基于混合并行的分布式训练优化研究

徐金龙, 李鹏飞, 李嘉楠, 陈飙元, 高伟, 韩林

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231200128

基于混合并行的分布式训练优化研究

徐金龙[1,3] 李鹏飞[2] 李嘉楠[2] 陈飙元[2] 高 伟[1] 韩 林[1]
1 国家超级计算郑州中心(郑州大学) 郑州 450000
2 郑州大学计算机与人工智能学院 郑州 450000
3 战略支援部队信息工程大学 郑州 450000
(longkaizh@163.com)

摘 要 大型神经网络训练是深度学习领域的一个热点话题,而分布式训练是基于多节点实现大型神经网络训练的最佳方法之一。分布式训练通常包含数据并行、层间并行和层内并行3种并行方法。然而现有的框架在层间并行时只能对模型进行手动切分,增加了模型设计的抽象复杂度,对此提出了节点约束关系搜索算法,实现了模型的自动切分。另外,在传统的数据并行和层间并行中,由于模型的复杂约束关系和通信操作的需要,计算和通信往往受到严格的序列化限制,为此引入了同步优化算法,实现了计算和通信的重叠,有效提高了整体训练的效率。实验对不同规模的GPT-2,AlexNet,VGG16和ResNet50模型进行训练,使用同步优化算法在6节点条件下可以将GPT2-XL,GPT2-LARGE和GPT2-MEDIUM模型的训练性能分别提升1.14倍、1.18倍和1.23倍,在1节点条件下将AlexNet,VGG16和ResNet50模型的训练性能分别提升1.31倍、1.14倍和1.03倍。实验结果表明,同步优化算法能够提升混合并行中的训练效率。

关键词:分布式训练;混合并行;自动切分;通信优化;梯度同步
中图分类号 TP391

Study on Distributed Training Optimization Based on Hybrid Parallel

XU Jinlong[1,3],LI Pengfei[2],LI Jianan[2],CHEN Biaoyuan[2],GAO Wei[1] and HAN Lin[1]
1 National Supercomputing Center in Zhengzhou(Zhengzhou University),Zhengzhou 450000,China
2 School of Computer and Artificial Intelligence,Zhengzhou University,Zhengzhou 450000,China
3 Strategic Support Force Information Engineering University,Zhengzhou 450000,China

Abstract Large-scale neural network training is a hot topic in the field of deep learning,and distributed training stands out as one of the most effective methods for training large neural networks across multiple nodes. Distributed training typically involves three parallel methods:data parallelism,inter-layer parallelism,and intra-layer parallelism. However,in existing frameworks,manual model partitioning is required for inter-layer parallelism,which increases the abstract complexity of model design. To address this issue,we propose a node-constrained relationship search algorithm that automates the model partitioning process. Moreover,in traditional data parallelism and inter-layer parallelism,strict serialization limits the overlap of computation and communication due to complex model constraints and the need for communication operations. To overcome this challenge,we introduce a synchronous optimization algorithm,enabling the overlap of computation and communication and effectively enhancing the overall training efficiency. The experiments involve training GPT-2 of different sizes,AlexNet,VGG16,and ResNet50 models. Using the synchronous optimization algorithm under a 6-node configuration,the training performance of GPT2-XL,GPT2-LARGE,and GPT2-MEDIUM models is improved,achieving speed-ups of 1.14,1.18,and 1.23,respectively. Under 1-node configuration,performance enhancements are also observed for AlexNet,VGG16,and ResNet50 models,with speed-ups of 1.31,1.14,and 1.03,respectively. The experimental results indicate that the synchronous optimization algorithm effectively enhances the training efficiency in mixed parallelism.

Keywords Distributed learning,Hybrid parallel,Automatic segmentation,Communication optimization,Gradient synchronization

近年来,深度学习领域一直在朝着大型神经网络的方向发展,大型神经网络的普及提升了GPU等硬件的计算能力。然而,训练大型网络的存储器要求已经远远超过了现代硬件加速器的DRAM容量。这就需要在基于大规模的GPU集群

到稿日期:2023-12-19 返修日期:2024-04-29
基金项目:河南省重大科技专项(221100210600)
This work was supported by the Major Science and Technology Project of Henan Province(221100210600).
通信作者:韩林(strollerlin@163.com)

上开发高效的算法来并行训练神经网络，以充分利用多机资源，加速模型训练过程。

分布式训练方式主要分为 3 种：数据并行、层内并行和层间并行。当然传统单一的分布式训练方式并不能完全满足日益增长的模型训练需要，在大型神经网络和海量数据集的驱动下出现了混合并行。混合并行通常会同时采用 2 种或 3 种分布式训练方法，例如框架 Megatron-LM[1]，PaddlePaddle[2] 和 DeepSpeed[3] 等。

但是，现有的混合并行框架在训练过程中仍然存在着缺点。

首先，现有的框架在使用过程中不够通用。对于数据并行，各个框架的底层基本都是采用 DP 的思想[4-7]，将数据集划分到各个 GPU 上，从而完成训练。而对于层间并行，各个框架在设计模型时基本都需要将 GPU 数量考虑进去[1,7]，对开发人员的专业素养有着极高的要求，不合理的模型划分会导致训练中出现不必要的通信开销，引发性能波动。如果要进行框架之间的迁移，则又需要重新设计模型的划分方式和接口，这大大增加了开发人员的工作负担。

其次，现有的混合训练过程没有充分解决通信开销问题。以数据并行和层内并行为例，数据集被划分到多个 GPU 设备后，再将模型根据层划分到另外的 GPU 设备上。因为每个设备使用的是数据集的一部分进行参数更新，因此在一次完整的训练之后，需要使用集合通信原语对层内并行编号相同而数据并行编号不同的 GPU 设备进行一次参数同步，同步完成之后进行下一轮的训练[8]。在整个流程中，通信主要集中在层内并行的层与层之间以及最后的参数同步过程。层与层之间的数据传输在其他框架中通常会使用 NCCL[9] 进行点对点的通信，但是由于国产 DCU 的特殊性，无法使用 NCCL 作为通信后端，因此需要考虑其他的通信后端。对于最后的参数同步过程，无法将计算和通信进行重叠，如果可以将最后的通信和计算进行重叠，那么必将加快整体的训练速度。因此，针对混合并行中的通信开销问题，是本文需要解决的第二个问题。

