的 CKPT 功能，当训练任务发生故障后，找准根因并手动恢复训练一般耗费数小时甚至数天. 在使用传统存储设备（如 HDD 或未优化的 SSD）时，I/O 性能的不足会严重影响 CKPT 读写操作和降低训练效率.

已有的大模型容错训练系统，如 TRANSOM[10]，DLRover[11-12] 等方案提供了 GPU、网络、内存等硬件故障检测与自动恢复机制，但在资源弹性伸缩时，并行策略无法调整或者只支持有限的数据并行（data parallel, DP）维度，此外，基于集群多级存储层次架构的异步保存与恢复机制[13-16]，虽然能最小化 CKPT 操作的延迟，但仍面临诸多挑战，特别是在存储瓶颈、恢复速度以及动态资源分配的复杂性上，难以实现真正的高效和无缝弹性扩展.

本文设计了一种面向大规模训练作业的弹性容错系统，如图 1 所示，该系统从功能上可以划分为异常作业高效管理（容错框架）、检查点高效存取和自适应并行策略调整 3 部分，能够将千亿参数模型端到端恢复时间从数小时缩短至 10 min 以内.

容错框架是整个容错系统的枢纽，负责感知故障的发生以及协调整个恢复流程. 恢复流程包括故障资源的检测及驱离、训练作业的重新部署、作业部署后的初始化环节. 在作业重新部署阶段，容错框

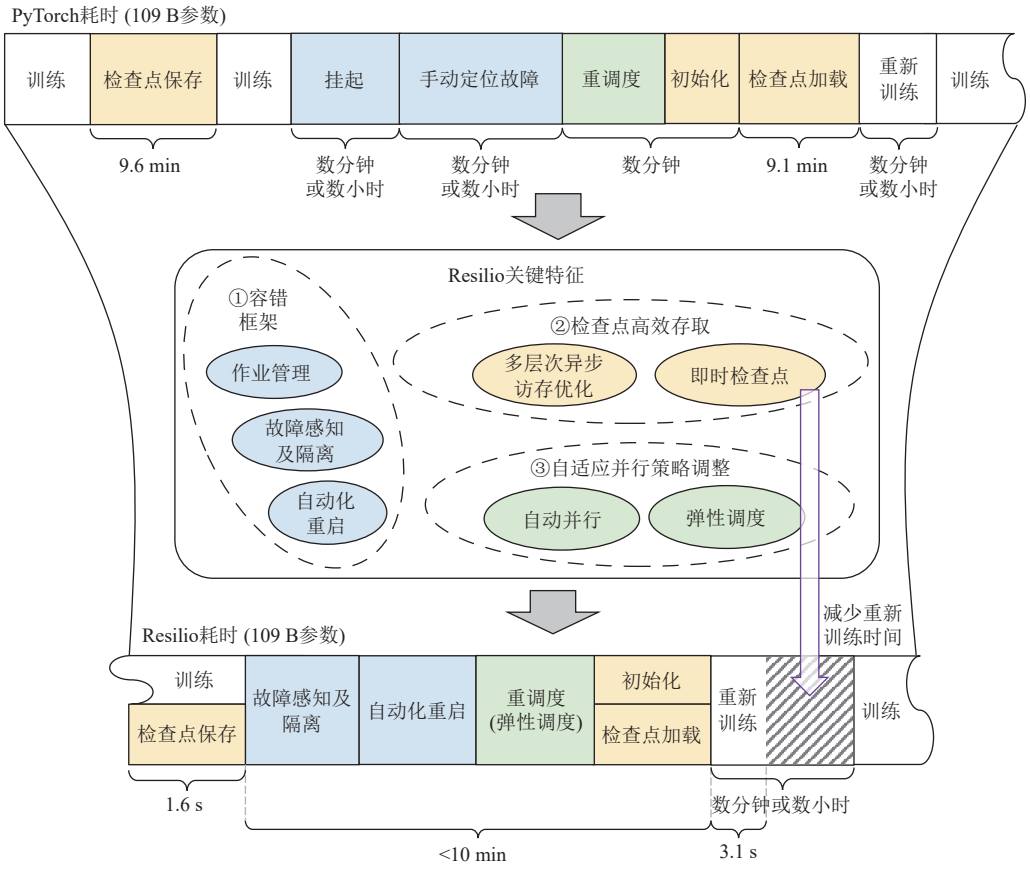

Fig. 1 Performance optimization decomposition of model fault recovery time for 109 billion parameters

图 1 1 090 亿参数模型故障恢复时间性能优化分解

架首先通过自动化并行策略模块生成匹配当前资源最优的并行策略，保证作业恢复之后拥有较好的训练性能，然后显式触发 CKPT，持久化当前作业的训练状态. 在作业初始化阶段，进程通过调用优化后的 CKPT 库提升 CKPT 加载速度，并通过即时检查点(just-in-time checkpoint, JIT CKPT)机制[15]缩短作业恢复后的重新训练时间. 本文的主要贡献有 3 点：

1)提出一种准确高效的容错框架. 该框架能够在典型故障发生的第一时间，准确定位故障原因，并且根据故障类型对作业进行层次化恢复，缩短故障导致的作业中断时长. 同时框架具有良好的可扩展性，能够兼容主流的大模型训练框架，如 Megatron-LM、DeepSpeed 等.

2)提出一种多层次异步 CKPT 优化方案，通过共享内存和本地磁盘缓存显著降低了故障恢复和加载开销. 此外，引入即时检查点机制，在故障发生时即时保存完整的 CKPT，减少了重新训练时间，并通过自适应超时设置优化了通信操作的故障识别与恢复，提升了大模型训练的效率与稳定性.

3)提出一种低开销的自动并行搜索系统，旨在为指定硬件资源下的大规模深度学习模型训练提供较优并行策略. 该系统能够在资源类型或数量发生变更时，迅速调整并搜索到最适合新资源配置的并行度，从而确保训练任务的高效性和稳定性.

1 相关工作

1.1 大模型容错训练

针对大模型训练的容错框架是指一套独立于训练作业的软件，通常需要具备作业异常感知、软硬件故障定位以及训练作业自动化恢复等核心功能组件，当感知到作业出现异常后，迅速定位到造成该异常的软硬件故障原因，并针对性恢复作业. 在故障定位方面，通常的做法是采集多维度检测指标，这些指标既包括 GPU、网络、内存等硬件设备的工作状态指标，还包含训练脚本、NCCL、CUDA 库等软件运行状态指标[17-19]，然后运用一些数据分析方法发现指标数据中所包含的故障模式. 比如 Wu 等人[10]重点关注故障分析的可解释性和效率，采用离群检测、聚类等传统算法分析指标数据，这种方法能检测到异常情况的发生，但是却无法准确地定位导致故障的根因. Hu 等人[20]则引入专门的 LLM，利用 LLM 的强大的文本

理解和分析能力，准确定位故障根因，但受限于LLM的输入长度限制，无法处理海量指标数据. 在作业恢复方面，通常的做法是重启整个作业，然后基于CKPT机制保存的作业快照数据恢复作业内各进程的状态，这种全局性的重启往往带来较长的作业中断时间. 因此业界有一些优化工作，蚂蚁集团提出的DLRover框架，支持GPU掉卡、NCCL超时等有限的故障类型. Wu等人[21]针对训练变慢类故障，提出了一种自适应多级缓解机制，通过调整微批次分布和并行化拓扑结构实现对训练作业的部分结构进行调整，避免作业整体重启，但这种方法在应对超大规模以及复杂通信拓扑时，方法的恢复效率以及有效性将大打折扣. Lao等人[22]在发生异常后，通过冻结机器、初始化备用机器、恢复状态、替换通信组成员以实现快速局部恢复，无需重启训练任务，显著减少恢复时间和开销，但该方法更适合处理运维场景下的作业迁移，对于突发的软硬件故障，往往无法完整地保留作业的状态，在一些极端情况下，这种局部重启的方法会退化成全局重启.

在大模型训练过程中，CKPT机制被广泛用于保存和快速恢复训练状态. 然而，超大容量CKPT的读写操作往往带来显著开销，甚至可能严重阻塞整个训练进程，导致GPU资源利用率低下. 为解决这一问题，现有研究提出了多种优化方案. Mohan等人[13]通过调整CKPT的频率来平衡开销与恢复效率，但这种方式本质上是一种折中解决方案，无法同时兼顾保存频率和运行时开销；Wang等人[14]利用NVMe和数据并行写入技术提升CKPT性能，尽管有效，但这种方法对高性能NVMe存储设备的高度依赖增加了系统成本和复杂性；Jiang等人[23]通过在数据并行组之间共享GPU工作者的数据缓解了存储瓶颈，减少了恢复期间的检索时间，虽然有助于减轻分布式文件系统的负载，但在实践中仍存在较长的训练暂停时间. 此外，Eisenman等人[24]通过量化推荐模型中的嵌入表以减少CKPT大小，然而量化过程本身可能引入误差，影响模型的最终表现，尤其在需要高度精确的应用场景下，这种权衡显得尤为关键. Wang等人[16]引入内存CKPT机制，即利用高带宽CPU内存进行CKPT创建，并将CKPT分片分布到对等节点上以最大化故障恢复的可能性，从而避免训练过程中的停滞，尽管提高了容错能力，但该方法也面临着网络通信潜在瓶颈的问题，并且需要复杂的调度算法确保节点间负载均衡和高效协作. Gupta等人[15]则利用大规模工作负载数据并行副本中的状态冗余实现高效的运行时CKPT，但该方法未充分考虑CKPT一致性及保存过程中出现的故障处理问题. DLRover框架提供了检查点优化组件Flash Checkpoint，通过异步持久化内存热加载大幅提升了CKPT的保存和加载速度，然而在节点故障的情况下，由于内存缓存容易出现不命中现象，该方法的有效性会受到影响.

