SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient

摘要

随着深度学习模型参数量的不断增大，模型的训练成本通常非常昂贵，而且这些模型的规模已经超过了单个加速器的处理能力，通常需要专门的高性能计算（HPC）集群。为了寻找更经济的替代方案，文章提出了 SWARM 并行算法，这是一种针对性能较差、异构且不可靠的设备设计的模型并行训练算法。现在常见的模型并行算法通常需要在设备间进行密集的通信，如果单个设备发生故障，整个训练过程可能会中断。而本文的目标是找到一种实用的方法，使用不可靠、异构的设备训练大型神经网络，这些设备具有较慢的互联网连接。研究人员通过分析模型大小对管道并行训练中通信和计算成本之间的平衡的影响，提出了 SWARM 并行算法。该算法旨在处理节点故障，通过优先选择具有较低延迟的稳定节点，以及定期重新平衡管道阶段，从而适应具有不同硬件和网络速度的设备，最终取得了不错的效果。

我将对本文提出的 SWARM 并行算法展开复现工作。

关键词：分布式机器学习；流水线并行

1 引言

随着人工智能领域的高速发展，各类模型的参数量与日俱增，然而大多数研究者的硬件条件并没有得到质的改善。那么长远看来，对于大模型研究的门槛只会越来越高，并不是所有研究者都具有大规模的服务器集群和丰富的算力资源，这在一定程度上会限制大模型训练的可及性。

而本文立足于改善这一情况，基于分布式机器学习领域为研究者提供一个大模型的廉价训练方案，显著降低了对昂贵硬件的依赖，使得更广泛的研究团队能够参与到大型模型的训练中来，这毫无疑问是十分有意义的一项工作。 [5]

另一方面，与本文有相同出发点的一些传统模型并行算法也存在一些缺陷。例如不能充分利用所有计算资源，常会面临异构硬件带来的“短板效应”。而本文提出的方法能够更有效地利用“可抢占”的实例和分散的资源，这些通常比传统的专用计算资源更经济，充分提高了计算资源的利用率。

基于此，选择本文进行复现，方向上符合社会需求，也符合我自己的研究方向；同时，本文具体的代码工作也有助于我对分布式机器学习领域建立基础的知识框架，丰富实践经验，为日后在该方向进一步的科研工作打好基础。

2 相关工作

2.1 模型并行训练

模型并行训练大体上可分为两种方法，一种是传统的模型并行，另一种是流水线并行。

在传统模型并行中，每个处理单元负责计算模型的一部分。例如，在一个深度神经网络中，不同的神经元或层的参数可能分布在不同的处理单元上。这种方法通常需要在每一层的计算之后进行数据的通信，因此，如果参数量过大，这种方式带来的通信开销是巨大的。

流水线并行，则是将模型的各个分层放到不同设备上，每个处理单元负责模型的一段。这类似于生产线，数据在模型的不同段之间流动。流水线并行减少了设备间的通信需求，因为每个处理单元仅在处理完其部分后将数据传递给下一个处理单元，仅涉及不同阶段的通信。但是传统流水线并行通常会带来较多的 bubble 时间，计算资源的利用率不够高。如 google 提出的 gpipe [3]。以及微软提出的 pipedream [2], 都是基于流水线并行的典型算法。

除此之外，NVIDIA 曾提出张量并行的方法 [6]，不同于传统模型并行和流水线并行的层间并行，该方法用于实现模型并行的层内并行的计算。

2.2 非 HPC 环境下的分布式训练

传统的模型并行训练技术是为具有快速可靠通信的同质设备群体设计的，适合高性能计算（HPC）环境。然而，这种基础设施并非总是可用，因此更经济的替代方案是使用“可抢占”的计算实例节点。这些环境对分布式训练提出了更多挑战，例如机器可能因故障或抢占而突然断开连接。