为了解决上面的问题，基于混合并行提出了针对通信的优化技术，并实现了对模型进行自动划分的方法，最后将这二者进行整合并完成测试。首先对模型进行追踪，获取到计算图的完整信息，根据计算图以及 GPU 的数量，将模型进行自动切分，切分中考虑到层与层之间的约束关系，尽量保证具有约束关系的层被分到同一个 GPU 设备上。针对通信优化问题，使用 MPI[10] 作为点对点通信后端，由于 MPI 本身支持异步通信，这一特性使得 MPI 能够实现更高的通信带宽，并允许计算和通信过程的重叠执行以提高整体性能。而对于最后的通信过程和计算重叠，由于模型分层的结构特性，考虑在每一层反向传播之后便进行同步，这样在后面层进行反向传播计算便和通信实现了重叠。对本文实现的两种技术进行实验分析，结果表明通信优化技术可以实现通信和计算的重叠，缩短整体训练时间，而自动划分又提高了开发人员的效率。

本文的主要贡献包括两个方面：

1）设计了一个针对深度学习模型的自动划分算法，该算法可以根据模型结构和硬件资源情况，自动确定模型各层的最佳划分方案。通过该算法实现的自动模型划分，可以有效减轻分布式训练的工作量，降低人工划分的失误成本。

2）针对分布式训练过程中的通信开销进行了优化。使用同步优化算法改进混合并行方式中的通信机制，实现计算和通信的重叠，加速整体训练流程。

1 背景介绍

1.1 大规模神经网络训练

随着深度学习网络模型的不断发展，神经网络的性能和准确率持续提升，同时网络模型的层数也从最初的数十层发展到现在的上百层，网络的计算负担不断增加，这显著增加了训练完整网络模型所需的时间成本。为了减少训练时间，采用分布式训练方法来处理网络模型成为了一种有效的方法。

Coates 等将分布式机器学习中的数据并行和模型并行思想迁移到深度学习领域[11]，在大规模集群上实现了高效的深度神经网络训练。数据并行将大规模的训练数据划分成多个部分，并使每个计算节点负责处理其中一个部分[12]。每个节点使用自己的部分数据计算梯度，并通过通信机制将梯度汇总，然后在全局模型上进行参数更新。使用模型并行时，一个完整模型会被分解成多个子模型，每个子模型在不同的计算节点上进行计算，最后将各自的输出进行整合，得到正确结果。

将模型并行概念进行细分，则可以进一步获得两种并行范式：层内并行和层间并行。常用的层内并行方案由 Megatron-LM 提出，它是一种高效的张量并行实现，即将神经网络一个层的计算划分到多个子任务，每个子任务在不同的 GPU 上执行计算层输出激活的部分，这些部分输出通过集体通信原语拼接在一起，以供下一层计算使用[12]。

层间并行，通常也被称为流水线并行，在工作的 GPU 之间对神经网络的层进行划分。传统层间并行在任意时刻只会使用一个 GPU，其余 GPU 都会处于闲置状态。Gpipe 是谷歌提出来的层间并行解决方案，通过将输入批次切分成更小的微批次，来有效应对传统层间并行所引发的 GPU 闲置问题。PipeDream 在 Gpipe 的基础上，使用 1F1B（One Forward pass followed by One Backward pass）策略进一步减少 GPU 显存开销和整体闲置时间[13]。

混合并行是一种同时利用数据并行、层内并行和层间并行等多种并行计算策略的方法。Megatron-LM 和 DeepSpeed 通过混合并行来训练规模极其庞大的 Transformer 神经网络，例如 GPT 3[14]，以更充分地利用多个计算资源，从而提高深度学习模型的训练效率。

1.2 通信优化研究

无论采用何种分布式训练策略，通信阶段往往都占据了训练过程的大部分时间。实现通信的基本操作是通信原语，这些原语用于协调来自不同计算节点的梯度信息，以确保全局模型的参数更新是基于整个数据集的梯度。图 1 展示了一些常见的通信原语，如 All-Reduce，All-Gather，ReduceScatter 以及 Broadcast 等，它们的应用实现了参数的同步，进而保证了训练结果的一致[15]。

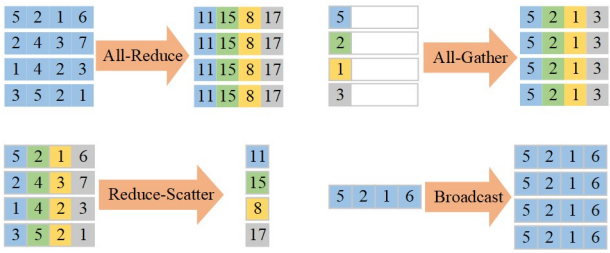


图 1 通信原语

Fig. 1 Communication primitives

为了减少通信开销，学术界积极从不同方向探索如何优化分布式训练中的通信过程，归纳起来主要分为 3 个方向：减少单次通信开销，减少通信次数，以及实现计算和通信重叠。

减少单次通信开销最广泛使用的技术是梯度压缩，其通过将浮点梯度转换为较小的整数或定点数，来减少梯度通信所需的带宽和时间。减少通信次数通常从两个方面进行考虑：一是增大每次通信的数据量，即使用大批次进行训练；二是通过 LocalSGD 这类算法，在一定时间间隔内执行通信操作，而不是在每个迭代步骤中都进行通信。

为了实现更高的计算和通信重叠比，异步更新算法成为了分布式训练的主流。Hogwild! 算法首次在分布式训练中使用了异步更新机制；由于异步更新算法会导致精度下降，Zhang 等基于 SGD 提出了弹性平均 SGD 算法来保证收敛时的精度；Zheng 等则提出了基于延迟补偿的异步随机梯度下降算法 DC-ASGD 来提高训练的效率和效果[16]。

为了探索通信优化的潜力，分布式训练框架 Poseidon 基于神经网络的层次依赖关系，提出了一种优化算法，称为无等待反向传播算法[17]。该算法在优化数据并行中的通信问题方面取得了显著的成果，成功实现了计算和通信的高效重叠。

