分布式系统与云计算调研报告：Ray 分布式计算框架

刘润, 王云鹏, 王灏洋 *

1 Ray 的相关背景

Ray 是 University of California, Berkeley (UCB) 构建的机器学习分布式框架。UCB 的教授 Ion Stoica 写过一篇文章《The Future of Computing is Distributed》[1]，里面详细讲述了 Ray 产生的缘由。在这篇文章中，作者提出当前程序面临的两大问题：一是“摩尔定律”的终结。摩尔定律指出，处理器的性能每 18 个月就会翻一番。然而在今天，处理器性能的增长率仅为 10-20%。由于程序对计算机性能的需求一直在增加，所以为了应对这一挑战，计算机架构师将注意力集中在构建特定领域处理器上，这将牺牲处理器的通用性；二是深度学习应用程序对算力的需求正在以惊人的速度增长。深度学习模型在训练环节、调参环节、模拟环节（深度强化学习）都会对算力有着很高的要求，如 OpenAI 在研究中发现，自 2012 年以来，实现最先进的机器学习结果所需的计算量大约每 3.4 个月翻一番 [2]，这已经远超摩尔定律的增长速度。

现如今，大数据和人工智能正迅速发展，程序所需的算力与硬件计算能力之间的差距也逐渐增大。为了弥补这一差距，一项有前景的方案是将这些计算任务分发给不同的主机，这就需要新的软件工具、框架，而 Ray 就是为此而生。

Ray 包含以下四个主要库：*Tune*(超参数调优)、*RLlib*(强化学习)、*RaySGD*(分布式机器学习模型训练) 和 *Datasets*(分布式数据加载)。我们着重讲述有关强化学习的库 Ray[*RLlib*]，这是因为 Ray 的优势与当前部署分布式强化学习应用所面临的困难有着高契合度，更能体现出 Ray 的优势。

强化学习 (Reinforcement Learning, RL) 不同于普通的监督学习，越来越多的强化学习应用程序必须在动态环境中运行，对环境变化作出反应，并采取一系列行动来实现长期目标，目标不仅是利用收集到的数据，而且还要探索可能采取行动的空间。比如通过学习用户操作手机的习惯，学习用户打开应用程序之间的关联性，就可以在一定条件下，向用户适时推荐用户可能想要使用的应用程序，而这种推荐的内容，就是一种行动空间集合中的元素。强化学习是一种基于延迟和有限反馈来学习在不确定环境中持续运行的方法。基于强化学习的系统已经取得了显著的成果，如谷歌的 AlphaGo 击败了人类世界冠军 [3]。

支持 RL 的应用程序应当满足 RL 的以下需求：

• RL 系统必须支持细粒度计算。比如在与物理世界交互时，强化学习产生的动作以毫秒级呈现，并能够执行大量的模拟。

• RL 系统必须能够支持异构性。因为执行强化学习的设备可能是算力很强的设备，在毫秒内就能完成模拟，而在算力很弱的设备上可能要消耗几个小时。

• RL 系统必须支持动态执行，因为模拟的结果或与环境的交互可能会改变未来的计算。

基于以上需求，我们需要一个动态计算框架，以毫秒级的延迟每秒处理数百万个异构任务。而 Ray

*三人同等贡献。刘润负责主笔第 1 小节，王云鹏负责主笔第 2 小节，王灏洋负责主笔第 3 和第 4 两个小节。

就满足这样的特点，以下几节我们将分别介绍 Ray 的基本结构、Ray 的编程模型与计算模型、Ray 的性能评估以及 Ray 的未来发展。

2 Ray 的基本结构

Ray 的基础结构可以参考这篇 arXiv 论文 [4]。在论文中，作者总结了该项工作的主要贡献：

• 设计并构建了第一个整合训练强化学习算法并提供强化学习服务的分布式框架。

• 为了支持工作负载，作者将执行者 (actor) 和任务并行抽象统一在一个动态任务执行引擎之上。

• 为了实现可扩展性和容错性，作者提出了一个系统设计原则，其中控制状态存储在一个分片的元数据存储中，所有其他系统组件都是无状态的。

• 为了实现可扩展性，作者提出一种自底向上的分布式调度策略。

图1展示了 Ray 的系统架构。

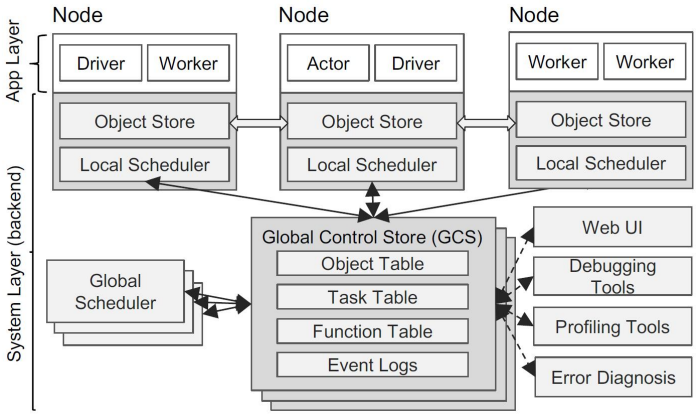


图 1: Ray 的系统架构

Ray 的架构包括:

(1) 实现 API 的应用层;

(2) 提供高可伸缩性和容错能力的系统层。

2.1 Ray 的应用层

应用层由三种类型的进程组成：Driver，Worker 和 Actor。

• Driver: 执行用户程序的进程。

• Worker: 用于处理任务的无状态进程，由 Driver 或其它的 Worker 调用。系统层可以自动唤醒 Worker 并分配任务。当声明一个远程函数时，该函数会自动发布给所有的 worker。

• Actor: 用于处理任务的有状态进程，与 Worker 一样，Actor 是串行执行任务的，除了每个任务都依赖于前一个任务执行所产生的状态。

2.2 Ray 的系统层

系统层由三个主要组件组成: 全局控制存储单元、分布式调度程序和分布式对象存储。所有组件都是可伸缩的和容错的。

2.2.1 全局控制存储单元 (Global Control Store, GCS)

全局控制存储 (GCS) 维护系统的整个控制状态。GCS 的核心是一个具有发布功能的键值存储单元。GCS 及其设计的主要目的是为一个每秒可以动态生成数百万个任务的系统保持容错和低延迟。节点故障时的容错需要一个解决方案来维护沿袭信息。现有的基于谱系的解决方案专注于粗粒度并行性，因此可以使用单个节点（例如，主节点、驱动程序）来存储谱系而不影响性能。但是，对于像模拟这样的细粒度和动态工作负载，这种设计不可扩展。因此，Ray 将持久沿袭存储与其他系统组件分离，允许每个组件独立扩展。

保持低延迟需要最大限度地减少任务调度的开销，这涉及选择在哪里执行，以及随后的任务分派，这涉及从其他节点检索远程输入。许多现有的数据流系统通过将对象位置和大小存储在一个集中的调度器中来耦合这些，当调度器不是瓶颈时，这是一种自然的设计。然而，Ray 所针对的规模和粒度要求将集中式调度器远离关键路径。对于像 allreduce 这样的分布式训练很重要的原语，在每个对象传输中都涉及调度程序是非常昂贵的，这既是通信密集型又对延迟敏感。因此，我们将对象元数据存储在 GCS 中而不是调度器中，将任务调度与任务调度完全解耦。

总之，GCS 显着简化了 Ray 的整体设计，因为它使系统中的每个组件都是无状态的。这不仅简化了对容错的支持（即，在出现故障时，组件只需重新启动并从 GCS 读取沿袭），而且还可以轻松地独立扩展分布式对象存储和调度程序，因为所有组件共享所需的状态通过 GCS。另一个好处是可以轻松开发调试、分析和可视化工具。

2.2.2 自下而上的分布式调度器

图2展示了 Ray 的自下而上的分布式调度器。

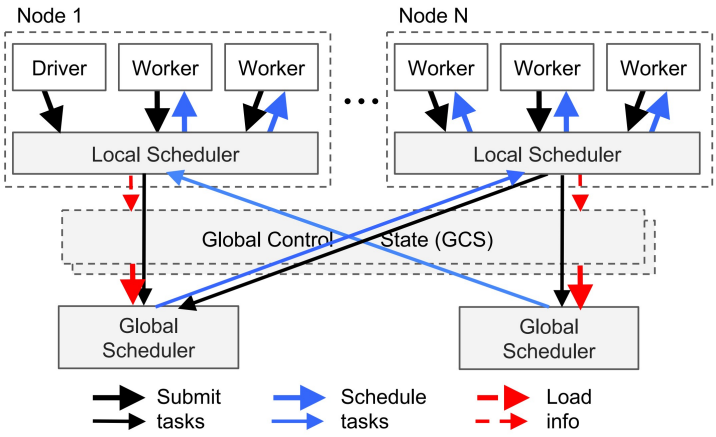


图 2: 自下而上的分布式调度器

Ray 需要每秒动态调度数百万个任务，这些任务可能只需要几毫秒。我们所知道的集群调度器都没有满足这些要求。大多数集群计算框架，如 Spark[5] 都实现了一个集中式调度器，它可以提供局部性，但延迟为几十毫秒。