3 本文方法

3.1 本文方法概述

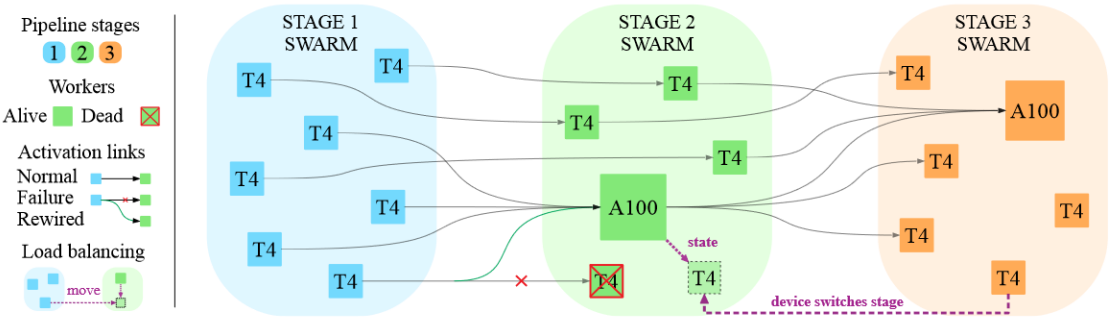

图 1. 方法示意图

上图是本文所提出的 swarm 并行算法的核心实现过程。swarm 并行算法所是基于流水线并行所实现的，但与传统的流水线并行算法如 Gpipe 和 Pipedream 不同，swarm 所使用的是动态的随机管道策略。

具体来说，本文中描述的算法的核心实现流程可以概括如下：1. 该算法在管道并行设置中运行，其中训练过程涉及多个阶段，每个阶段是模型的部分分层。2. 在每个阶段，算法写

入与当前阶段关联的分布式哈希表 (DHT)，从而发现服务于该阶段的所有计算节点。3. 每个阶段还同时维护基于设备性能队列，算法进入一个阶段时，将使用该阶段队顶的设备对该阶段进行训练。4. 一个阶段训练结束后，算法将基于设备的训练时间对各个阶段的设备性能队列进行更新。5. 当系统处于活动状态时，算法继续进入下一阶段，迭代上述步骤，直至训练完成。该算法的目的是根据从请求队列大小获得的负载信息定期重新平衡计算节点，使各个计算节点在不同阶段灵活调度，负载高的阶段多分配节点，负载少的阶段少分配节点，从而平衡各个阶段的负载，实现动态平衡。

4 复现细节

4.1 与已有开源代码对比

本文在 github 所开源的源代码中，禁用了 wandb 库函数，使其不能在 weight and bias 网站上实施跟踪监测实验的相关数据信息。wandb 是 Weight and Bias 的缩写，这是一个与 Tensorboard 类似的参数可视化平台。不过，相比较 TensorBoard 而言，Wandb 更加的强大。而在本次复现中，实验所涉及到的数据众多，包括实验本身的各类参数、损失以及 GPU 的相关信息。基于此，我修改了源代码，在全局开启了 wandb 库的追踪，并在相应的数据接口上引入了相应代码。最终代码可实现在远程实现监控实验相关信息，并且可以可视化各类数据信息及其动态变化过程。

由于本文所提出的是一个基于分布式机器学习的并行计算方案，涉及到众多不同设备的相关信息，wandb 库的相关实现有助于更完整地对整个实验进行性能分析和对比，丰富实验细节。

同时，由于本项目的在 github 上的代码库还在持续更新中，相关的代码和脚本文件完整性还有待充实。因此，基于我在复现中遇到的一些问题，我也对一小部分代码以 shell 文件进行少部分更改，使其成功运行。

4.2 实验环境搭建

在本次复现的环境配置方面，采取了控制端与数据端分离的方法，总共需要配置三个环境。一是用于监测实验情况的 monitor 程序环境，二是用于进行实际训练的 GPU 上的环境，三是用于指挥训练流程的 trainer 程序环境。

首先，对于 monitor 程序的环境配置。它的最主要作用有两点。第一就是开启 wandb 库函数对本次实验的监控接口，实现 monitor 监管程序的本职工作。另一方面，该程序同时开启一个基于分布式哈希表 DHT 的初始对等节点，使的后面开启的 GPU 训练节点能与这个初始对等节点发起连接，组成一个完整的 DHT 训练网络，这也是本文底层核心的网络架构。