综上所述，尽管上述研究各自提供了一定程度上的解决方案，但它们在实际应用中也面临着从软硬件依赖、故障种类，到算法的复杂性与效率等方面的挑战. 因此需要全面考虑这些限制因素，以开发出更加高效和可靠的大模型容错训练系统.

1.2 大模型弹性训练

大模型弹性训练指的是在深度学习中，特别是针对Transformer架构的大模型，通过动态调整计算资源来优化训练过程的技术. DLRover通过预扩展、扩展和后扩展三阶段实现弹性训练. 三阶段的描述为：根据历史任务获取初始资源分配，在线拟合生成多个候选计划，并采用加权贪心算法确定最终执行计划，同时结合实时负载实现动态负载分片. DLRover支持采用数据并行策略的训练作业的弹性伸缩，当作业中某些数据并行实例出现故障后作业仍能在小规模下继续运行，当有新的健康资源可用后，能重新扩展至原有规模运行，但不支持复杂并行训练策略下的弹性恢复. 字节的MegaScale针对云场景进行了优化，默认集群节点资源充足. 如果某个节点因故障被移除，它能够迅速将受影响的任务pod（Kubernetes资源编排系统中能够创建和部署的最小单元）重新部署到健康的节点上，保证训练任务的持续进行，但该方案无法对单个任务的资源配置进行弹性伸缩. Li等人[25]提出的EasyScale主要为了解决在资源弹性下保持模型一致性准确度的问题. 它引入了EasyScaleThread（EST）抽象，将分布式模型训练过程与硬件资源分配解耦. 多个EST可以动态地在GPU之间进行上下文切换，从而在资源弹性下保持训练行为的一致性. 但是EasyScale目前主要聚焦在数据并行，对大模型训练场景下其他并行策略暂不支持. Subramanya等人[26]提出的Sia旨在为异构深度学习集群资源分配高效的调度方案. 该系统采用一种新的在线学习方法为每个作业在每种GPU类型上建立吞吐模型；引入了一个新的整数线性规划算法来处理作业规模和异构性，该系统在每个调度轮次都会考虑所有的GPU分配（数量和类型），估计其吞吐量并选择最佳的资源分配方案. Wagenländer等人[27]提出了一种用于深度学习框架的状态管理库tenplex，

能够在训练过程中动态改变 GPU 分配和作业并行性. 通过将作业表示为可并行化的张量集合，可以快速生成新配置. 但 tenplex 主要聚焦在数据并行，而目前典型的大模型训练采用多种并行策略. 总之，为了应对大模型训练任务中更复杂的集群配置和工作负载，亟需一种能弹性适应集群资源变化和动态选择最优模型训练配置的优化技术.

2 大模型训练容错系统

本文提出的大模型弹性训练容错系统 Resilio 如图 2 所示，其涵盖容错框架、多层次高效检查点以及自动并行 3 个部分，相关内容将在本节中逐一展开.

2.1 容错框架

容错框架主要提供自动化故障恢复系统，它是一套独立于训练作业的运行时系统，主要负责感知系统内故障的发生，并且根据不同故障类型触发相应的自动化运维操作. 该框架主要包含故障感知模块、生命周期管理模块、CKPT 控制模块、日志及指标上报模块，各模块的功能如下：

1)故障感知模块. 主要负责分析指标采集模块获取的设备及任务指标数据，判断故障是否发生. 该模块由一个控制节点和若干分散在不同计算节点内的守护进程组成，每个守护进程与训练进程绑定，通过拦截进程执行的核心代码，来判断训练进程的健康状态. 遵循 CUDA 和 NCCL 库中核心计算和通信函数接口，本文设计了一套轻量级拦截库，通过 LD_PRELOAD 机制预先加载. 拦截到函数调用之后，在函数的执行前后设置 CUDA Event 用于统计核心计算和通信函数的耗时. 拦截库将函数耗时指标持续写入共享内存中，故障检测守护进程持续读取共享内存中的指标数据. 由于健康的训练作业呈现周期性特征，当故障检测守护进程发现指标频率和指标数值出现大幅度的变化时，判定当前训练进程存在异常. 健康状态的判定主要分为 2 个层次：Error 和 Slowdown.

当守护进程发现训练进程抛出致命异常时，会将该进程的状态设置为 Error 状态，并上报给控制节点，控制节点会通过生命周期管理器停止所有训练进程，并触发故障排查流程，该流程会检测各个 pod 内软件栈的适配性、硬件设备可用性、数据存储路径的访问权限等内容，然后结合捕获的异常日志，据此判断故障的根因. 由于分布式应用发生的异常存在传播性，即单个节点的异常会导致依赖它的节点出现异常. 因此工作节点在捕获到异常日志后会上报到控制节点，控制节点首先比对异常日志的时间戳来判断最先出现异常的节点，然后向该节点发送诊断指令，节点响应指令开始检测 GPU、网卡、存储设备的状态，最终将检测结果连同异常日志时间戳前后分钟内的系统日志、kubelet 日志、pod 日志返回

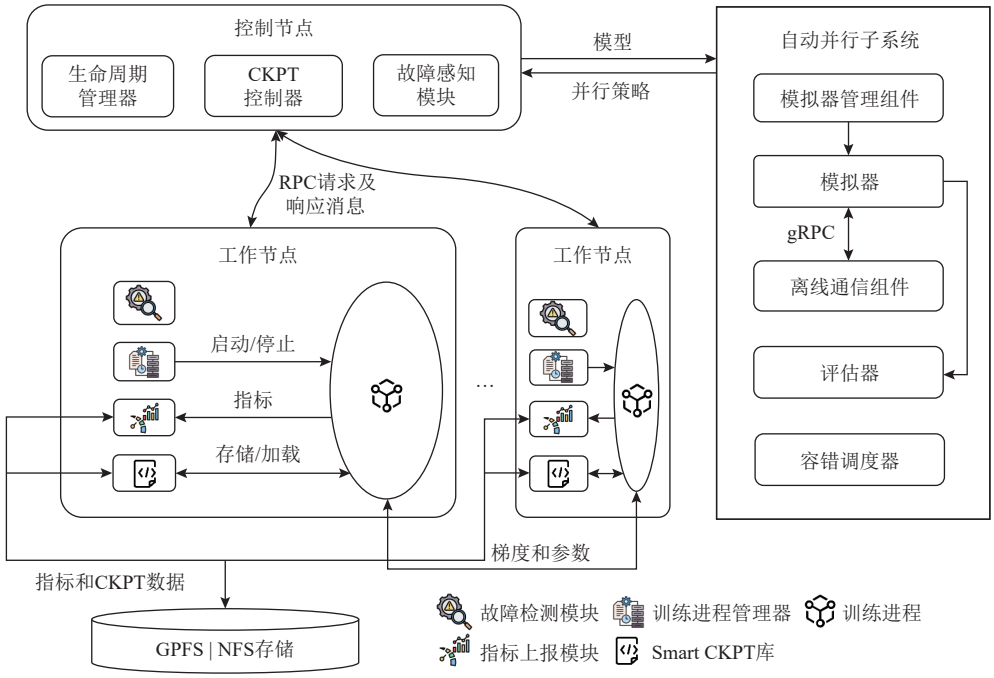

Fig. 2 Overall architecture of Resilio

图 2 Resilio 整体架构

给控制节点.

守护进程通过拦截训练进程触发的计算和通信函数调用来统计这些调用的频次和耗时数据. 通过分析这些时序指标的变化来挖掘训练变慢时的模式, 当守护进程感知到训练进程变慢后, 会将该进程的状态置为 Slowdown 状态, 并上报给控制节点, 控制节点会通过生命周期管理器暂停所有训练进程, 同样触发故障排查流程. 与 Error 状态下检测流程不同的是, Slowdown 类型故障的检测流程可以并行地在不同节点上运行小规模分布式矩阵相乘作业, 通过比对不同节点集内作业的完成时间、集合通信带宽等信息, 确定导致作业变慢的节点集.

2)生命周期管理器. 主要负责响应作业提交命令, 完成作业所需资源的调度、环境可用性校验、分布式进程创建、故障发生后的层次化恢复等.