上述通信优化方法尽管在降低通信整体开销方面发挥了一定作用，但对于混合并行中的通信问题却缺乏详尽深刻的阐释。大多数优化方法仅关注单一数据并行中的通信问题，而缺乏对混合并行的全面考量。本文深入研究了混合并行中存在的通信问题，从计算和通信重叠的角度提出了同步优化算法，加快了训练速度。

2 系统架构设计

整个系统构建在 PyTorch 深度学习框架之上。当使用 PyTorch 进行单机训练时，通常的训练步骤为：处理数据集、定义模型、优化器、损失函数和正式训练，如图 2(a)所示。为了降低开发人员的学习成本，减小对 PyTorch 代码的侵入程度，整个系统的实现独立于 PyTorch 本身，用户通过调用设计好的接口即可实现完整的混合并行训练，相比单机训练主要添加了建立通信组、划分数据集、生成分布式模型、注册通信钩子以及分布式训练接口，整体训练流程如图 2(b)所示。分布式训练的核心在于各个节点之间的通信，因此在训练开始时需要定义通信组，例如在数据并行和层间并行混合并行中，需要在不同的数据并行进程节点之间建立通信，而在层间并行中则需要在不同的层间并行进程节点之间建立通信。针对数据并行，需要将数据集进行划分，确定每个 GPU 会获得的数据集量。如果选择层间并行，则将事先定义好的模型传递给图分片算法，以进行自动切分，确定各个 GPU 上的分片位置，生成分布式模型。然后，通过注册通信钩子，实现梯度同步策略的修改，这是同步优化算法的核心。本文将重点介绍模型自动划分的实现，以及同步优化算法如何进行计算和通信重叠，从而加快混合训练的速度。

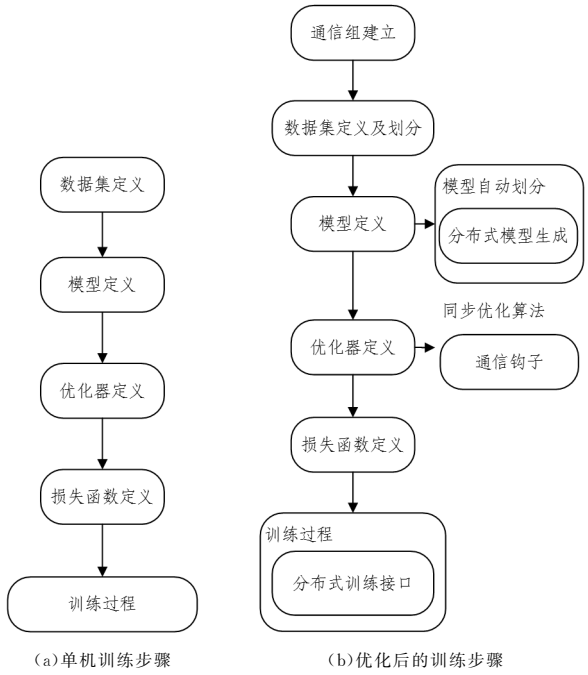


图 2 整体训练架构

Fig. 2 Overall training architecture

3 模型自动划分的实现

本章主要对模型自动划分算法进行具体介绍。图 3 展示了模型自动划分的整体流程：首先是获取模型的计算图信息，接着通过约束搜索算法得到节点与分片之间的映射关系，最后通过映射关系划分计算图得到计算图分片。

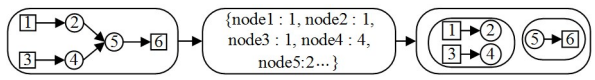


图 3 模型自动划分的整体流程

Fig. 3 Overall process of automatic model partitioning

3.1 计算图获取

为了获取 PyTorch 计算图信息，必须使用一个统一的方法对模型进行追踪遍历。在 PyTorch 中，使用图追踪工具 torch.fx[18]对整个模型进行追踪并将模型的前向计算逻辑转换成一个可操作的中间表示 *IR*，写入到计算图 GraphModule 中。

经过追踪获取到的模型计算图 GraphModule 包含了模型的完整信息。为了实现模型的自动划分，需遍历模型计算图，根据 GPU 数量等信息将完整计算图划分为相应子图，进而将子图分配给不同的 GPU 来构造完整的流水线。此外，子图与子图之间的数据传输是层间并行的主要通信开销，合理的划分方式可减小层间传输的通信量。因此，本文提出一种约束搜索算法，在模型分片之前确定计算图中各个节点的约束关系。这个约束关系不仅可以减少层间传输的通信量，

还可以保证每个分片的参数量相似，以使得 GPU 负载均衡。

3.2 约束关系确定

计算图是一种有向无环图，计算图中的节点表示具体的操作，例如加法、乘法、激活函数等，边表示数据流向。节点之间的约束关系是计算图自动划分的关键。具体来说，约束关系指每个节点和子图之间的映射关系。为了确定映射关系，需要从两方面进行考虑：一是数据的依赖关系，即确保每个分片的输入数据都源自上一个分片的输出，数据不会跨分片传输；二是关注 GPU 负载状态，确保每个分片的参数量基本一致，从而使得 GPU 的利用率保持基本相同。

因此，本文基于 PyTorch 计算图 GraphModule 设计了约束关系搜索算法来确定映射关系，如算法 1 所示。算法首先计算出模型的参数总量，并根据划分数量 n 确定每个分片的参数量（第 2－5 行）。然后遍历计算图的每个节点，根据节点的类型进行相应处理（第 7－21 行）：计算图 GraphModule 的节点类型分为占位节点、调用节点和输出节点等[18]，遇到占位节点会将该节点放入节点分片映射关系中；对于调用节点，遍历其参数列表 args。为了确保下一个分片的参数来自于上一个分片，当分片参数不在上一分片时，将该参数所在的分片与当前分片之间的所有分片进行合并（第 13－14 行）。接下来，将节点分配给分片以确定节点和分片之间的映射关系 node_shard_map，并更新分片的参数数量（第 16－17 行）。如果当前分片的参数量大于预先设置好的每个分片参数量 per_shard_size，则创建一个新的分片 ID（第 18－20 行）。最后，返回节点分片映射关系 node_shard_map。