为了满足上述要求，我们设计了一个由全局调度器和每节点本地调度器组成的两级分层调度器。为了避免全局调度器过载，在节点创建的任务首先提交到节点的本地调度器。本地调度器在本地调度任务，除非节点过载（即，其本地任务队列超过预定义的阈值），或者无法满足任务的要求（例如，缺少 GPU）。如果本地调度程序决定不在本地调度任务，则将其转发给全局调度程序。由于这个调度器首先尝试在本

地调度任务（即，在调度层次结构的叶子），我们称它为自底向上调度器。

全局调度器会考虑每个节点的负载和任务的约束来做出调度决策。更准确地说，全局调度器识别具有任务所请求类型的足够资源的节点集，并且在这些节点中选择提供最低估计等待时间的节点。在给定节点，这个时间是 (i) 任务在该节点排队的估计时间（即任务队列大小乘以平均任务执行时间）和 (ii) 任务远程的估计传输时间的总和。输入（即远程输入的总大小除以平均带宽）。全局调度器通过心跳获取每个节点的队列大小和节点资源可用性，以及任务输入的位置及其来自 GCS 的大小。此外，全局调度器使用简单的指数平均计算平均任务执行和平均传输带宽。如果全局调度器成为瓶颈，我们可以通过 GCS 实例化更多共享相同信息的副本。这使得我们的调度器架构具有高度可扩展性。

2.2.3 内存分布式对象存储

为了最小化任务延迟，我们实现了一个内存分布式存储系统来存储每个任务或无状态计算的输入和输出。在每个节点上，我们通过共享内存实现对象存储。这允许在同一节点上运行的任务之间进行零拷贝数据共享。作为数据格式，我们使用 Apache Arrow[6]。

如果任务的输入不是本地的，则在执行之前将输入复制到本地对象存储。此外，任务将其输出写入本地对象存储。复制消除了由于热数据对象导致的潜在瓶颈，并最大限度地减少了任务执行时间，因为任务仅从本地内存读取数据/向本地内存写入数据。这增加了计算密集型工作负载的吞吐量，这是许多 AI 应用程序共享的配置文件。为了低延迟，我们将对象完全保留在内存中，并根据需要使用 LRU 策略将它们驱逐到磁盘。

与现有的集群计算框架（如 Spark[5]）一样，对象存储仅限于不可变数据。这消除了对复杂一致性协议的需要（因为对象没有更新），并简化了对容错的支持。在节点故障的情况下，Ray 通过 lineage 重新执行来恢复任何需要的对象。存储在 GCS 中的谱系在初始执行期间跟踪无状态任务和有状态参与者；我们使用前者来重建商店中的对象。

为简单起见，我们的对象存储不支持分布式对象，即每个对象适合单个节点。像大型矩阵或树这样的分布式对象可以在应用程序级别实现为期货的集合。

2.3 Ray 的加法示例

图3展示了 Ray 添加 a 和 b 并返回 c 的端到端示例。图3说明了 Ray 如何通过一个简单的示例实现端到端的工作，该示例将两个对象 a 和 b 可以是标量或矩阵）相加，并返回结果 c。远程函数 add() 在初始化时自动注册到 GCS 并分发给系统中的每个工作人员。

图3a 显示了由驱动程序调用 add.remote(a,b) 触发的逐步操作，其中 a 和 b 分别存储在节点 N1 和 N2 上。驱动程序将 add(a,b) 提交给本地调度程序（步骤 1），本地调度程序将其转发给全局调度程序（步骤 2）。接下来，全局调度程序查找 add(a,b) 的位置，在 GCS 中的参数（步骤 3）并决定在节点 N2 上安排任务，该节点存储参数 b（步骤 4）。节点 N2 的本地调度程序检查本地对象存储是否包含 add(a,b) 的参数（步骤 5）。由于本地商店没有对象 a，它会在 GCS 中查找 a 的位置（步骤 6）。得知 a 存储在 N1，N2 的对象存储在本地复制它（步骤 7）。由于 add() 的所有参数现在都存储在本地，因此本地调度程序在本地工作人员处调用 add()（步骤 8），它通过共享内存访问参数（步骤 9）。