3)CKPT 控制器. 主要负责显式触发 JIT CKPT 操作以及预加载 CKPT 数据. 故障感知模块在获知集群内故障的信息之后, 会根据故障的严重程度进行不同级别的运维处理, 而对于一些仅影响作业运行性能或者判定健康进程可以完成一次完整的 CKPT 保存时, 会暂停作业的执行并显式触发 CKPT, 这种方式保证了通过 CKPT 机制持久化的作业快照版本与故障发生时的作业版本具有较小的差异; 同时在作业重启恢复过程中, 各个训练进程在加载 CKPT 快照数据之前会执行一些其他的初始化, 比如集合通信组构建. 本模块会根据各训练进程信息, 提前从持久化存储中将必要的 CKPT 数据加载到本地磁盘之中, 隐藏部分数据传输的耗时.

4)指标上报模块. 主要负责采集训练进程执行过程中的指标和日志数据. 这些数据既包括集群内计算节点上 CPU、内存、磁盘、GPU、NPU 等硬件设备的工作状态数据, 也包括训练任务执行 CUDA Kernel、NCCL 通信等操作的运行状态数据. 通过定义一套统一的指标采集规范, 确保对于不同厂商设备按照规范采集相应类型的指标后, 能够准确判断具体设备的健康状态.

2.2 检查点高效存储

2.2.1 多层次异步访存优化

如图 3 所示, 训练过程中的一次迭代分为前向(forward, FW)、反向(backward, BW)、优化器(optimizer, O)权重更新三个部分, AI 训练框架中默认的 CKPT 保存操作发生在模型权重更新之后, 一次 CKPT 保存(checkpoint save, C)包括从 GPU 显存拷贝数据到 CPU 内存, 然后从 CPU 内存拷贝数据到本地磁盘以及远程共享存储, 最后进行全局同步保证分布式 CKPT 版本一致. 这种 CKPT 保存过程主要面临

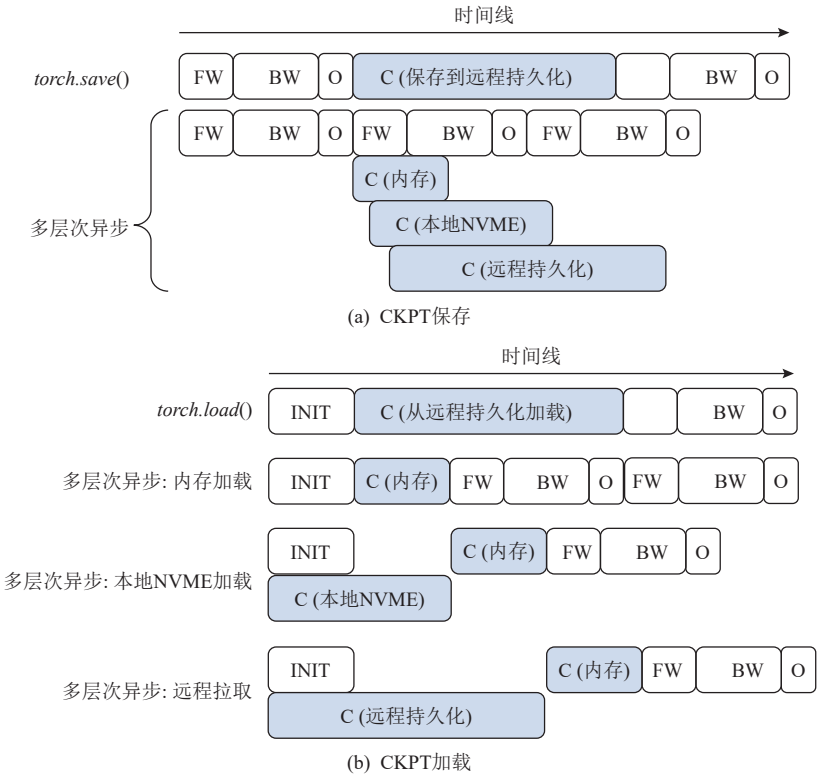

Fig. 3 Multi-layer asynchronous CKPT save and load

图 3 多层次异步 CKPT 保存和加载

3 个挑战: 1)CKPT 由数据庞大的张量组成, 传统的序列化方法开销大; 2)GPU 显存到 CPU 内存的数据传输过程未充分利用带宽; 3)CKPT 保存过程及最后的同步过程均会阻塞训练, 造成大量的 GPU 计算资源闲置.

本文提出的多层次异步 CKPT 方案针对性地做了 3 点优化:

1)解析 CKPT 元信息(张量形状、大小以及所在网络层等数据), 只对元信息进行序列化, 大幅减少全量序列化带来的开销;

2)使用锁页内存及 CUDA 多流机制充分利用 GPU 显存到 CPU 内存的传输带宽;

3)由于前向和反向计算不会修改 CKPT 内容, 所以可以重叠上一轮迭代 CKPT 保持操作与本轮迭代的前反向计算, 同时将 CKPT 数据切分成多个小块异步传输到本地 NVME 和远程持久化存储, 全局同步过程在传输线程中完成, 避免阻塞训练进程.

在分层存储架构中, 分块大小与存储介质的物理特性及访问模式强相关, 在异步传输至本地 NVME 的过程中, 以 4 KB 作为块大小精准匹配 NVME 页大小, 最大化写入效率. 远程持久化存储时, 数据的切片大小设置为 4 MB, 因为节点间网络传输通常基于 RDMA 或 TCP/IP 协议, 较大的分块可以减少网络协议头占比. 远程持久化存储作为多级存储的最后一级, 包含本地存储所有的 CKPT 历史数据, 由于本地存储空间有限, 通过文件覆盖方式, 可配置仅在本地保存最近一段时间的 CKPT 数据.

CKPT 恢复发生在训练开始前, 原生的 PyTorch 会在初始化结束后, 从远程持久化存储中读取最新的 CKPT 版本, AI 框架自带的 CKPT 加载过程主要面临的挑战是 CKPT 全部从远程存储拉取, 对存储服务的负载压力非常大, 很容易达到存储带宽瓶颈, 造成加载时间过长以及大量 GPU 资源闲置. 本文提出的多层次异步 CKPT 在这方面做了 3 点优化:

1)使用共享内存作为一级缓存, 针对无需重启节点的故障, 可以直接从 CPU 内存读取 CKPT, 从而大幅缩短 CKPT 加载时间;

2)使用本地磁盘作为二级缓存, 针对需要替换节点的故障时, 除了新节点需要从远程持久化存储拉取 CKPT, 其他进程能够直接从本地磁盘读取 CKPT, 大幅减少对远程存储的带宽依赖和竞争;

3)此外, 在模型初始化阶段, 同时检查内存中是否存在 CKPT 副本, 不存在时会从本地 NVMe 或者远程持久化存储加载 CKPT 到内存, 并且与初始化阶段重叠以隐藏 CKPT 加载耗时.

2.2.2 即时检查点

典型的大模型训练通常采用 3D 并行策略, 包括数据并行(data parallel, DP)、张量并行(tensor parallel, TP)和流水线并行(pipeline parallel, PP). 数据并行通过将训练数据集分割成不同批次大小(batch size, BS), 每个批次分配到不同的计算设备, 每个设备都持有一份完整的模型副本, 独立进行前向和反向传播计算, 然后通过 AllReduce 集合通信操作同步梯度并更新模型参数; 张量并行则将模型的张量(如权重和梯度)分散到多个设备上进行计算, 通信量要求高并且频繁, 一般适用于节点内单机多卡; 流水线并行将模型的不同层分配到不同的设备上, 每个设备负责一部分计算任务, 数据在设备之间按顺序传递, 通过点对点通信, 通信带宽要求相对较低. 大模型训练的整个过程存在 3 个特征: 1)大模型训练 3D 并行策略中数据并行维度进程的模型副本一致, 模型参数更新前需要全局同步; 2)训练故障基本为单点故障, 很少有大规模的节点故障; 3)某个进程发生故障后会导致其他健康进程在某一次集合通信操作中超时.

结合大模型训练并行策略的特点以及故障发生的规模, 我们设计了 JIT CKPT, 能够在故障发生时立即保存最近完整的 CKPT, 从而大幅度降低了常规周期性保存 CKPT 带来的重新训练时间, 提升模型有效训练时间. 图 4 展示了 JIT CKPT 的工作流程.

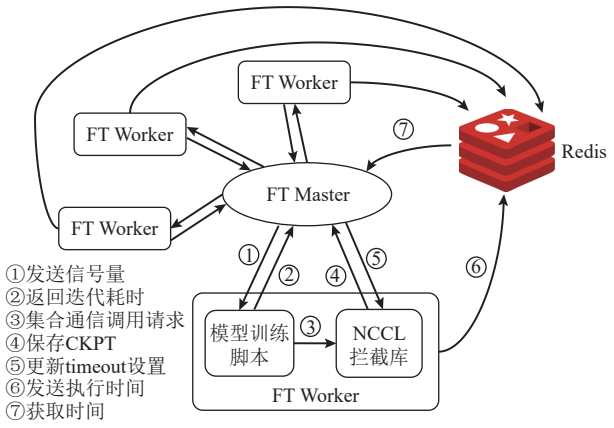

Fig. 4 The workflow of JIT CKPT

图 4 JIT CKPT 工作流程

1)容错工作节点(fault-tolerance worker, FT Worker)内部通过拦截 NCCL 集合通信函数库, 获取训练进程所需要的通信算子, 并统计通信算子所消耗的时间, 实时发送给 Redis 数据库进行保存.