算法 1 约束关系搜索算法

```
输入：计算图 G，划分数量 m
输出：节点到分片的映射 node_shard_map
1. shard_id＝0
2. for name，module in G. named_modules() do
3.     param_size[name]＝compute_size(module)
4. end
5. per_shard_size＝param_size / n
6. for node in G. nodes do
7.     if node. is_placeholder then
8.         node_shard_map[node. name]＝shard_id
9.     else if node. is_output then
10.        break
11.    else
12.        for arg in node. args do
13.            if arg not in prev_shard then
14.                merge_shards()
15.            end
16.        assign_node_to_shard(node，shard_id)
17.        update_shard_params()
18.        if now_shard_size ＞ per_shard_size then
19.            shard_id＝shard_id＋1
20.        end
21.    end
22. end
```

3.3 计算图划分

映射关系 nodeshardmap 是一种类似于{node1：1，node2：1，node3：2，node4：2}结构的字典，字典的键存储了计算图节点信息，而相应的值表示该节点对应的分片 ID。相较于直接对原有计算图进行划分，创建新的计算图显然更为方便，因此计算图划分算法遍历原有计算图并根据映射关系创建新的分片信息，具体细节可参见算法 2。

算法 2 分片算法

```
输入：计算图 G，节点和分片映射 node_shard_map
输出：进行分片后的计算图列表 shard_list
1. for node in G. nodes do
2.     if node in next_shard then
3.         prev_shard. insert_node(output)
4.         shard_count＋＋
5.         shard_list. append(prev_shard)
6.         create_new_graph()
7.         get_new_input()
8.     end
9.     if exits new_input then
10.        set_new_input()
11.    end
12.    if node. op is not output then
13.        new_graph. insert(node)
14.    else
15.        prev_shard. insert_node(output)
16.        shard_list. append(prev_shard)
17.     break
18. end
```

算法在遍历计算图过程中，对每个节点进行判断。如果判断结果表明该节点应该位于下一个分片中，那么会在前一个分片的末尾插入一个输出节点（第 2－3 行）。接着，将分片数量增加 1，并将新的分片添加到计算图列表 shard_list 中；随后，创建新的计算图分片，将当前节点作为新分片的输入节点（第 3－8 行）。如果刚才创建了新的输入节点信息，便设置该节点为新分片的输入（第 9－11 行）。如果节点不是 output 节点，则将该节点放入新的计算图分片中，否则向上一分片添加 output 节点并将该分片添加到 shard_list 中（第 12－17 行）。最后，遍历完所有计算图节点后，返回经过分片后的计算图列表 shard_list。

分片算法将完整的计算图划分为分片列表后，就可以将模型分片分配给不同的 GPU 设备，从而实现自动层间并行。以图 4 中的 GPT-2 Small 模型为例，该模型包括词向量编码和位置编码、12 个 Transformer 解码器（其中每个 Transformer解码器包含 1 个多头自注意力层、1 个前馈层和 2 个 LayerNorm 层）以及 1 个 LayerNorm 层[19]。如果想要进行 4 路流水并行训练，就需要将模型分为 4 个分片。经过完整的自动划分算法后，第一个分片包含词向量编码层，第二个分片包含位置编码以及 Dropout 层，第三个分片为 6 个 Transformer 解码器，第四个分片包含所有未被划分的剩余层。经过划分后的模型可以分配给不同的 GPU 设备，从而方便接下来的训练过程。由于是混合并行，不仅需要考虑如何进行层间并行划分，还需要考虑如何优化混合并行中的通信。接下来将讲解同步优化算法的具体实现。

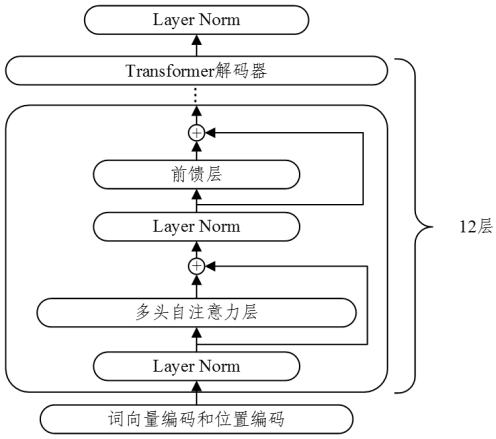


图 4 GPT-2 Small 模型结构

Fig. 4 Model structure of GPT-2 Small

4 通信优化算法

在数据并行加层间并行的混合并行中，模型梯度的计算和传输按照先进先出的顺序进行[8]：首先框架将数据集进行切分以实现数据并行，之后将数据集分片分给不同的层间并行组，每个层间并行组获取数据集分片后执行流水线训练。在计算出模型的所有层的梯度后，再将梯度同步传输给同一数据并行组中的其他节点。

图 5 给出了 2 路数据并行 4 路层间并行的混合并行训练流程，其中 DRANK 表示数据并行编号，蓝色表示前向传播，橘黄色为反向传播，绿色为通信同步过程，编号 0,1,2,3,4,5,6,7 表示将一个批次划分成了 8 个微批次，每个微批次都会独立进行前向传播和反向传播。在最后一个微批次的反向传播之后，每个 GPU 设备会执行通信操作，与在另一数据并行组中对应的 GPU 进行同步，例如 GPU 1 会与 GPU 5 进行同步。在整个训练阶段，计算过程主要与 GPU 有关，而通信操作涉及网络设备，这两者本质上并没有直接的冲突。然而传统的混合并行训练通常将计算和通信视为独立的过程，导致计算和通信的执行变得串行化，显然这种串行的计算和通信方式会降低整体训练速度。

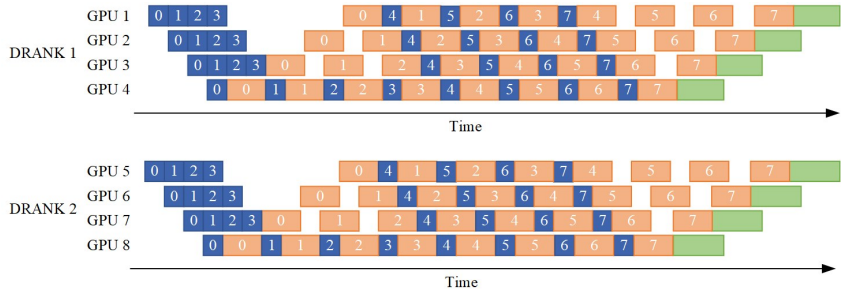


图 5 数据并行与层间并行的混合并行训练过程(电子版为彩图)