另一方面，对于 GPU 上的环境配置。这是真正进行训练所用到的相关环境，因此在 GPU 上需要安装 pytorch 相关的用于分布式训练和并行计算的库。同时，实际启动这些 GPU 节点时，需要指定要连接的初始 DHT 对等节点的 IP 地址，也就是在 monitor 程序所生成的节点地址。

最后，是关于 trainer 程序环境的配置。上面两个环境相当于已经配置好了底部数据端的所有相关环境，而 trainer 程序所负责的就是控制端的环境配置，指挥整个项目进行运转。

trainer 程序会指定训练相关的超参数和基本信息，同时在合适的阶段选择不同的 GPU 训练节点来参与工作，实现整个 swarm 并行算法的全流程的控制。

综上，monitor 程序环境以及 GPU 环境组成整个项目的底层数据端，并建立一个完整的 DHT 网络作为实验的网络架构。而 trainer 程序作为实验顶层的控制端，指挥数据端的相关操作和数据流动。由此实现整个算法的运转。如下所示：

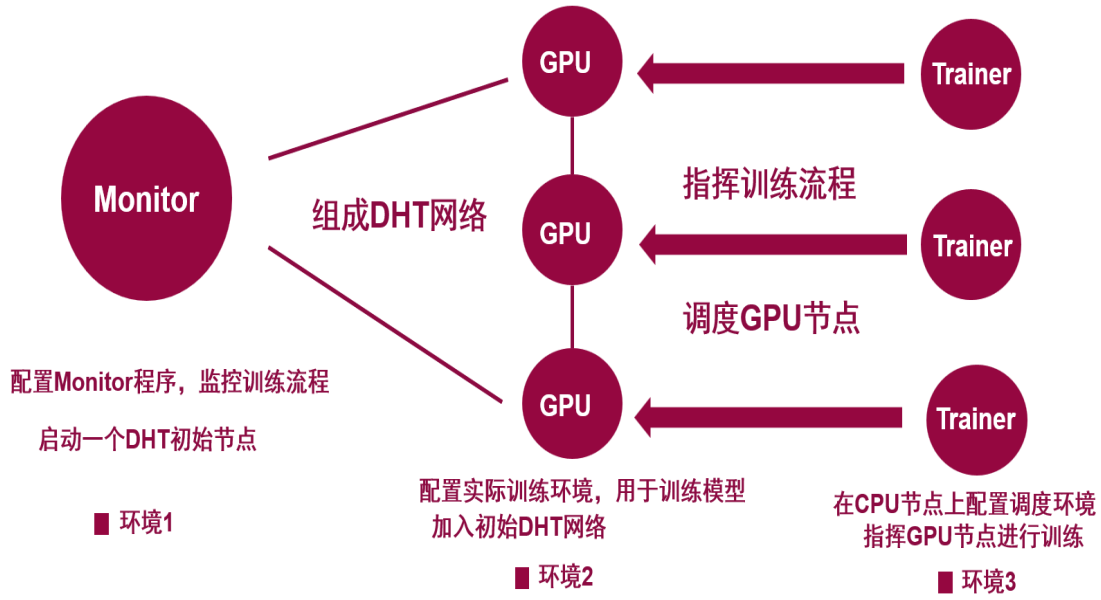

图 2. 环境配置示意图

4.3 算法细节

input peer index i, current peer stage s_{cur}, total number of stages S, rebalancing period T

while active **do**
  Sleep for T seconds
  Measure q_i as the local request queue size
  Write (i, q_i) as the key-subkey pair to DHT[s_{cur}]
  Initialize minimum and maximum load stages: $s_{min} = s_{max} := -1$,
  $l_{min} := \infty, l_{max} := -\infty$
  for s in $1, \ldots, S$ **do**
    Initialize the load buffer $L = 0$
    for (j, q_j) in DHT[s] **do**
      $L := L + q_j$
    end for
    if $L > L_{max}$ **then**
      $s_{max} := s,\ L_{max} := L$
    end if
    if $L < L_{min}$ **then**
      $s_{min} := s,\ L_{min} := L$
    end if
  end for
  if $s_{cur} = s_{min}$ **then**
    // Migrate to the maximum load stage
    Initialize the minimum load peer $i_{min} := -1, q_{min} := \infty$
    for (j, q_j) in DHT[s] **do**
      if $q_j < q_{min}$ **then**
        $i_{min} := j,\ q_{min} := q_j$
      end if
    end for
    if $i_{min} = i$ **then**
      // This peer should migrate
      $s_{cur} := s_{max}$
      Download up-to-date parameters from peers in s_{max}
    end if
  end if
end while