2)容错主节点(fault-tolerance master, FT Master)周期性地从 Redis 处取出通信算子的耗时信息, 自适

应修改超时配置并及时同步更新到 FT Worker.

3)当某个进程或节点故障导致当前进程触发超时后，该 FT Worker 发送超时事件给 FT Master.

4)FT Master 收集到所有超时事件后，推演故障进程并判断当前健康进程是否具备保存完整 CKPT 的条件. 具体而言，FT Master 能够判断出发送超时事件的进程为健康进程，未发送超时事件的进程为卡死故障进程，根据故障进程所在数据并行组(组中 CKPT 互为副本)中是否有可用的健康进程进而判断是否具备保存完整 CKPT 的条件. 若是，则通知 FT Worker 进行 CKPT 保存；否则，结束所有 FT Worker.

2.3 自适应并行策略调整

在容错系统中，自动并行技术对提升系统的可靠性和训练效率不可或缺，当节点出现故障时，集群可能出现无法提供空闲健康节点作为替代的情况，从而导致任务所需资源数量或类型的变更. 为了应对这种风险，在容错系统中引入自动并行机制能快速地将任务重新部署到可用的集群资源上，确保计算任务的连续性不受影响. 这种灵活性不仅增强了系统的容错能力，还能有效地提升资源利用率. 其次，自动并行技术能通过优化负载均衡策略，进一步提高训练效率并缩短模型的训练周期，这对于大规模深度学习任务尤为重要. 此外，在大规模分布式深度学习场景中系统的自动并行化还能大大降低手动配置复杂分布式环境的需求，从而简化运维流程以及减少人为错误的引入.

因此，本节设计了低开销高精度的层级式自动并行子系统 HPPS(hierarchical parallel partitioning sub system). 如图 5 所示，该子系统由自动并行主体系统和基础架构层 2 个关键组件组成.

1)自动并行主体系统. 该系统由模拟器管理组件、模拟器、离线通信组件和评估器组成. 模拟器管理组件负责对并行搜索空间剪枝，模拟器是 HPPS 最核心的组件，可在指定硬件资源和并行度下得到训练任务的单次迭代时间. 模拟器通过编译模型计算图，得到模型的计算成本、模型流水线并行相邻层之间需要传输的字节数，并利用端到端建模算法(综合考虑计算和通信)，全面评估单次迭代的时间开销和资源占用. 离线通信组件利用 NCCL-tests[28] 采集集群不同数据量下的通信开销，量化数据传输的时间和资源消耗. 评估器根据集群空闲资源和模拟器输出的结果自动进行性能优先级排序、资源匹配过滤、多目标优化等流程，评估得到当前最优的硬件配置和并行策略.

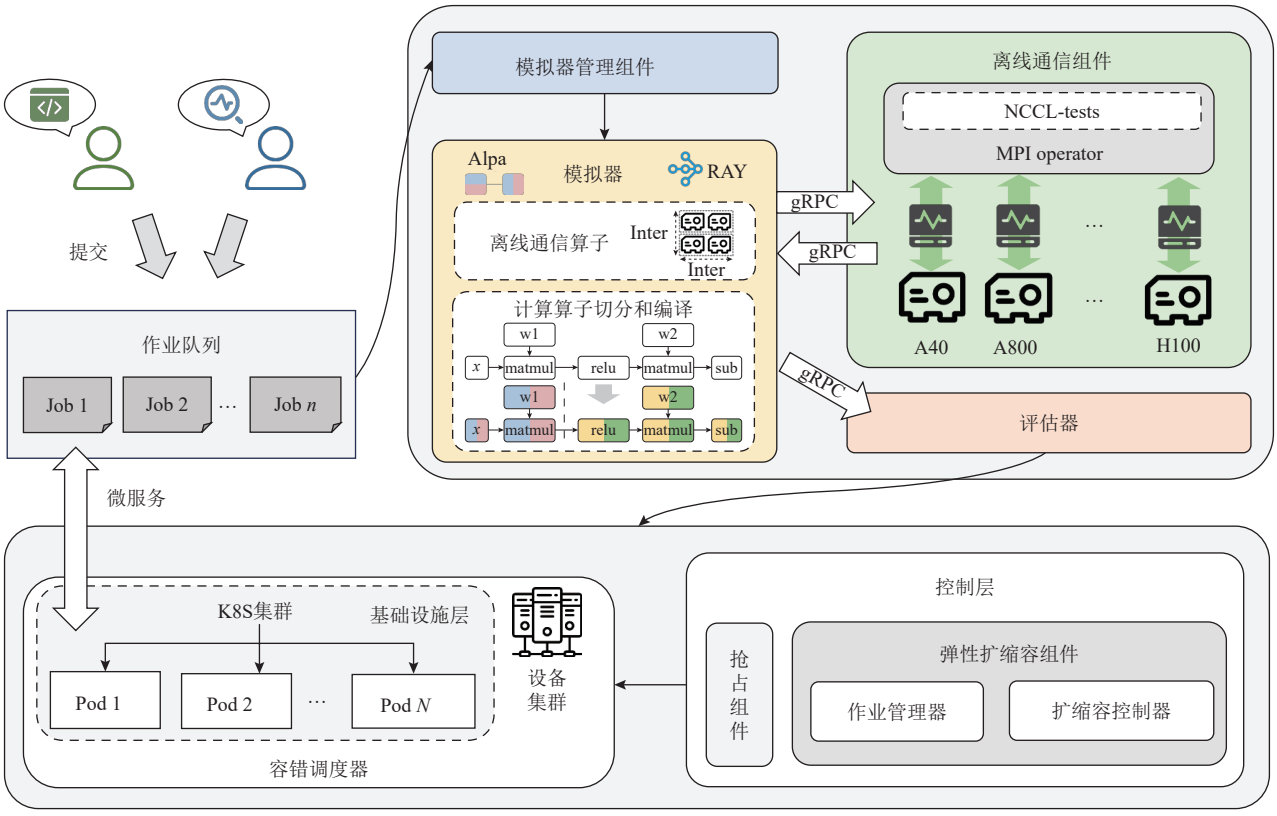

Fig. 5 Workflow diagram of auto-parallel submodule

图 5 自动并行子模块工作流程图

2)基础架构层. 由容错调度器和弹性扩缩容组件组成. 容错调度器提供具有容错调度功能的调度策略, 具体而言, 调度器需要对容错系统标记的故障节点进行反亲和性调度. 除具有容错调度功能外, 容错调度器还具有多租户隔离、资源超发、任务抢占等特性. 弹性扩缩容组件负责在深度学习数据并行训练过程中根据集群负载自动调整数据并行度. 通过与自动并行的模拟器管理组件紧密协作, 该模块可在保证训练过程连续性的同时, 高效地利用可用资源, 从而降低空闲浪费并提升整体吞吐率. 相较于传统手动配置方式, 弹性扩缩容组件不仅简化了集群管理流程, 也显著提升了大规模分布式训练的可伸缩性与灵活性.

HPPS 的工作流程和各组件间的数据传输关系如图 5 所示. 模拟器管理组件接受用户提交的训练作业, 为每种组合方式启动一个模拟器实例. 模拟器管理组件首先根据用户申请的资源量N生成$device_mesh=\{(n,m)\}$. 其中, n为节点数, m为每节点的 GPU 卡数, 且$N==n\times m$. 随后, 对$device_mesh$进行优先级排序并根据集群资源进行资源过滤. 然后, 按照“节点内优先数据并行和张量并行, 节点间优先流水线并行”的策略, 枚举模型训练的并行度.

模拟器实例接收由模拟器管理组件传入的 GPU 资源类型、资源个数和流水线并行度, 自动搜寻当前配置下的最优自动并行策略并评估模型的端到端时延. 具体而言, 可以划分为 3 个步骤.

1)模型计算图编译和流水线阶段聚合. 基于 CPU 编译模型生成静态计算图, 对计算中间表示(intermediate representation, IR)进行分析, 得到算子的每秒浮点运算次数(floating point operations per second, FLOPs)和算子间通信字节数. 其中, 使用动态规划算法对计算图按照指定流水线并行度进行切分. 流水线切分原则为: 尽量保证每个流水线阶段 FLOPs 相同的情况下最小化流水线阶段之间的通信量.