Fig. 5 Mixed parallel training process of data parallel and inter-layer parallel

深度神经网络具有层次化的结构，其中每一层仅依赖于相邻层的参数。为了充分利用这种层次化结构，本文提出了一种同步优化算法，将通信方式从原有的梯度统一传输修改为按层次进行的传输同步。具体来说，一旦某一层的梯度在反向传播过程中被计算出来，该层的计算结果会立即进行传输，使得最后一个梯度传输和最后一个微批次的反向传播重叠，而无需等待最后一个微批次的反向传播。图 6 展示了优化前后通信同步方式的变化，同步优化算法在最后一个微批次的反向传播中，针对层的结构进行通信同步，实现并行的计算和通信。比较图 6(a)和图 6(b)可以发现，优化算法有效缩短了训练的整体时间。

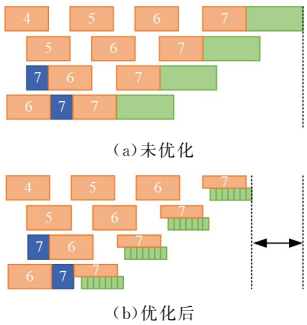


图 6 优化前后通信同步的变化

Fig. 6 Changes in communication synchronization methods before and after optimization

需要注意的是，并行计算和通信的方式本身并不会减少通信的总时间，而是通过在计算过程中隐藏部分通信开销来实现并行执行通信操作。并行方法在计算设备空闲时执行通信操作的这种并行方式可以最大限度地利用计算资源和通信资源，减少计算和通信整体的时间开销，从而在整体上加速训练过程。接下来将详细介绍同步优化算法的实现。

4.1 梯度通信定位

梯度通信定位指在分布式训练中决定何时以何种方式在哪些地方进行梯度信息的传递。不同的分布式训练策略在梯度通信位置上存在着不同。在数据并行的分布式训练中，不同设备上处理不同的数据批次，梯度通信通常发生在每个批次的训练结束后。而对于如图 5 所示的层间并行和数据并行的混合并行，由于在同一 DRANK 中不同设备负责处理模型的不同部分，而在不同 DRANK 中 GPU 1 与 GPU 5 使用模型的相同部分处理不同的数据，因此不同 DRANK 中对应的 GPU 设备需要在批次训练结束之后进行梯度同步；又因为层间并行中一个批次被分成了多个微批次进行同时训练，所以需要在最后一个微批次的反向传播之后进行梯度通信，即在编号为 7 的微批次反向传播完成之后进行梯度同步操作。由于同步优化算法的核心是修改通信同步方式，因此确定梯度通信位置是实现同步优化算法的第一步。

在具体实现上，需要找到最后一个微批次的执行时机。观察层间并行的整体流程不难看出：每个 GPU 的启动与否

由前一个或后一个 GPU 是否发送消息来决定。第一个 GPU 仅发送前向传播消息和接收后向传播消息，最后一个 GPU 仅接收前向传播消息和发送后向传播消息，而中间的所有 GPU 既发送又接收前向传播或后向传播的消息。由于发送消息是 GPU 主动进行的操作，无法被直接感知，因此对于发送消息不予以处理，仅统计接收消息的数量。通过这种方式，可以量化每个 GPU 接收的消息数。以图 5 为例，GPU 0 接收消息数为 8，GPU 1-GPU 2 接收消息数为 16，GPU 3 接收消息数为 8。当每个 GPU 剩余接收消息数为 1 时，即可确定此时仍有最后一个微批次尚未进行反向传播。接着设置标志变量为 True，通信操作监测到标志变量变化并为 True 时，开始执行梯度通信。通过将 GPU 消息数进行量化，确保了梯度通信位置。接下来修改此处的通信方式，以实现同步优化算法。

4.2 同步优化策略

同步优化算法的原理是将混合并行中的梯度按照层的结构进行同步，即每一层反向传播后直接进行梯度通信请求。为了实现这一功能，需要使用到钩子技术（Hooking）。钩子是计算机程序设计术语，简单来说，指通过拦截软件模块间的执行流程来扩展原有应用程序的技术。同步优化算法对模型的每一层注册一个钩子，当该层反向传播结束后自动执行注册好的钩子，而由于层间并行中不同设备存放模型的部分信息，并且每个批次被分成了多个微批次进行流水训练，因此需要对每个设备上的模型分别注册钩子，且只在最后一个微批次执行钩子。

同步优化算法的具体实现如算法 3 所示。首先初始化一个同步队列，用于存储模型的每一层信息，并注册通信钩子。钩子内部会通过标志变量 RunFlag 判断整个训练过程是否为最后一次反向传播，并从同步队列中读取层信息，以进行梯度同步（第 1－2 行和 9－18 行）。需要注意的是，钩子只会在每层反向传播之后执行，这里仅仅是定义钩子的具体内容。注册完钩子后，接下来采用消息驱动的方式进行混合训练。具体而言，GPU 如果还有未接收消息，将持续进行消息监听以判断消息来源。如果消息来自前一个 GPU，则执行前向传播，否则执行后向传播（第 20－21 行）。如果没有剩余消息需要接收，接下来则进行最后一层的反向传播。之后，在每一层梯度计算完成后，注册好的通信钩子会被自动执行（第 22－24 行）。GPU 0 在完成上一个微批次的前向传播后，会自动获取下一个微批次并进行前向传播和发送（第 25－28 行）。由于存在消息监听器，最后一个 GPU 在完成微批次的前向传播后将进行反向传播，并将反向传播的梯度发送给前一个 GPU，以确保前面的 GPU 可以执行反向传播（第 29－31 行）。最后，将标志变量 RunFlag 设置为 False，完成训练。