图3b 显示了由分别在 N1 处执行 ray.get() 和在 N2 处执行 add() 触发的逐步操作。在调用 ray.get(idc) 时，驱动程序使用 add() 返回的未来 idc（步骤 1）检查本地对象存储中的值 c。由于本地对象存储不存储 c，它在 GCS 中查找它的位置。此时，没有 c 的条目，因为尚未创建 c。结果，N1 的对象存储向对象表注册了一个回调，以便在创建 c 的条目时触发（第 2 步）。同时，在 N2，add() 完成其执行，将结果

c 存储在本地对象存储中（步骤 3），然后将 c 的条目添加到 GCS（步骤 4）。结果，GCS 使用 c 的条目触发对 N1 的对象存储的回调（步骤 5）。接下来，N1 从 N2 复制 c（第 6 步），并将 c 返回给 ray.get()（第 7 步），最终完成任务。

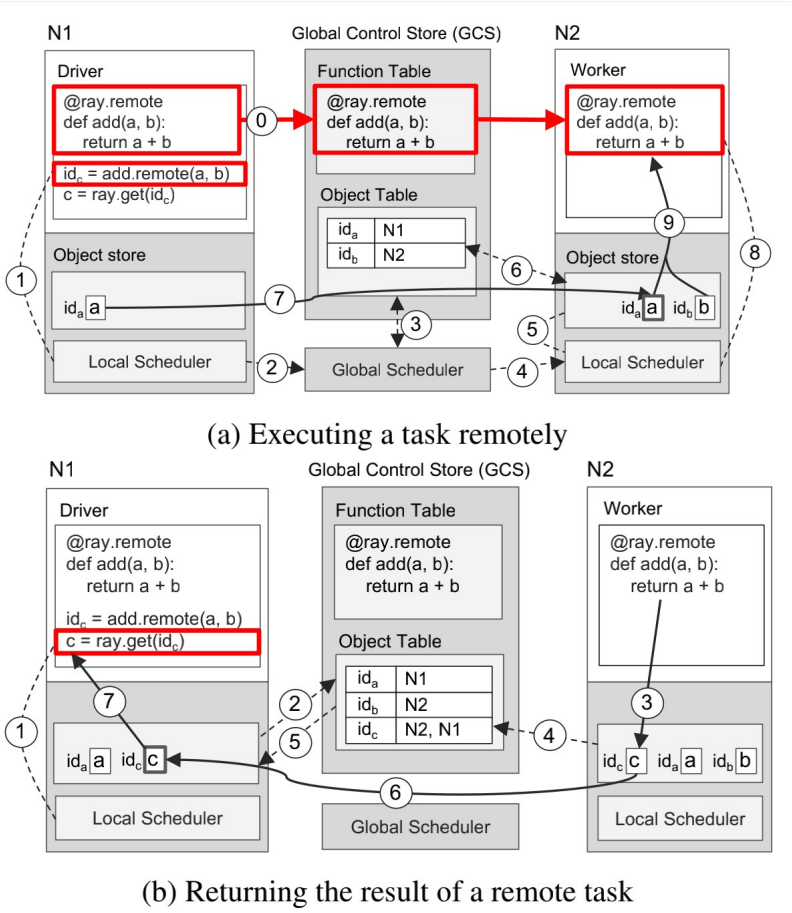


图 3: 添加 a 和 b 并返回 c 的端到端示例。

3 Ray 的性能评估

在对 Ray 进行性能评估的过程中，重点验证了以下几个问题：

- Ray 能否达到此前在延迟，拓展性和容错率方面的提出的要求？
- 使用 Ray 的 API 编写的分布式原语（例如 allreduce）有哪些开销？
- 在强化学习工作负载的背景下，Ray 与专用的 training、serving 和 simulation System 相比如何？
- 相比于针对强化学习任务定制的系统，Ray 具有哪些优势？

所有实验都在 Amazon 云服务上运行，我们在实验中均使用 m4.16xlarge CPU 实例和 p3.16xlarge GPU 实例。如使用其他配置，我们会在文中进行特殊的说明。

3.1 Microbenchmarks

- Locality-aware task placement.

细粒度 load balance 和 locality-aware placement 是 Ray 任务的主要优点。actors 一旦被放置，就无法将其计算移动到大型远程对象，而 tasks 可以。Ray 通过共享对象存储统一了 tasks 和 actors，允许开发人员使用 tasks，例如，对模拟 actors 生成的输出进行昂贵的后处理。

• End-to-end scalability.

全局控制存储（Global Control Store，GCS）和自底向上的分布式调度器的一个好处是能够水平扩展系统，以支持细粒度任务的高吞吐量，同时保持容错和低延迟任务调度。

• Object store performance.