定义 1. 流水线切分算法.

$$\min \max_{s=1,\cdots,S} c_s(1-\partial)v \leqslant f_s \leqslant (1+\partial)v, \tag{1}$$

其中S代表流水线并行度; c_s代表第s个流水线阶段的通信量, 即它从前面的流水线阶段获取的总字节数; ∂为超参, 表示流水线阶段 FLOPs 的上下界; f_s代表第s个流水线阶段的 FLOPs, 即属于当前阶段的算子的 FLOPs 之和.

定义 2. 各流水线阶段的平均 FLOPs.

$$v=\frac{m}{S}, \tag{2}$$

其中, v代表每个流水线阶段的平均 FLOPs, m是通过$jax.make_jaxpr$函数获得. 若多个切分的最大通信量相同, 则算法更倾向于返回各个流水线阶段 FLOPs 极差更小的解.

2)计算资源评估和并行策略搜索. 完成对模型的指定并行切分后, 模拟器会生成每个流水线阶段的计算图. 对切分后的每个流水线阶段进行独立编译, 得到流水线阶段的计算时间、算子间通信字节数以及峰值显存. 当出现计算时间为无限大或者峰值显存大于 GPU 物理显存的情况时, 表示该种并行方式无法在当前的资源配置下正确执行, 此时模拟器会对不满足情况的并行方式进行剪枝. 对于满足资源的自动并行策略则通过调用离线通信组件接口获得通信时间. 本文采用算法 1 完成并行策略的搜索.

算法 1. 并行策略搜索算法.

输入: 用户提交任务申请的 GPU 卡数M, 集群 GPU 信息T, 模型参数大小S;

输出: 候选的并行度矩阵$\boldsymbol{L}$.

① 初始化$\boldsymbol{L}\leftarrow\varnothing$, $i=0$, $j=0$;

② 获取作业的资源维度: $\boldsymbol{R}\leftarrow(M,T)$, 其中$\boldsymbol{R}=\boldsymbol{r}_i(\boldsymbol{r}_1,\boldsymbol{r}_2,\cdots,\boldsymbol{r}_n)$, $\boldsymbol{R}$为有序矩阵, $r_i=(m,n)$, $m\times n==M$, m为节点数, n为节点内 GPU 卡数, n越大越优先;

③ 根据集群资源进行过滤, 去除资源维度无法满足的情况, $\boldsymbol{R}'\leftarrow filter\_one(\boldsymbol{R},T)$;

④ if $len(\boldsymbol{R}')>1$ then

⑤ $\quad\boldsymbol{R}''=filter\_two(\boldsymbol{R}')$;

⑥ else

⑦ $\quad\boldsymbol{R}''=\boldsymbol{R}'$;

⑧ end if

⑨ for $\boldsymbol{r}_i$ in $\boldsymbol{R}''$

⑩ $\quad pp_j=m_i$;

⑪ $\quad(dp_j,tp_j)=fun(pp_j,S)$;

⑫ $\quad\boldsymbol{L}.append(pp_j,dp_j,tp_j)$;

⑬ end for

⑭ return $\boldsymbol{L}$.

算法 1 中, $\boldsymbol{L}$表示算法返回的并行度矩阵, $\boldsymbol{R}$表示硬件维度的有序矩阵, 其中第 1 维度表示节点个数, 第 2 维度表示节点内 GPU 个数, 按照第 2 维度的大小进行排序; $\boldsymbol{R}''$为经过 2 次剪枝后的矩阵; 函数fun根据模型大小和流水线并行度, 选择合适的数据并行度和流水线并行度.

3)综合计算和通信开销, 计算端到端时延. 在调用离线通信组件接口获得通信时间后, 按照式(3)计

算得到当前训练任务单次迭代的时间.

定义 3. 训练任务单次迭代时间.

$$t=\sum p_i+\sum c_{(i,i+1)}+(B-1)\times\max(p_i), \quad (3)$$

其中 p_i 为第 i 个流水线阶段的计算时间，$c_{(i,i+1)}$ 为第 i 和第 i+1 个流水线阶段之间的通信时间，B 为微批次大小.

随后，模拟器将评估得到的端到端时延和自动并行策略输入至评估器组件. 评估器组件会在接收到指定训练的所有模拟器输出结果后，根据训练迭代时间进行排序. 评估器将选择性能较优的几个候选项进行调度，并按照最优配置进行调度. 如果集群空闲资源不足，则顺延尝试调度次优选项，直到调度成功或者所有候选项都调度失败为止. 此时，调度工作由容错调度器实现. 在容错系统中检测到故障节点后，容错调度器将故障节点标记为坏节点并进行计数. 坏节点累积达到一定次数后，在系统调度中会降低坏节点的打分或者设置为不可调度，以维持系统稳定.

此外，HPPS 子系统还能为分布式作业提供支持. 当评估结果为分布式数据并行的训练任务时将开启自动扩缩容功能，设置扩缩容触发器（GPU 利用率和租户资源配额）. HPPS 子系统首先通过资源监控模块捕获实时算力使用状况，并根据触发器设置触发相应的扩缩容操作. 弹性扩缩容管理器据此触发相应的扩缩容操作，包括启动或销毁训练实例，并完成网络拓扑与通信组网的动态调整.

3 实验与结果

本节对弹性训练系统在大模型训练过程应用的效果进行实验和分析.

3.1 实验设置

本文实验环境包含由 12 节点 NVIDIA A100 GPU 组成的集群 Cluster-*A*、6 节点 NVIDIA L20 GPU 组成的集群 Cluster-*B* 以及单节点 NVIDIA A800GPU 组成的 Cluster-*C*. 测试的模型结构主要为 GPT-2 和 OPT，模型大小为 13~1 090 亿. 各个服务器的具体配置分别如表 1~3 所示.

Table 1 Configuration Parameters of Server Cluster-*A*

表 1 Cluster-*A* 服务器配置参数

参数	配置环境
CPU	Intel Xeon Platinum 8378A@3.0 GHz
CPU 核数	128
操作系统	Ubuntu 22.04 LTS
GPU	8 × NVIDIA A100
内存/TB	1
网络带宽/Gbps	25

Table 2 Configuration Parameters of Server Cluster-*B*

表 2 Cluster-*B* 服务器配置参数

参数	配置环境
CPU	Intel Xeon Gold 6526Y@3.9 GHz
CPU 核数	32
操作系统	Ubuntu 22.04 LTS
GPU	8 × NVIDIA L20
内存/GB	540
网络带宽/Gbps	200

Table 3 Configuration Parameters of Server Cluster-*C*

表 3 Cluster-*C* 服务器配置参数

参数	配置环境
CPU	Intel Xeon Gold 5317 CPU@3.00 GHz
CPU 核数	48
操作系统	Ubuntu 22.04 LTS
GPU	4 × NVIDIA A100
内存/TB	2

3.2 故障类型与恢复时间

本节测试了大模型训练过程中遇到的 6 种典型故障后容错系统端到端自动恢复耗时，故障类型的释义参见表 4.

Table 4 Explanation of Fault Types

表 4 故障类型释义

故障类型	释义
致命异常	训练进程抛出的不可自动修复异常，比如 GPU OOM、NCCL 异常等
pod 崩溃	Kubernetes 控制层不合理的迁移操作、节点故障、OOM 等情况导致的 pod 直接中断退出
pod 误杀	研发或运维人员误操作导致 pod 被删除
网络断开	计算节点网络断开
节点宕机	计算节点由于断电、故障导致的宕机
组件故障	容错机制核心组件自身遭遇的故障

本文首先验证了采用 Megatron-LM 在 Cluster-*A* 集群预训练 GPT-109B 模型时容错框架的工作效果. 如图 6 所示，从实验结果看，当作业遭遇典型软硬件故障时，容错框架能够稳定地快速恢复训练作业，端到端恢复过程涵盖故障感知及定位、故障节点驱逐、失败任务重调度、进程组初始化、基于 CKPT 数据恢复训练状态等.