图 3. 算法伪代码

我所要复现目标算法的核心代码如上图所示。具体来说，主要可以分为以下几个流程和步骤：

（1）初始化：添加所有可用于训练的计算节点，建立优先级队列，其中每个 stage 对应一个队列

（2）定义相关函数：基于优先级参数定义了禁用节点和选择服务节点的函数，用于后续使用

（3）前向传播：按顺序遍历每个流水线阶段，进入一个 stage 读入该流水线阶段对应的优先级队列，选择对应的设备用于处理该任务

（4）更新队列：根据处理的任务，基于 EMA 更新各个设备的性能，从而更新流水线阶段的优先级队列，从而在下一次选择时能发生动态变化

在实际运行中，首先，我会先启动代码目录文件下的 start-monitor.py 文件，此时程序文件会产生一个 DHT 的初始对等节点地址，此时记录这个 p2p 地址，如下图所示；

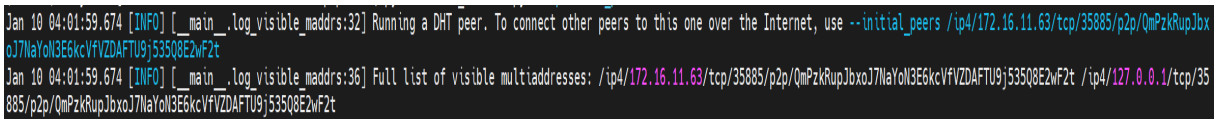

图 4. 监管程序运行示意图

第二步，在两个计算节点上分别启动目录文件下的 server 脚本，并同时指定脚本所使用的 initial-peers 为第一步所产生的 p2p 地址，如下所示；

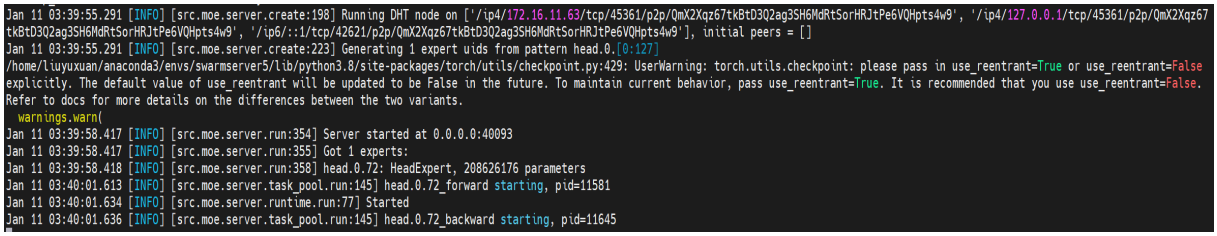

图 5. 训练节点运行示意图

最后，在 cpu 节点上启动 trainer 脚本，此时也同样需要指定该脚本的 initial-peers 为第一步所产生的 p2p 地址，此时，得以使分别承担不同功能的三种类型的节点加入同一个分布式网络中，利用分布式哈希表来协调管理，使得训练能顺利进行。

5 实验结果分析

本次复现由于实验条件有限，只选择两块 RTX3090 的显卡来进行实验，实验模型为 GPT2，训练所用数据集为 The Pile 数据集,The Pile 是一个大型、多样化的开源语言建模数据集，由许多较小的数据集组合而成。其目的是从尽可能多的模式中获取文本，以确保使用 The Pile 训练出来的模型具有更广泛的泛化能力。