为了评估 Object store performance 的性能，我们跟踪了两个指标：IOPS（针对 small objects）write throughput（针对 big objects）。在图4，随着对象大小的增加，单个客户端的写入吞吐量超过 15GB/s。对于 big objects，memcpy 控制对象创建时间。对于 small objects，主要的开销是在客户机和对象存储之间的串行化和 IPC。

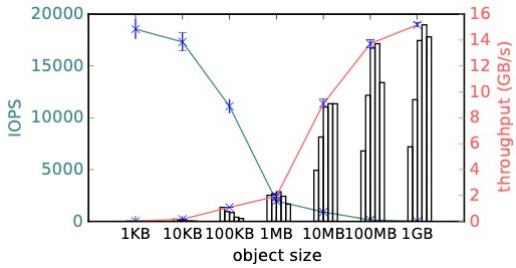


图 4: 对象存储 write throughput 和 IOPS。在 16 核实例（m4.4XL）上，从单个客户端来看，对于 big objects，吞吐量超过 15GB/s（红色），对于 small objects，吞吐量超过 18K IOPS（青色）。系统使用 8 个线程复制大于 0.5MB 的对象，1 个线程复制 small objects。条形图说明 1、2、4、8、16 个线程的吞吐量。结果是 5 次运行的平均值。

• GCS fault tolerance

为了在保持低延迟的同时提供强大的一致性和容错性，我们在 Redis 之上构建了一个轻量级的 chain replication 层。图5a 模拟了向 GCS 记录或读取 Ray tasks，其中 key 为 25 字节，value 为 512 字节。客户端以尽可能快的速度发送请求，一次最多有一个正在运行的请求。客户机（收到显式错误或重试超时）或来自 chain 中的任何服务器（已收到显式错误）。总的来说，重新配置导致 client-observed 到的最大延迟不到 30 毫秒（这包括故障检测和恢复延迟）。

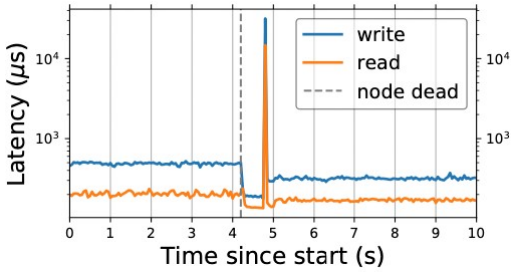


图 5: 从提交任务的客户端查看的 GCS 读写延迟时间线。该链从 2 个副本开始。我们手动触发重新配置，如下所示。当 t 4.2s 时，chain member 被杀死；紧接着，一个新的 chain member 加入，启动状态转移，并将链恢复为双向复制。尽管进行了重新配置，但最大 client-observed 延迟不超过 30 毫秒。

• GCS flushing

Ray 可以定期将 GCS 的内容刷新到 disk。在图5b 中，我们按顺序提交 5000 万个空任务，并监控 GCS 内存消耗。正如预期的那样，它随着跟踪任务的数量线性增长，最终达到系统的 memory 容量。在这一点上，系统将陷入停滞，工作负载无法在合理的时间内完成。通过定期 flush GCS，我们实现了两个目标。首先，将内存占用限制在用户可配置的级别（在 microbenchmark 中，我们采用了一种积极的策略，将消耗的内存保持在尽可能低的水平）。其次，刷新机制为长时间运行的 Ray 应用程序提供了一种自然的将 snapshot lineage 到磁盘的方法。

• Recovering from task failures

我们演示 Ray 使用 durable GCS lineage storage 从工作节点故障中透明恢复并弹性扩展的能力。工作负载，在 m4.xlarge 实例上运行, 由驱动程序提交的 100ms 任务的 linear chains 组成。当节点被移除时（在 25 秒、50 秒、100 秒），本地调度器重构链中先前的结果以继续执行。总体而言，每个节点的吞吐量始终保持稳定。

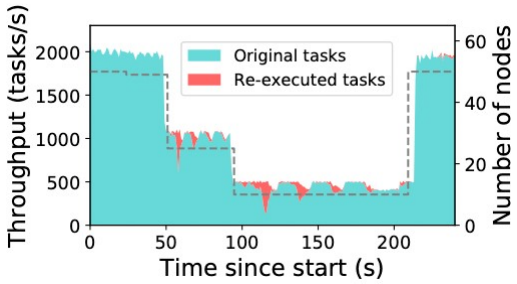


图 6: 在删除节点时，Ray 会重建丢失的任务依赖关系（虚线），并在重新添加节点时恢复到原始吞吐量。每个任务都是 100ms，取决于以前提交的任务生成的对象。

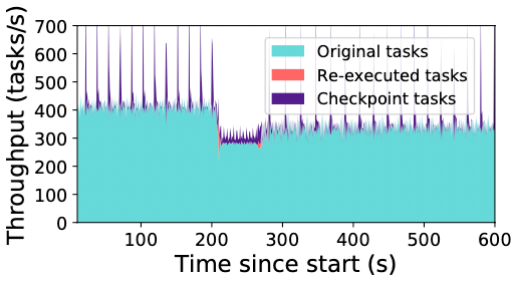


图 7: actors 从他们最后的检查点重建。在 t=200 秒时，我们杀死 10 个节点中的 2 个，导致集群中 2000 个参与者中的 400 个在其余节点上恢复（t=200-270 秒）。