在测试所选择的 6 类故障下，当发生网络断开和

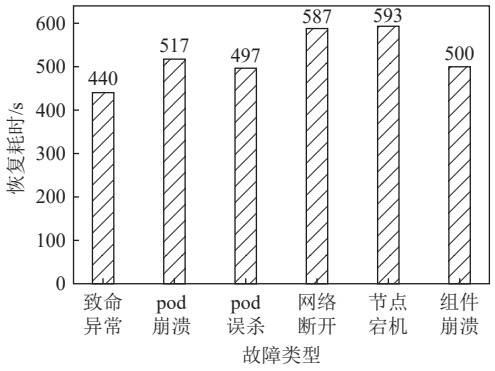

Fig. 6 End-to-end recovery time-consuming for GPT-109B model

图 6 GPT-109B 模型端到端恢复耗时

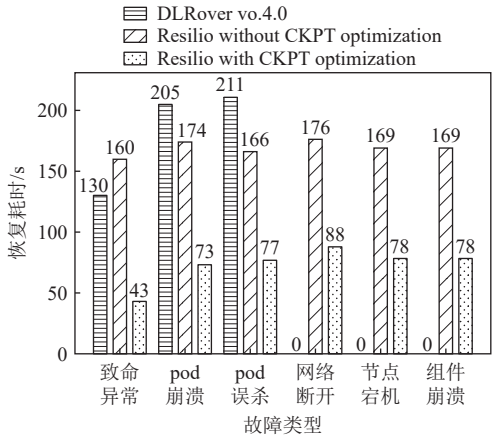

Fig. 7 End-to-end recovery time-consuming for GPT-13B model

图 7 GPT-13B 模型端到端恢复耗时

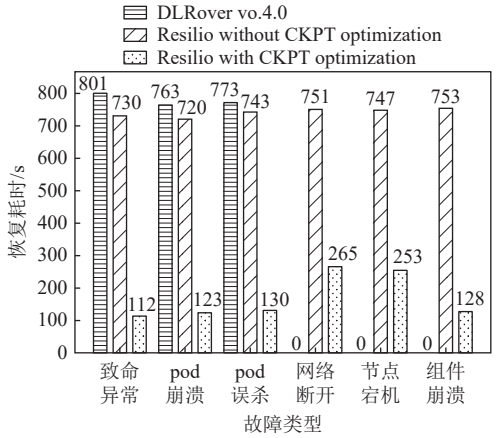

Fig. 8 End-to-end recovery time-consuming for GPT-70B model

图 8 GPT-70B 模型端到端恢复耗时

节点宕机时, 故障节点上的 pod 会持续处于终止状态而不退出, DLRover 框架无法处理这种情况, 因此整个训练作业会长时间悬挂. Resilio 框架在管理训练作业时, 每个训练进程会周期性地向控制节点上报心跳信息, 对于上述情况发生时, 控制节点能够准确感知故障节点, 然后强制删除故障节点之上的训练进程以及 pod, 同时将该故障节点从资源列表驱逐, 并重新拉起 pod. 在本文的测试中, 对于一些 GPU 故障, DLRover 虽然能够拉起作业, 但是由于对应的 GPU 资源未被驱逐, 因此下次重启, 作业可能仍然被调度到这些故障 GPU, 导致作业无法重启恢复. 而 Resilio 提供了一种轻量级的 GPU Device Plugin 实现, 当发生此类异常后, Resilio 会将故障 GPU 设备从节点上维护的可用 GPU 设备清单中剔除, 并通过 Kubelet 上报给 Kubernetes, 不仅避免了 pod 重调度失败的问题, 并且提供了卡粒度的故障资源驱逐. 此外由于故障具有随机性的特征, 当故障导致 DLRover 控制组件异常退出时, 异常作业无法稳定恢复, 而 Resilio 的控制组件提供高可用机制, 能够很大程度保证作业的稳定恢复.

为了进一步验证 Resilio 框架效果, 本文又在 Cluster-*B* 集群上进行了对比实验. 图 7 和图 8 分别对比了 GPT-13B 和 GPT-70B 模型训练作业出现异常时的端到端恢复耗时.

相比于 DLRover vo.4.0, Resilio 能够更快地恢复作业, GPT-70B 和 GPT-13B 模型耗时分别缩短 86% 和 67%. 性能收益主要来源于如下 2 部分:

1)Resilio 在作业重启后的训练进程协调(rendezvous)阶段更加高效. Resilio 在每次重启作业后重新进行通信组初始化过程, 而 DLRover 则采用的是将故障 pod 重新调度后, 在下次协调窗口重新加入到通信组, 实测发现, 等待协调窗口的时间和协调本身的总耗时会比重新构建通信组的耗时更长.

2)Resilio 的 CKPT 加载耗时相对更低, 在框架层面, Resilio 在重新调度待恢复的 pod 时, 会有亲和性机制. 以 pod 崩溃这类故障为例, 当判断造成 pod 崩溃的原因并非节点故障时, 在下次调度 pod 时会优先将 pod 调度至原节点, 而由于该节点保存有作业的 CKPT 数据, 因此提升了 CKPT 加载阶段的缓存命中率, 大大缩短此阶段的耗时.

此外, DLRover 采用了一种扩展 PyTorch 核心接口的方法, 相当于自实现了一套 torchrun, 这种方式一定程度限制了用户创建训练作业的方式, 技术演进路径也将和主流的 PyTorch 逐渐分离. 而 Resilio 框架并不侵入训练框架层, 仅关注分布式进程的生命周期管理, 能够支持各类训练作业的容错恢复, 对上层训练作业透明. 在稳定性方面, Resilio 表现更佳, 在软硬件故障测试过程中, DLRover 并不能稳定恢复

作业，即使对于一些能够最终恢复的情况，也需要多次尝试才可完成恢复，其中造成这个情况的主要原因包括 network-check 误报、内存不足（out of memory, OOM）、异常传播等.

3.3 检查点读写性能

表 5 展示了不同大小的 GPT 模型在不同存储和加载方式下的时间消耗情况.

Table 5 Comparison of CKPT Loading Time

表 5 CKPT 加载时间对比

模型	CKPT 大小/GB	加载时间/s			
		原生 NFS	DLRover Memory	Resilio NVME	Resilio Memor
GPT-109B	1 536	546.6	3.8	44.3	1.5
GPT-70B	904	322.0	2.3	25.1	1.1
GPT-13B	176	60.5	2.2	17.8	0.9

具体来看，表 5 中列出了 3 个大小不同的 GPT 模型：GPT-109B、GPT-70B 和 GPT-13B，它们对应的 CKPT 大小分别为 1.5 TB、904 GB 和 176 GB. 原生 NFS 列表示使用网络文件系统（network file system, NFS）进行模型加载. 可以看出，随着模型大小的增加，加载时间显著增长，例如 GPT-109B 模型需要 546.62 s，这种基于 NFS 的加载方式在处理大容量数据时表现较差. 在考虑网络带宽和延迟的限制下，内存缓存方案与 DLRover 均采用内存作为缓存介质. 相较于传统的加载方式，从内存中加载数据能够实现 67~364 倍的速度提升，且随着模型规模的增大，这一加速效果更加显著，当模型规模达到 1 090 亿时，最大可实现 364 倍的速度提升. 此外，通过应用多流传输、锁页内存及异步加载等技术手段，本文研究提出的内存缓存加载方法相比 DLRover 实现了 2~2.5 倍的性能提升. 本文研究进一步引入了 NVMe 作为二级缓存机制，在内存缓存未命中的情况下，利用本地 NVME 缓存加速数据加载，同时在模型初始化阶段进行 CKPT 预加载，从而重叠部分 CKPT 加载时间，缩短 CKPT 恢复耗时. 实验结果表明，相对于传统完全依赖 NFS 进行加载的方式，该优化措施能够直接从重启后的节点 NVME 中恢复 CKPT，可以显著降低访问 NFS 带来的延迟，将 CKPT 加载速度提高 12 倍. 这一系列改进为高效加载更大规模模型提供了新的解决方案.

表 6 展示了不同规模的 GPT 模型在各种存储介质下的保存时间情况.

类似于表 5 所呈现的数据，采用原生 NFS 进行模型保存时，由于 NFS 的带宽限制，导致保存过程耗时巨大. DLRover 首先采用同步方式将数据保存到内存中，然后异步完成持久化存储，显著缩短了训练暂停的时间，该时间大致等同于数据保存至内存所需的时间，相较于传统的保存方法实现了约 275 倍的效率提升. 在此基础上，本文新增了一项优化策略，即对保存到内存的操作也实施异步处理，使得保存耗时能够与下一轮迭代的耗时部分重叠. 这一改进主要依据是以单次训练迭代的前反向计算不会修改模型参数，通过重叠 CKPT 数据传输过程与计算过程实现对传输开销的隐藏，因此整个 CKPT 保存过程只会产生 0.2 s 固定的阻塞训练时间（对应表 6 中 Resilio Block 列）. 随着模型规模的增大，加速效果更加明显，在 GPT-109B 模型中，相比 DLRover，本文方案实现了高达 10.5 倍的速度提升. 这表明，对于大规模模型的保存操作，所提出的优化方法具有显著的性能优势.

Table 6 Comparison of CKPT Saving Time

表 6 CKPT 保存时间对比 s

模型	原生 NFS	DLRover Memory	Resilio NVME	Resilio Memory	Resilio Block
GPT-109B	579.0	2.2	69.4	1.63	0.2
GPT-70B	335.0	1.3	53.8	1.18	0.2
GPT-13B	66.2	1.2	27.1	0.90	0.2

JIT CKPT 的测试环境如表 2 所示，表 7 展示了启用 JIT CKPT 功能后引入的性能开销.