将两个 rtx3090 作为训练节点加入训练，完成整个训练流程。相应实验数据在 wandb 上记录：

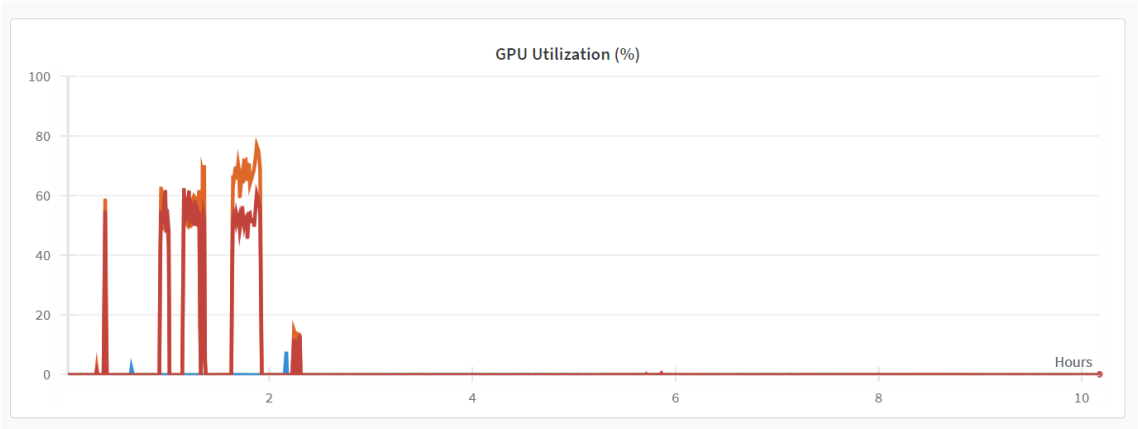

图 6. 实验过程 GPU 使用率

如上图所示，橙线和红线分别代表两个显卡。可知两块显卡在训练过程中基本能实现 swarm 算法所要求的并行性，使得两块显卡能同时服务于模型的训练。但受限于只有两块显卡，其中存在不小的空白时间，即类似于 gpipe 中提到的 bubble time，这些时间主要是用于相应权重、参数、梯度信息的 I/O 加载时间，造成 gpu 在实际训练时的堵塞。