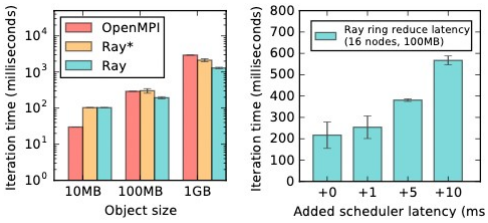
• Recovering from actor failures

通过在依赖关系图中将 actor 方法调用直接编码为有状态边，我们可以重用与图 11a 中相同的对象重构机制，为 stateful computation 提供透明的故障诊断。Ray 还使用用户定义的检查点函数来限制参与者的重新构造时间（图7）。在最小的开销下，检查点只允许重新执行 500 个方法，而不需要检查点就可以重新执行 10k 个方法。

• Allreduce

对于许多机器学习工作负载来说，Allreduce 通信机制原语非常重要。在这里，我们评估 Ray 是否能够以足够低的开销本地支持 ring allreduce 实现，以匹配现有实现。我们发现，Ray 在 100MB 的内存中跨 16 个节点完成 allreduce 200 毫秒和 1GB 英寸 1200ms，出人意料地分别比 OpenMPI（v1.10）（一

种流行的 MPI 实现）高出 1.5 倍和 2 倍（图8a）。我们将 Ray 的性能归因于它使用多线程进行网络传输，充分利用 AWS 上节点之间的 25Gbps 连接，而 OpenMPI 则在单个线程上发送和接收数据。对于较小的对象，OpenMPI 通过切换到较低开销的算法而优于 Ray，这是我们计划在未来实现的优化。



(a) Ray vs OpenMPI (b) Ray scheduler ablation

图 8: （a）所有还原的平均执行时间为 16M4.16xl 节点。每个辅助进程在不同的节点上运行。Ray* 将 Ray 限制为 1 个发送线程和 1 个接收线程。(b）Ray 的低延迟调度对于 allreduce 至关重要。

3.2 Building blocks

端到端应用程序需要 training、serving 和 simulation 的紧密耦合。在本节中，我们将这些工作负载中的每一个隔离到一个说明典型 RL 应用程序需求的设置中。由于针对 RL 的灵活编程模型，以及设计用于支持此编程模型的系统，Ray 与这些单独工作负载的专用系统的性能相匹配，有时甚至超过专用系统的性能。

• Distributed Training

我们利用 Ray actor 抽象来表示 model replicas，实现了数据并行同步 SGD。模型权重通过 allreduce（5.1）或 Parameter 服务器进行同步，这两种服务器都是在 Ray API 之上实现的。在图9中，我们使用相同的 TensorFlow 模型和合成数据发生器对 Ray（同步）参数服务器 SGD 实现的性能与最先进的实现进行了评估。我们仅与基于 TensorFlow 的系统进行比较，以准确测量 Ray 施加的超负荷，而不是深度学习框架本身之间的差异。在每次迭代中，模型副本参与者并行计算梯度，将梯度发送到分片 Parameter 服务器，然后从 Parameter 服务器读取求和的梯度，以用于下一次迭代。图9显示，Ray 与 Horovod 的性能相匹配，并且在分布式 TensorFlow 的 10% 以内（在分布式复制模式下）。这是由于能够表达与 Ray 的通用 API 中这些专用系统相同的应用程序级优化。一个关键的优化是在单次迭代中对梯度计算、传输和求和进行流水线处理。为了将 GPU 计算与网络传输重叠，我们使用自定义 TensorFlow 操作符将张量直接写入 Ray 的对象存储。

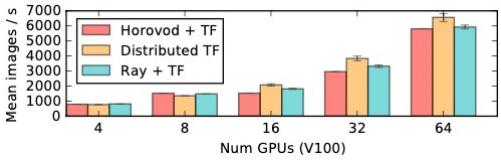


图 9: 分发 ResNet-101 TensorFlow 模型的训练时达到的每秒图像数（来自官方 TF 基准）。所有实验均在 p3.16xl 实例上进行，通过 25Gbps 以太网连接，workers 按照 Horovod[7] 中的做法为每个节点分配了 4 个 GPU。我们注意到一些与先前报告的测量偏差，可能是由于硬件差异和最近 TensorFlow 性能的改进。我们在所有运行中都使用 OpenMPI3.0、TF1.8 和 NCCL2。