Table 7 Performance Overhead of Our Proposed Scheme

表 7 本文方案性能开销

模型	CKPT 大小/GB	迭代耗时/s	迭代耗时（开启 JIT）/s	开销/%
GPT-70B	904	2.91	2.96	1.65
GPT-13B	176	2.71	2.75	1.30

由于本文方案的应用涉及使用 NCCL 拦截库，本文通过一系列实验测试了不同模型下的性能影响，结果表明整体性能开销低于 2%，这为实际生产环境中的应用提供了可行性. 表 8 进一步阐述了在发生单点故障时，本文方案机制所带来的恢复优势. 具体而言，当故障发生时，诊断过程耗时为 5~6 s，约为 1 轮迭代时间的 2 倍. 这是因为在 NCCL 拦截库检测到超

Table 8 Failure Recovery Time-Consuming of Our Proposed Scheme

表 8 本文方案故障恢复耗时

模型	CKPT 大小/GB	故障诊断耗时/s	CKPT 内存恢复耗时/s	CKPT RDMA 恢复耗时/s
GPT-70B	904	6.0	1.22	7.5
GPT-13B	176	5.4	1.02	4.4

时事件后,经过一轮迭代时间,FT Master 能够收集到所有相关的超时事件,并据此反推出故障源. 与传统的脚本检测方法相比,这种主动发现故障的方法显著缩短了故障检测时间. 此外,结合本文提出的容错系统和多层级 CKPT,在所有健康进程将 CKPT 保存到内存之后即可立即重启任务,使得被重启的健康进程可以直接从内存中读取 CKPT. 对于需要重启的故障进程,由于缺乏本地 CKPT,其将从相同数据并行组的其他成员的内存中读取 CKPT,该过程借助 RDMA 技术实现,并且利用巨页注册机制来减少 RDMA 内存锁定带来的开销. 同时,将 RDMA 读取操作异步化,使其与模型初始化过程重叠,从而隐藏这部分开销. 最终结果显示,本文方案的恢复耗时与直接从内存加载 CKPT 相当,例如 GPT-70B 模型的恢复耗时仅需 1.22 s. 这些优化措施共同提升了大规模模型训练系统的容错能力和效率.

上述分析讨论了本文方案在常规场景下的性能影响,然而在极端情况下(如高频故障发生时),该系统可能面临显著的额外同步开销.

为全面评估本文方案的鲁棒性,将进一步针对极端场景展开深入分析. 具体而言,本文方案的开销主要由 2 部分组成:

1)训练过程中固定开销. 主要由拦截库引入,如表 7 所示,该部分开销控制在训练总时间的 2% 以内.

2)故障恢复时的动态开销. 包括故障检测和即时检查点保存耗时,GPT-70B 模型在表 2 的测试环境下这部分开销为 7.22 s. 如表 9 数据所示,即使在极端故障场景(日均 100 次故障)下,该开销占比仍能保持在 1% 以下.

Table 9 Effect of Fault Frequency on CKPT Overhead

表 9 故障频率对 CKPT 开销的影响

故障频率(次/天)	故障检测耗时/s	保存耗时(本文)/s	开销占比/%
10	6.0	1.22	0.08
100	6.0	1.22	0.84

3.4 自动并行模块性能对比分析

自动并行组件的测试环境如表 3 所示,模型配置信息如表 10 所示.

Table 10 Testing List of HPPS Model

表 10 HPPS 模型测试列表

模型	*Billion*	批次大小	流水线并行度
OPT	[1.3, 2.7, 6.7]	256	{4,2}
GPT-2	1.5	256	{4,2}

针对本文 HPPS 子系统测试了指定流水线并行度下模拟器的总耗时和精度. HPPS 采用两级并行切分方式,首先完成流水线切分,在指定流水线切分的情况下再进行 DP 和 TP 的搜索. 从图 9 可以看出,和 Alpa[29] 相比,HPPS 通过解耦算子内并行(intra-operator parallelism)和算子间并行(inter-operator parallelism)能显著降低搜索空间和时间开销. 实验结果表明,在表 10 所示的模型配置下,HPPS 模拟器耗时为 61.99~135.48 s,相比 Alpa 的 310.67~591.23 s 耗时,HPPS 的模拟效率为 Alpa 的 2.72~5.63 倍,平均提升为 4.36 倍,显著提升了自动并行的执行效率.

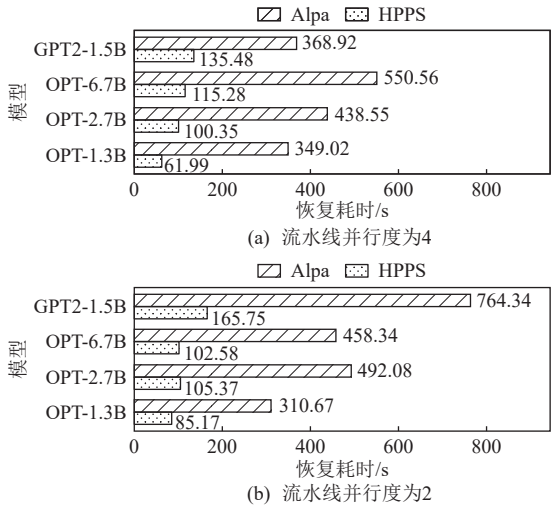

Fig. 9 Comparison of time consumption for HPPS simulator

图 9 HPPS 模拟器耗时对比

在指定流水线并行度下,HPPS 完成了 TP 和 DP 策略的搜索,对比 Alpa 的仅指定流水线切分方法,在完成流水线切分后,可通过 XLA[30] 编译出流水线阶段的可执行文件,解析流水线阶段的计算时间和通信量字节数总和. 根据不同的 GPU 数量和并行方式,调用离线通信接口获取通信时间. 最后根据式(3)计算得到单次迭代的训练时间. 表 11 所示为 HPPS 模拟器的精度数据,整体精度误差为 5.18% ~12.81%,平均误差为 7.77 %. 可见,在精度损失较小的情况下,HPPS 自动并行系统可以极大地提升自动并行最优切分策

Table 11 Accuracy Comparison of HPPS Simulator

表 11 HPPS 模拟器精度对比

模型	*Billion*	HPPS 单次迭代时间/s	Alpa 单次迭代时间/s	精度误差/%
OPT	1.3	5.23	4.95	5.59
OPT	2.7	8.17	8.62	5.18
OPT	6.7	16.61	19.05	12.81
GPT-2	1.5	8.26	7.68	7.51

略的搜寻速度.

HPPS 子系统依赖容错框架的资源监测，动态分配 GPU 资源使得资源利用率得到提升. 为了阐述 Resilio 框架在资源利用率方面的优势，本节采用模型 FLOPS 利用率（model FLOPS utilization, MFU）[31] 来评估不同参数大小下模型的算力利用表现.

基于表 3 集群和表 9 模型，本节对比了在 HPPS 和 Alpa 分别搜寻最优并行配置下模型训练输出的 MFU 大小，如表 12 所示. 结果表明，采用 HPPS 搜索得到的并行配置进行训练时，MFU 值可提升 2.05%~16.2%，平均提升 8.37%.

Table 12 Improvement of MFU Based on HPPS Test Model

表 12 基于 HPPS 测试模型的 MFU 提升 %

模型	HPPS	Alpa	MFU 提升
OPT-1.3B	35.93	33.88	2.05
OPT-2.7B	50.72	38.60	12.12
OPT-6.7B	64.64	48.48	16.16
GPT-2-1.5B	31.40	28.27	3.13

4 总 结

大模型训练因其对计算资源的长期高强度占用，常引发软硬件的频繁且多样故障，进而导致训练过程中断或效率大幅降低，这种故障不仅会影响训练任务的顺利完成，还可能造成长时间的资源浪费，因此，如何探寻故障源头、快速恢复训练进程并缩减训练停滞时间成为大规模分布式训练中的一项重要挑战. 为了应对这一问题，本文提出了一种大模型弹性容错系统，旨在提高大模型训练过程中遇到的各类故障恢复能力，并提高整体训练的可靠性与效率. 该系统能够针对训练中出现的网络与节点故障、训练进程崩溃等典型问题，提供自动化的恢复方案，确保千亿参数的大模型训练中，端到端的故障恢复时间不超过 10 min，同时将模型中断后的重新训练时间缩短至单次训练迭代时间. 此外，该系统还结合了模型分布式训练的特性与硬件存储的层次结构，通过多层次的 CKPT 读写优化，以及设计 CKPT 即时保存机制，大幅度地减少了模型训练中断后的训练恢复耗时. 同时，为了在集群资源动态变化时提高训练效率和 GPU 资源的使用率，本文还研发了一种模型训练自动配置子系统，该系统借助性能模拟器对模型训练的并行策略进行精准评估，自动确定最佳的资源分配方案，并通过与容错调度组件协同工作，确保大模型训练在资源有限的场景下能够高效运行，实现大模型的弹性训练.

综上所述，本文提出的弹性容错系统在解决大规模模型训练中的软硬件故障问题、提升容错恢复速度、优化 CKPT 机制以及自动化配置并行策略等方面，展示了显著的优势. 其在百卡规模、百亿参数大模型训练任务中的应用，不仅能够确保训练任务的高效与稳定完成，还能够显著提升系统对故障的容错能力和对集群资源的利用率，为大规模分布式深度学习训练提供了一种有效的解决方案.