算法 3 同步优化算法

输入：模型分片 mode_shard，数据分片 batch_size，层间并行数 inter，数据并行数 data，GPU 编号为 $G_{i,j}$，训练次数 I

```
1. Initialize SyncQueue Q //初始化同步队列 Q
2. RegisterCommunicationHandle(Q)
3. for i → 1～I do
4.     micro_batch=DivideBatchSize(batch_size)
5.     Train(micro_batch)
6.     WaitCommunicationFinish()
7. end
8. RunFlag=false
9. Function RegisterCommunicationHandle(Q):
10.    while RunFlag do
11.      if last_backword then
12.        l=Q.pop()
13.        syncAllReduce(l)
14.        if l == 1 then
15.          FinishCommunication()
16.        end
17.      end
18.    end
19. Function Train(micro_batch):
20.    while recMsgNum! =0 do
21.      recvMsgListener()
22.      if recMsgNum == 0 then
23.        last_backword=True
24.      end
25.      if Gi== 0 and micro_batch.size()! =0 then
26.        next_micro_batch=micro_batch.pop()
27.        output=Forward(next_micro_batch)
28.        send(output,Gi+1,j)
29.      else if Gi == inter －1 then
30.        output=Backward()
31.        send(output,Gi－1,j)
32.    end
```

5 实验评估

5.1 实验环境

本文实验环境采用的工具链版本为：PyTorch 1.13.0，OpenMPI 4.1.6 和 DTK 22.10.1。

实验平台为国家超级计算郑州中心的“嵩山”超级计算系统，操作系统为 CentOS7.6，采用处理器＋加速器架构，每个节点上含有 1 个 CPU 处理器和 4 个 DCU 加速器，处理器采用海光一号 CPU，单片上拥有 32 个核心，最多可同时开启 64 个线程，稳定运行可达 3.0GHz 以上。DCU 作为“嵩山”超级计算系统的硬件加速器，Rocm 版本为 DTK 22.10.1，具有 16GB 的全局内存。具体的硬件性能指标如表 1 所列。

表 1 海光一号 DCU 加速器的硬件指标

Table 1 Hardware indicators of Haiguang No.1 DCU accelerator

编号	硬件性能指标	参数
1	Compute Units	64
2	Core Clock Rate	1.6GHz
3	GDDR5 Memory Clock Rate	1000MHz
4	Memory Size	16GB
5	Peak Memeory Bandwidth	1TB/s
6	L2 Cache Size	1024kB
7	Local Data Store per CU	64kB
8	Single Precision Peak Performance	6.5TFlops

5.2 测试集与实验方案

5.2.1 框架选择

本文在 Pytorch 基础上构建出使用自动划分算法和同步

优化算法的测试框架，该框架默认开启自动划分算法，关闭同步优化算法。在接下来的实验中，本文将用术语“基准框架”指代未使用同步优化算法的情况，而“优化框架”将用于描述使用了同步优化算法的情况。为了确定基准框架与现有成熟框架之间的差异程度，本文使用了该基准框架与具备混合并行能力的深度学习训练框架 DeepSpeed 进行对比。选择 DeepSpeed 作为对比框架，主要是考虑到了超算 DCU 架构的特殊性，能够使用混合并行进行分布式训练的框架相对较少。例如，Megatron-LM 是由英伟达开发的，专为英伟达显卡设计，因此在超算 DCU 环境下不太适用。

5.2.2 模型选择

本文采用了不同大小的 GPT-2，AlexNet，VGG16 和 ResNet50 模型进行性能评估，其中 GPT-2 模型可以通过 Transformer 解码器层数、编码层大小和注意力头的数量这 3 个参数来决定模型的完整参数大小。为了验证自动划分算法的通用性，使用 Hugging Face 的 Transformers 库[20]实现 GPT-2 模型和使用 Torchvision 对 AlexNet，VGG16 和 ResNet50 模型进行定义。Transformers 库中的 GPT-2 模型分为 4 个版本：GPT2-SMALL，GPT2-MEDIUM，GPT2-LARGE 和 GPT2-XL，具体细节如表 2 所列。实验首先使用 GPT2-SMALL 模型进行模型训练，从而验证自动划分算法和优化算法对收敛结果没有影响。接着，在不同节点数量上进行 GPT2 模型训练，从而验证优化算法的加速效果。节点选择上，以 2 个节点(8 个 DCU)作为起始的训练，并扩展到 6 个节点(24 个 DCU)。最后，在 1 个节点(4 个 DCU)上训练 AlexNet，VGG16 和 ResNet50 模型，并利用 4 个节点(16 个 DCU)对 GPT 2 不同模型进行训练，以探索模型参数量对优化算法的影响。

表 2 不同 GPT2 模型的细节

Table 2 Details of different GPT2 models

名称	Decoder 层数	编码层大小	注意力头数	参数量
GPT2-SMALL	12	768	12	1.24×10^8
GPT2-MEDIUM	24	1024	16	3.55×10^8
GPT2-LARGE	36	1280	20	7.44×10^8
GPT2-XL	48	1600	25	1.50×10^9

5.2.3 数据集选择

本文使用 Wikitext-103 数据集训练所有 GPT 模型，该数据集是由维基百科中的文本内容构建而成。它包含了维基百科的文章、段落和句子，并涵盖了广泛的主题，包括科学、历史、文学、艺术等，超过了 1 亿个英文单词，一共 22 万多条数据。

针对 AlexNet，VGG16 和 ResNet50 模型，本文使用 CIFAR-10 数据集。该数据集包含有 60000 张彩色图像，共计 10 个类别，每个类别有 6000 张图像。

5.2.4 评价指标

实验使用两个指标进行分析，即训练时间和模型吞吐量。训练时间是在相同训练条件下，模型训练迭代一轮的时间。模型吞吐量为 $samples/sec$，即每秒处理的样本数量：

$$samples/\mathrm{sec}=\frac{batch\_size}{T} \tag{1}$$

其中，$batch\_size$ 为批次大小，即一次训练的样本数量；T 为每个批次所需的训练时间。

5.3 实验结果分析

5.3.1 准确性分析

准确性分析是确保自动划分算法和同步优化算法对模型收敛性不会产生不利影响。在深度学习训练中，损失函数曲线通常被认为是验证训练准确性的一个重要依据，不合理的损失变化可以反映出模型训练过程的错误。为了验证自动划分算法以及同步优化算法的准确性，使用 2 路数据并行和 2 路层间并行训练 GPT2-SMALL 模型，并在其基础上记录使用同步优化算法前后的损失变化情况。图 7 展示了使用同步优化算法前后模型训练的损失变化，可以发现损失变化曲线基本相同，这确定了同步优化算法的准确性。