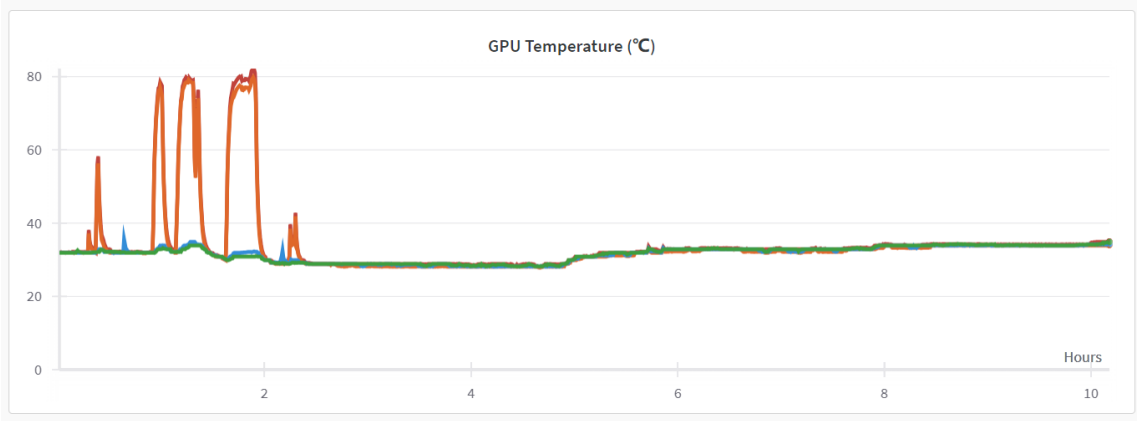

图 7. 实验过程 GPU 温度

同样，对于上图所示的 GPU 的温度信息，同样对应橙线与红线，其与图 6 中 GPU 使用率互相呼应，可反应两块 GPU 在训练过程中的使用情况。

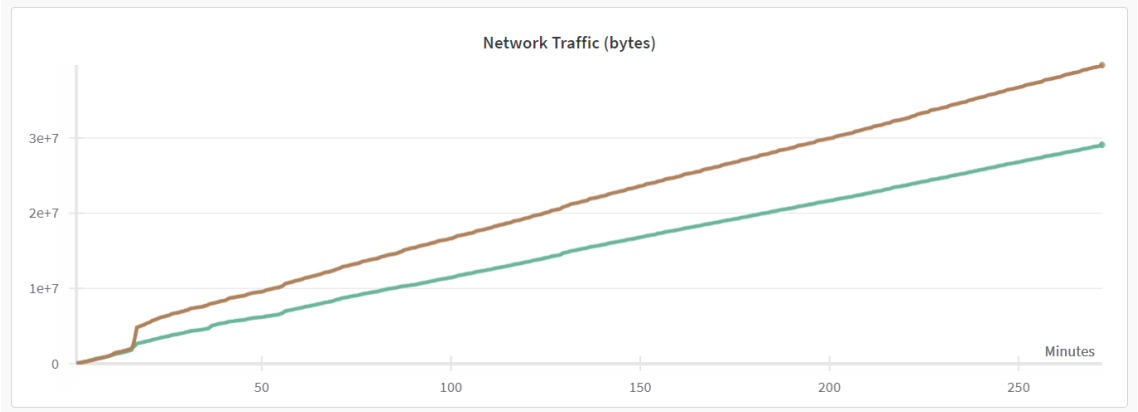

图 8. GPU 网络情况

最后是关于两块显卡的网络通信情况。由图 8 可知，两块 GPU 在训练全程能保持良好的通讯，反应其在不同流水线阶段时，计算节点可以较为通顺的交换不同阶段所要用到的参数信息。

根据以上分析可知，这两块 GPU 在此次模型的训练过程中充当两个训练节点，且能根据 SWARM 并行算法的要求并行地完成模型的训练，并在训练全程保持良好的通信情况。

但从图 6、图 7 中可以看出两块 GPU 在实际训练过程中具有不少的阻塞时间，没有完全发挥出理论上能达到的流水线并行效率。这可能是因为受限于只用了两块 GPU 训练，相应的流水线阶段无法很好地完成折叠。

6 总结与展望

综上所述，本次复现在有限的实验条件下，利用两块 GPU 复现了文中所提出的 SWARM 并行性算法。但从实验分析可知，SWARM 并行算法在实际应用中的效果与实际的参数、实际的实验配置（例如使用几块 GPU、将训练分为几个阶段等）都息息相关，因此该算法对于某些特定条件下的分布式训练环境可能具有不小的局限性。就以本次复现为例，仅仅使用两块 GPU，难以将整个模型的流水线完美运行起来，有很多的运行时间难以压缩。

另一方面，本次复现工作让我对 SWARM 算法有了一个较为深入的了解。但我在本次的复现改进工作只涉及对于源码和部分脚本文件的修改，以及最终数据可视化部分的呈现。并未触及到对算法内部细节的优化和修改。对于未来的工作，我会尝试深入挖掘算法内部细节，进一步考察是否有可以优化的部分。

未来，随着深度学习领域的不断发展，势必会有越来越多的大模型出现在人们视野，同时模型的参数量也势必会与日俱增。因此，分布式机器学习这一领域的研究势必会有越来越大的实用价值。而对于未来可优化的部分以及研究方向，可主要考虑以下几点：

（1）. 针对于实际训练的应用，不使用单一的 SWARM 算法，而是可以尝试和微软的 DeepSpeed 框架 [1] 以及 Pytorch 框架的 DDP（Distributed Data parallelism） [4] 相结合，充分折叠模型训练过程中的阻塞时间

（2）. 沿着分布式机器学习这一方向深入，SWARM 并行性算法所基于的是分布式学习中的流水线并行技术，未来可尝试与数据并行、张量并行等技术结合，以应对更多的下游应用场景

（3）. 分布式训练只是深度学习训练、推理加速的其中一个方向，可考虑将 SWARM 并行性算法与训练加速、推理加速的更多方向所结合，例如模型剪枝、量化压缩、KV cache 等技术

参考文献

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

[2] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

[3] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[4] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[5] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. SWARM parallelism: Training large models can be surprisingly communication-efficient. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 29416–29440. PMLR, 23–29 Jul 2023.

[6] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.