• Serving

Model serving 是端到端应用程序的重要组成部分。Ray 主要关注在同一动态任务图中（例如，在 Ray 上的 RL 应用程序中）运行的模拟器的嵌入式模型服务。在此设置中，低延迟对于实现高利用率至关重要。为了说明这一点，在图10中，我们与使用开源 Clipper 系统比较而不是 REST 相比，使用 Ray actor 服务策略实现的服务器吞吐量。在这里，客户端和服务器进程都位于同一台机器上（p3.8X 大型实例）。这通常适用于 RL 应用程序，但不适用于 Clipper 等系统处理的一般 web 服务工作负载。由于其低开销的序列化和共享内存抽象，Ray 实现了一个数量级的高吞吐量，用于接收大输入的小型完全连接策略模型，并且在更昂贵的剩余网络策略模型上也更快，类似于 AlphaGo Zero 中使用的、接收较小输入的策略模型。

System	Small Input	Larger Input
Clipper	4400 ± 15 states/sec	290 ± 1.3 states/sec
Ray	6200 ± 21 states/sec	6900 ± 150 states/sec

图 10: 专用服务系统 Clipper[8] 和两个嵌入式服务工作负载的 Ray 的吞吐量比较。我们使用剩余网络和小型全连接网络，分别用 10ms 和 5ms 进行评估。服务器由客户端查询，每个客户端发送大小分别为 4KB 和 100KB 的状态，每批 64 个。

• Simulation

RL 中使用的模拟器产生可变长度（“时间步长”）的结果，由于训练的紧密循环，必须在可用时立即使用。任务异构性和及时性要求使得 BSP 风格的系统很难有效支持模拟。为了演示，我们将（1）一个 MPI 实现与（2）一个 Ray 程序进行比较，前者在 3 轮中提交在 n 个核上运行的 3n 并行模拟，两轮之间有一个全局屏障，后者在同时将模拟结果收集回驱动程序的同时发布相同的 3n 任务。表 4 显示，这两个系统都具有良好的可扩展性，但 Ray 的吞吐量高达 1.8 倍。这激发了一个编程模型，该模型可以动态生成和收集细粒度仿真任务的结果。

System, programming model	1 CPU	16 CPUs	256 CPUs
MPI, bulk synchronous	22.6K	208K	2.16M
Ray, asynchronous tasks	22.3K	290K	4.03M

图 11: OpenAI Gym 中 Pendulum-v0 模拟器每秒的时间步长 [9]。在大规模运行异构模拟时，Ray 允许更好的利用率。

3.3 RL Applications

由于没有一个能够将 training、serving 和 simulation 紧密结合的系统，强化学习算法现在只能作为一次性解决方案来实现，这使得很难将需要不同计算结构或利用不同体系结构的优化结合起来。因此，通过在 Ray 中实现两个具有代表性的强化学习应用程序，我们能够匹配甚至超越专门为这些算法构建的定制系统。主要原因是 Ray 的编程模型的灵活性，它可以表示应用程序级优化，这需要大量的工程效率才能移植到定制系统，但 Ray 的动态任务图执行引擎可以透明地支持。

• Evolution Strategies

为了评估大规模 RL 工作负载上的 Ray，我们实施了 Evolution Strategies（ES）算法，并与参考实现 [10] 进行了比较，参考实现是一个专门为该算法构建的系统，它依赖于 Redis for mes saging 和用于数据共享的低级多处理库。该算法定期向一组工作人员广播新策略，并汇总大约 10000 个任务的结果（每个任务执行 10 到 1000 个模拟步骤）。

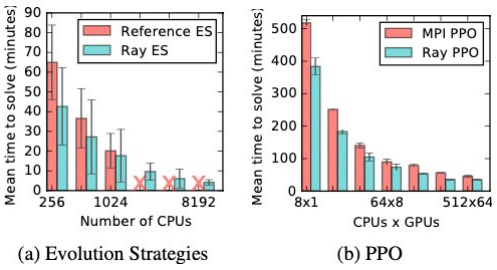
如图12a 所示，在 Ray 上的实现可扩展到 8192 个内核。将可用的堆芯增加一倍，平均完成时间加速比为 1.6×。相反，专用系统无法在 2048 个核上完成，系统中的工作超过了应用程序驱动程序的处理能力。为避免此问题，Ray 实现使用参与者的聚合树，平均时间为 3.7 分钟，是最佳发布结果（10 分钟）的两倍多。使用 Ray 的串行实现的初始并行化只需要修改 7 行代码。借助 Ray 对嵌套任务和参与者的支持，通过分层聚合提高性能很容易实现。相比之下，参考实现有数百行代码专用于工作人员之间通信任务和数据的协议，需要进一步的工程来支持分层聚合等优化。