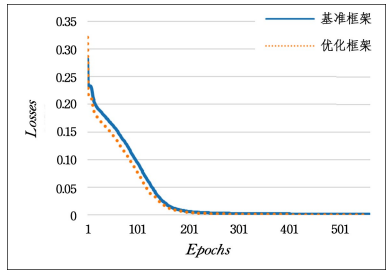


图 7 同步优化算法前后 loss 的变化

Fig. 7 Loss changes before and afterusing synchronous optimization algorithm

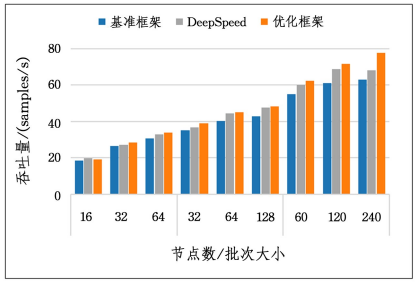
5.3.2 吞吐量分析

图 8 展示了在不同训练配置和训练资源下，GPT2-MEDIUM，GPT2-LARGE 和 GPT2-XL 模型在基准框架、DeepSpeed 和优化框架下的吞吐量变化。与基准框架和 DeepSpeed 的情况相比，优化框架分别可以实现最高 1.23 倍和 1.14 倍的加速效果。

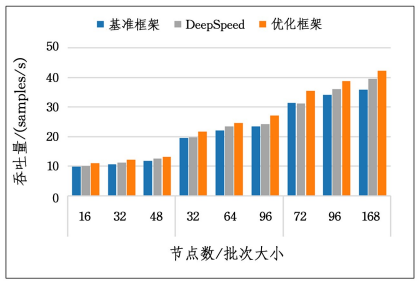
图 8(a)给出了 GPT2-MEDIUM 的训练情况。作为 3 个模型中参数量最小的训练模型，在 2 节点批次大小为 16 以及 32 时，与基准框架相比，其只能实现 1.03 和 1.08 倍的加速效果，这主要是由于此时的微批次大小仅仅被设置为了 1 和 2。由于同步优化算法依赖于最后一个微批次，微批次越大，模型的训练时间越长，此时同步优化算法的效果更加明显，所以在相同节点条件下，批次大小设置为 64，微批次大小设置成 8，与基准框架相比吞吐量可以提升 1.11 倍。在 6 节点、批次大小为 240 的条件下，吞吐量提升 1.23 倍，提升效果最大。与 DeepSpeed 相比，优化框架的加速效果较为有限，最小仅为1.01 倍。在具体条件下，例如 2 节点 16 批次的情况下，由于模型参数较少且微批次大小较小，DeepSpeed 执行的参数更新操作相对较少，因此 DeepSpeed 的加速效果优于优化框架。

图 8(b)给出了 GPT2-LARGE 的训练情况。GPT2-LARGE 的性能趋势与 GPT2-MEDIUM 相似。优化框架相对于 DeepSpeed，能够实现 1.05～1.14 倍的加速。在 2 节点、4 节点和 6 节点情况下，优化框架的吞吐量分别是基准框架的1.11～1.13 倍、1.11～1.15 倍和 1.13～1.18 倍。与图 8(a)对比可以发现，在 2 节点优化前后训练 GPT2- LARGE

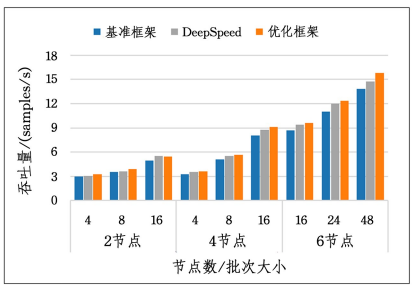
时吞吐量数值减小，但是加速效果大于 GPT2-MEDIUM。主要原因在于模型参数量增加并且数据集大小不变，导致吞吐量下降，而在加速效果上，由于参数量增加，节点间需要传输更多的参数数据，同步优化算法的整体效果会随着通信量的增加更加明显。



(a)GPT2-MEDIUM



(b)GPT2-LARGE



(c)GPT2-XL

图 8 不同节点数量与不同批次大小下模型训练吞吐量的变化

Fig. 8 Changes of throughput with different number of nodes and different batch sizes

图 8(c)给出了 GPT2- XL 的训练情况。DeepSpeed 的训练结果与 GPT2-MEDIUM 条件下相似，在较小微批次情况下，DeepSpeed 的优化效果与优化框架基本相当。然而在 6 节点、48 批次情况下，优化框架的加速效果明显超过了 DeepSpeed，实现了最高 1.08 倍的加速效果。值得一提的是，观察 4 节点、16 批次与 6 节点、16 批次可以发现，与基准框架相比，优化框架的吞吐量加速效果由 1.11 降为 1.12。批次大小不变且节点数量增加，导致微批次数量减少并且第一个计算设备获取到的模型分片参数量变小，使得通信参数量减少，性能提升降低。因此在未来的工作中，将进一步针对模型自动划分问题进行优化，使得模型划分更加合理。

观察图 8 模型的吞吐量情况可以发现，在模型相同时，随着节点数量的增加，同步优化算法的提升效果逐渐改善，例如 GPT2- XL 由 1.08 倍的加速效果提升至 1.14 倍。这表明，同步优化算法在更大的集群规模中的表现得更为显著。

5.3.3 训练时间分析

为了进一步确定模型参数量对同步优化算法的影响，使用 GPT 2 的 4 个模型在 4 节点、相同批次、相同微批次条件下进行训练测试，观察使用优化算法前后训练一轮的时间开销，如图 9 所示。图 9 中，蓝色为基准框架训练一轮的时间，灰色为 DeepSpeed 训练一轮的时间，橙色为优化框架训练一轮的时间，而黄色折线为模型优化前后训练加速比情况。从图中可以明显看出，使用同步优化算法可以减少训练一轮的时间，同步优化算法将 GPT2-XL 模型训练一轮的时间由 7.75 h 减少到 6.745 h，缩短接近 13% 的时间。另外，随着模型参数量的增加，加速比曲线呈上升趋势，表明同步优化算法随着模型参数量的增加将具有更加显著的优势。

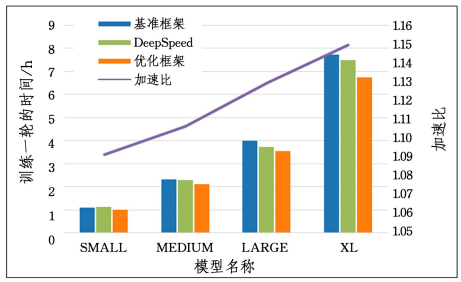


图 9 4 节点相同批次大小下模型训练时间及加速比(电子版为彩图)