作者贡献声明：李焱负责论文选题以及框架、实验设计并撰写论文；杨偲乐负责检查点功能设计相关章节撰写；刘成春负责撰写容错框架设计、故障检测等内容；王林梅和田瑶琳负责自动并行设计相关章节撰写；张信航参与检查点优化工作和协助论文修改；朱昱参与容错框架的整体设计工作；李蒨蒲和孙磊参与模型测试工作；颜深根、肖利民和张伟丰提出指导意见并修改论文.

参 考 文 献

[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]//Proc of the 31st Int Conf on Neural Information Processing Systems (NIPS). Red Hook, NY: Curran Associates, 2017: 6000–6010

[2] Zhai Ennan, Cao Jiamin, Qiankun, et al. Research on network infrastructure in the era of large models: Challenges, stage achievements, and prospects[J]. Journal of Computer Research and Development, 2024, 61(11): 2664−2677 (in Chinese)
(翟恩南，操佳敏，钱坤，等. 面向大模型时代的网络基础设施研究：挑战、阶段成果与展望[J]. 计算机研究与发展，2024，61(11)：2664−2677)

[3] Zhang Susan, Roller S, Goyal N, et al. Opt: Open pre-trained transformer language models[J]. arXiv preprint, arXiv: 2205.01068, 2022

[4] Dubey A, Jauhri A, Pandey A, et al. The Llama 3 herd of models[J]. arXiv preprint, arXiv: 2407.21783, 2024

[5] Floridi L, Chiriatti M. GPT-3: Its nature, scope, limits, and consequences[J]. Minds and Machines, 2020, 30: 681−694

[6] Meta. Pytorch: Tensors and dynamic neural networks in Python with strong GPU acceleration[CP/OL]. 2016[2024-02-19]. https://github.com/pytorch/pytorch

[7] Google. Tensorflow: An open source machine learning framework for everyone[CP/OL]. 2015[2024-02-19]. https://github.com/tensorflow/te-nsorflow

[8] Microsoft. DeepSpeed[EB/OL]. 2023[2024-02-19]. https://github.com/microsoft/DeepSpeed

[9] Shoeybi M, Patwary M, Puri R, et al. Megatron-LM: Training multi-

billion parameter language models using model parallelism[J]. arXiv preprint, arXiv: 1909.08053, 2019

[10] Wu Baodong, Xia Lei, Li Qingping, et al. TRANSOM: An efficient fault-tolerant system for training LLMs[J]. arXiv preprint, arXiv: 2310.10046, 2023

[11] Ant Group. DLRover: An automatic distributed deep learning system[CP/OL]. 2025[2025-02-19]. https://github.com/intelligent-machine-learning/dlrover

[12] Ant Group. DLRover's technical practice of training stability assurance for thousands of calorie-level large models on Kubernetes [EB/OL]. 2025[2025-02-19]. https://github.com/intelligent-machine-learning/dlr-over/blob/master/docs/blogs/stabilize_llm_training_cn.md

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

[14] Wang Guanhua, Ruwase O, Xie Bing, et al. FastPersist: Accelerating model checkpointing in deep learning[J]. arXiv preprint, arXiv: 2406.13768, 2024

[15] Gupta T, Krishnan S, Kumar R, et al. Just-in-time checkpointing: Low cost error recovery from deep learning training failures[C]//Proc of the 19th European Conf on Computer Systems (EuroSys). New York: ACM, 2024: 1110−1125

[16] 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 (SOSP). New York: ACM, 2023: 364−381

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

[18] Xiang Wu, Li Yakun, Ren Yuquan, et al. Gödel: Unified large-scale resource management and scheduling at ByteDance[C]//Proc of the 14th ACM Symp on Cloud Computing (SoCC). New York: ACM, 2023: 308−323

[19] Liu K, Kosaian J, Rashmi K V. ECRM: Efficient fault tolerancefor recommendation model training via erasure coding[J]. arXiv preprint, arXiv: 2104.01981, 2021

[20] Hu Qinghao, Ye Zhisheng, Wang Zerui, et al. Characterization of large language model development in the datacenter[C]//Proc of the 21st USENIX Symp on Networked Systems Design and Implementation (NSDI). Berkeley, CA: USENIX Association, 2024: 709−729

[21] Wu Tianyuan, Wang Wei, Yu Yinghao, et al. FALCON: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training[J]. arXiv preprint, arXiv: 2410.12588, 2024

[22] Lao C L, Yu Minlan, Akella A, et al. TrainMover: Efficient ML training live migration with no memory overhead[J]. arXiv preprint, arXiv: 2412.12636, 2024

[23] 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 (NSDI). Berkeley, CA: USENIX Association, 2024: 745−760

[24] Eisenman A, Matam K K, Ingram S, et al. Check-N-Run: A checkpointing system for training deep learning recommendation models[C]//Proc of the 19th USENIX Symp on Networked Systems Design and Implementation (NSDI). Berkeley, CA: USENIX Association, 2022: 929−943

[25] Li Mingzhen, Xiao Wencong, Yang Hailong, et al. EasyScale: Elastic training with consistent accuracy and improved utilization on GPUs[C]//Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis (SC). New York: ACM, 2023: 1−14

[26] Subramanya S J, Arfeen D, Lin Shouxu, et al. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling[C]//Proc of the 29th Symp on Operating Systems Principles (SOSP). New York: ACM, 2023: 642−657

[27] Wagenländer M, Li Guo, Zhao Bo, et al. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections[C]//Proc of the 30th Symp on Operating Systems Principles (SOSP). New York: ACM, 2024: 195−210

[28] NVIDIA. NCCL Tests: Check both the performance and the correctness of NCCL operations[CP/OL]. 2017[2025-02-19]. https://github.com/N-VIDIA/nccl-tests

[29] 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 (OSDI). Berkeley, CA: USENIX Association, 2022: 559−578

[30] Google. XLA: An open-source machine learning (ML) compiler for GPUs, CPUs, and ML accelerators[CP/OL]. 2022[2025-02-19]. https://github.com/openxla/xla

[31] Chowdhery A, Narang S, Devlin J, et al. Palm: Scaling language modeling with pathways[J]. Journal of Machine Learning Research, 2023, 24(240): 1−113

Li Yan, born in 1985. PhD, senior engineer. Member of CCF. His main research interests include high performance computing, heterogeneous computing, and deep learning.

李 焱, 1985 年生. 博士, 高级工程师. CCF 会员. 主要研究方向为高性能计算、异构计算、深度学习.

Yang Sile, born in 1994. Master. His main research interests include heterogenous computing and network communication.

杨偲乐, 1994 年生. 硕士. 主要研究方向为异构计算、网络通信.

Liu Chengchun, born in 1992. Master. His main research interests include high performance computing and heterogeneous computing.

刘成春, 1992 年生. 硕士. 主要研究方向为高性能计算、异构计算.

Wang Linmei, born in 1985. Master. Member of CCF. Her main research interests include high performance computing and heterogeneous computing.

王林梅, 1985 年生. 硕士. CCF 会员. 主要研究方向为高性能计算、异构计算.

Tian Yaolin, born in 1999. Master. Her main research interests include optimization of LLM, heterogeneous computing, and visual SLAM.

田瑶琳, 1999 年生. 硕士. 主要研究方向为大语言模型优化、异构计算、视觉 SLAM.

Zhang Xinhang, born in 2000. Master. His main research interest includes distributed training of deep learning.

张信航, 2000 年生. 硕士. 主要研究方向为深度学习分布式训练.

Zhu Yu, born in 1998. PhD candidate. His main research interests include LLM training architecture design and multi-core architecture.

朱　昱, 1998 年生. 博士研究生. 主要研究方向为大模型训练架构设计、多芯粒架构.

Li Chunpu, born in 1978. Bachelor. Her main research interests include heterogeneous computing and cloud computing.

李蒓蒲, 1978 年生. 学士. 主要研究方向为异构计算、云计算.

Sun Lei, born in 1980. Bachelor. His main research interests include heterogeneous computing and edge computing.

孙　磊, 1980 年生. 学士. 主要研究方向为异构计算、边缘计算.

Yan Shengen, born in 1986. PhD, associate professor. His main research interests include heterogeneous computing, intelligent computing, and machine learning systems.

颜深根, 1986 年生. 博士, 副教授. 主要研究方向为异构计算、智能计算、机器学习系统.

Xiao Limin, born in 1970. PhD. Distinguished member of CCF. Distinguished researcher at Lenovo Research. His main research interests include computer architecture, heterogeneous intelligent computing, high performance computing, and intelligent computing chip.

肖利民, 1970 年生. 博士. CCF 杰出会员. 联想研究院首席研究员. 主要研究方向为计算机体系结构、异构智能计算、高性能计算、智能计算芯片.

Zhang Weifeng, born in 1968. PhD. Corporate VP at Lenovo Group. His main research interests include computer architecture, heterogeneous intelligent computing, wireless communication, and intelligent computing chip.

张伟丰, 1968 年生. 博士. 联想集团副总裁. 主要研究方向为计算机体系结构、异构智能计算、无线通信、智能计算芯片.