• Proximal Policy Optimization

我们在 Ray 中实现了 Proximal Policy Optimization（PPO）[11]，并与使用 OpenMPI 通信原语的高度优化参考实现 [12] 进行了比较。该算法是一种异步分散-聚集算法，在模拟参与者执行任务时，会将新任务分配给他们将卷展栏返回给驱动程序。在收集 320000 个模拟步骤之前，提交任务（每个任务产生 10 到 1000 个步骤）。策略更新每格式 20 个步骤的 SGD，批量大小为 32768。本例中的模型参数约为 350KB。这些实验是使用 p2 进行的。16xlarge（GPU）和 m4.16xlarge（高 CPU）实例。

如图12所示，Ray 实现在所有实验中都优于优化的 MPI 实现，同时使用了一小部分 GPU。原因是 Ray 具有异构性意识，允许用户通过以任务或参与者的粒度表示资源需求来利用非对称体系结构。然后，Ray 实现可以利用 TensorFlow 的单进程多 GPU 支持，并尽可能将对象固定在 GPU 内存中。由于需要将卷展异步收集到单个 GPU 进程，因此无法轻松地将此优化移植到 MPI。事实上，[12] 包括两个 PPO 的定制实现，一个用于大型集群的 MPI，另一个用于 GPU 的优化，但仅限于单个节点。

Ray 允许一种适用于这两种场景的实现。Ray 处理资源异构性的能力也将 PPO 的成本降低了 4.5 倍 [13]，因为只有 CPU 的任务可以在更便宜的高 CPU 实例上调度。相反，MPI 应用程序通常表现出对称的体系结构，其中所有进程运行相同的代码并需要相同的资源，在这种情况下，防止使用仅 CPU 的机器进行扩展。此外，MPI 实现需要按需实例，因为它不能透明地处理故障。在 4× 便宜的 spot 实例中，Ray 的容错和资源感知调度一起将成本降低了 18×10。



(a) Evolution Strategies (b) PPO

图 12: Humanoid-v1 任务 [9] 中调用 ASCOREOF60000 的时间。（a）Ray ES 实现可扩展到 8192 个内核，平均时间为 3.7 分钟，是最佳发布结果的两倍多。专用系统无法运行超过 1024 个内核。在这个基准上，ES 比 PPO 快，但显示出更大的运行时差异。（b）Ray PPO 实现的性能优于专门的 MPI 实现 [12]，GPU 更少，成本更低。MPI 实现要求每 8 个 CPU 使用 1 个 GPU，而 Ray 版本最多需要 8 个 GPU（并且每 8 个 CPU 使用的 GPU 不得超过 1 个）。

4 Ray 的未来发展

如今，没有一个通用系统能够有效地支持 training、serving 和 simulation 的紧密循环。为了释放这些 core building blocks 并满足新兴 AI 应用程序的需求，Ray 在单个动态任务图中统一了 task-parallel 和 actor programming models，并采用了由全 GCS 和 a bottom-up distributed scheduler 支持的可扩展

架构。该体系结构同时实现的编程灵活性、高吞吐量和低延迟对于新兴人工智能工作负载尤其重要，因为人工智能工作负载产生的任务在资源需求、持续时间和功能方面都不同。我们的评估表明，线性可扩展性高达每秒 180 万个任务，透明的容错能力，以及在几个当代 RL 工作负载上的显著性能改进。因此，Ray 为未来 AI 应用程序的开发提供了灵活性、性能和易用性的强大组合。

参考文献

[1] I. Stoica, "The future of computing is distributed." `https://www.anyscale.com/blog/the-future-of-computing-is-distributed`, 2020.

[2] OpenAI, "Ai and compute." `https://openai.com/blog/ai-and-compute/`, 2019.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot, "Mastering the game of go with deep neural networks and tree search," *Nature*, 2016.

[4] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," 2018.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[6] A. Arrow. `https://arrow.apache.org/`.

[7] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," 2018.

[8] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," 2016.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[10] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[12] O. B. high-quality implementations of reinforcement learning algorithms. `https://github.com/openai/baselines`.

[13] E. I. Pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`.