Fig. 9 Model training time and acceleration ratio with 4 nodes at the same batch size

此外，本文使用了 4 个 DCU 对 AlexNet，VGG16 和 ResNet50 模型进行训练对比，观察在基准框架、DeepSpeed 和优化框架下训练 20 轮次的时间开销，具体结果如图 10 所示。

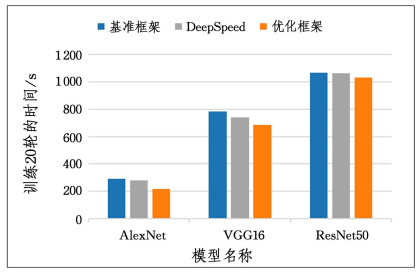


图 10 4 个 DCU 条件下训练 20 轮的时间开销

Fig. 10 Time cost of training 20 epochs with 4 DCU

相较于基准框架，DeepSpeed 确实带来了一定程度的优化。但相较于优化框架，其优化效果有限，在 VGG16 上优化框架可以缩短约 12.64% 的训练时间，而 DeepSpeed 仅能缩短约 5% 的训练时间。此外，在 ResNet50 模型上，优化框架比基准框架仅能减少 3% 左右的训练时间，这一加速效果相较于 AlexNet 和 VGG16 模型较小。这主要是由于 ResNet50 属于计算密集型模型，计算任务占据了主要部分，而 AlexNet 和 VGG16 为通信密集型模型，其中通信时间相对较长[21]。因此，ResNet50 使用同步优化算法优化通信后的效果逊于 AlexNet 和 VGG16 模型。

结束语 本文面向混合并行的分布式训练，研究了层间并行的模型划分方式，并实现了模型的自动划分。另外，通过改进通信方式解决了层间并行和数据并行结合的通信问题，

实现了计算和通信的重叠。该优化方式在 GPT-2，AlexNet，VGG16 和 ResNet50 模型训练中取得了显著的成果，实现了最高 1.23 倍的加速比。但是，同步优化算法主要致力于解决层间并行中通信和计算不充分重叠的问题，而对层内并行尚未进行深入研究。未来的研究将着眼于层内并行中通信的优化，以提高训练效率。此外，对于模型自动划分的方式，现有方法划分标准过于单一，导致模型在使用同步优化算法后的加速性能不够稳定，未来也将进一步探索更为精细的划分标准，以加快模型整体训练的流程。

参 考 文 献

[1] SHOEYBI M, PATWARY M, PURI R, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism[J]. arXiv: 1909.08053, 2019.

[2] AO Y, WU Z, YU D, et al. End-to-end adaptive distributed training on paddlepaddle[J]. arXiv: 2112.02752, 2021.

[3] RASLEY J, RAJBHANDARI S, RUWASE O, et al. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters[C]// Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020: 3505-3506.

[4] SERGEEV A, DEL BALSO M. Horovod: fast and easy distributed deep learning in TensorFlow[J]. arXiv: 1802.05799, 2018.

[5] GAN S, JIANG J, YUAN B, et al. Bagua: scaling up distributed learning with system relaxations[J]. Proceedings of the VLDB Endowment, 2021, 15(4): 804-813.

[6] LI, ZHAO S A, VARMA Y A, et. al. Pytorch distributed: Experiences on accelerating data parallel training[J]. VLDB Endowment, 2020, 13(12): 3005-3018.

[7] SINGH S, BHATELE A. AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning[C]// 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2022: 606-616.

[8] SONG J, YIM J, JUNG J, et al. Optimus-CC: Efficient Large NLP Model Training with 3D Parallelism Aware Communication Compression[C]// Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 2023: 560-573.

[9] JEAUGEY S. Nccl 2.0[C]// GPU Technology Conference (GTC). 2017.

[10] GABRIEL E, FAGG G E, BOSILCA G, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation [C]// Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11. Springer Berlin Heidelberg, 2004: 97-104.

[11] COATES A, HUVAL B, WANG T, et al. Deep learning with COTS HPC systems[C]// International Conference on Machine Learning. PMLR, 2013: 1337-1345.

[12] WANG E D, YAN R D, GUO Z H et al. Review of distributed training systems and their optimization algorithms [J]. Chinese Journal of Computers, 2024, 47(1): 1-28.

[13] NARAYANAN D, HARLAP A, PHANISHAYEE A, et al. PipeDream: Generalized pipeline parallelism for DNN training [C]// Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 1-15.

[14] BROWN T, MANN B, RYDER N, et al. Language models are few-shot learners[J]. Advances in Neural Information Processing Systems, 2020, 33: 1877-1901.

[15] PATARASUK P, YUAN X. Bandwidth optimal all-reduce algorithms for clusters of workstations[J]. Journal of Parallel and Distributed Computing, 2009, 69(2): 117-124.

[16] ZHENG S, MENG Q, WANG T, et al. Asynchronous stochastic gradient descent with delay compensation[C]// International Conference on Machine Learning. PMLR, 2017: 4120-4129.

[17] ZHANG H, ZHENG Z, XU S, et al. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters[C]// 2017 USENIX Annual Technical Conference. 2017: 181-193.

[18] REED J, DEVITO Z, HE H, et al. Torch.fx: Practical program capture and transformation for deep learning in python[J]. Proceedings of Machine Learning and Systems, 2022, 4: 638-651.

[19] RADFORD A, WU J, CHILD R, et al. Language models are unsupervised multitask learners[J]. OpenAI blog, 2019, 1(8): 9.

[20] WOLF T, DEBUT L, SANH V, et al. Transformers: State-of-the-art natural language processing[C]// Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. 2020: 38-45.

[21] XU Y, DONG D, XU W, et al. SketchDLC: A sketch on distributed deep learning communication via trace capturing[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2019, 16(2): 1-26.

XU Jinlong, born in 1985, Ph.D, master's supervisor. His main research interests include high-performance computing and paralle compilation.

HAN Lin, born in 1978, Ph.D, associate professor, is a senior member of CCF (No. 16416M). His main research interests include compiler optimization and high-performance computing.

（责任编辑：柯颖）