Ray: 面向新兴人工智能应用的分布式框架

Philipp Moritz*, Robert Nishihara*, Stephanie Wang, Alexey Tumanov, Richard Liaw,
Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, Ion Stoica
加州大学伯克利分校

摘要

下一代人工智能应用将持续与环境进行交互并从这些交互中学习。这些应用对系统提出了新的严苛要求，包括性能和灵活性方面。在本文中，我们考虑了这些需求并提出了 Ray——一个用于解决这些问题的分布式系统。Ray 实现了一个统一的接口，可以表达任务并行和基于角色的计算，由单一动态执行引擎支持。为了满足性能要求，Ray 采用分布式调度器和分布式且容错的存储来管理系统的控制状态。在我们的实验中，我们展示了系统每秒可处理超过 180 万个任务的扩展能力，以及在几个具有挑战性的强化学习应用中比现有专用系统更好的性能。

1 引言

在过去的二十年中，许多组织一直在收集并尝试利用不断增长的大量数据。这导致了大量分布式数据分析框架的开发，包括批处理 [? ? ?]、流处理 [? ? ?] 和图处理 [? ? ?] 系统。这些框架的成功使组织能够将大型数据集的分析作为其商业或科学战略的核心部分，并开启了" 大数据" 时代。

最近，以数据为中心的应用范围已扩展到包含更复杂的人工智能 (AI) 或机器学习 (ML) 技术 [?]。典型案例是 *监督学习*，其中数据点伴随着标签，而将数据点映射到标签的主要技术由深度神经网络提供。这些深度网络的复杂性导致了另一波专注于深度神经网络训练及其在预测中使用的框架涌现。这些框架通常利用专门的硬件（如 GPU 和 TPU），目标是在批处理环境中减少训练时间。例子包括 TensorFlow [?]、MXNet [?] 和 PyTorch [?]。

然而，人工智能的前景远比传统监督学习广阔。新兴的 AI 应用必须越来越多地在动态环境中运行，对环境变化做出反应，并采取一系列行动来实现长期目标 [? ?]。它们不仅要利用收集的数据，还要探索可能的行动空间。这些更广泛的要求自然地纳入*强化学习* (RL) 范式中。RL 处理在不确定环境中基于延迟和有限反馈持续运行的学习 [?]。基于 RL 的系统已经产生了显著的结果，例如 Google 的 AlphaGo 击败人类世界冠军 [?]，并且开始应用于对话系统、无人机 [?] 和机器人操作 [? ?]。

RL 应用的核心目标是学习一个策略——从环境状态到行动选择的映射——以随时间产生有效的性能，例如赢得游戏或驾驶无人机。在大规模应用中找到有效策略需要三种主要能力。首先，RL 方法通常依赖*仿真*来评估策略。仿真使探索多种不同的行动序列选择成为可能，并了解这些选择的长期后果。其次，与监督学习对应物类似，RL 算法需要执行*分布式训练*，基于通过仿真或与物理环境交互生成的数据改进策略。第三，策略旨在为控制问题提供解决方案，因此必须在交互式闭环和开环控制场景中*提供*策略服务。

这些特性带来了新的系统需求：RL 系统必须支持*细粒度*计算（例如，在与现实世界交互时以毫秒级渲染动作，并执行大量仿真），必须支持时间上的*异质性*（例如，仿真可能需要毫秒或小时）和资源使用上的异质性（例如，训练使用 GPU 而仿真使用 CPU），并且必须支持 *动态*执行，因为仿真或与环境交互的结果可能改变未来的计算。因此，我们需要一个能够以毫秒级延迟处理每秒数百万异构任务的动态计算框架。

为大数据工作负载或监督学习工作负载开发的

*贡献相同

现有框架无法满足 RL 的这些新需求。批量同步并行系统，如 MapReduce [?]、Apache Spark [?] 和 Dryad [?] 不支持细粒度仿真或策略服务。任务并行系统，如 CIEL [?] 和 Dask [?] 对分布式训练和服务支持有限。流处理系统如 Naiad [?] 和 Storm [?] 也是如此。分布式深度学习框架如 TensorFlow [?] 和 MXNet [?] 不自然支持仿真和服务。最后，模型服务系统如 TensorFlow Serving [?] 和 Clipper [?] 既不支持训练也不支持仿真。

虽然原则上可以通过拼接几个现有系统来开发端到端解决方案（例如，Horovod [?] 用于分布式训练，Clipper [?] 用于服务，CIEL [?] 用于仿真），但实践中这种方法因应用程序内这些组件的*紧密耦合*而不可行。因此，研究人员和从业者目前为专门的 RL 应用构建一次性系统 [? ? ? ? ? ?]。这种方法对分布式应用的开发施加了巨大的系统工程负担，实质上将调度、容错和数据移动等标准系统挑战推给了每个应用。

在本文中，我们提出了 Ray，一个通用的集群计算框架，使 RL 应用的仿真、训练和服务成为可能。这些工作负载的需求范围从轻量级和无状态计算（如仿真）到长时间运行和有状态计算（如训练）。为满足这些需求，Ray 实现了一个统一接口，可以表达*任务并行*和*基于角色*的计算。*任务*使 Ray 能够高效地动态负载均衡仿真，处理大型输入和状态空间（例如图像、视频），并从故障中恢复。相比之下，*角色*使 Ray 能够高效支持有状态计算，如模型训练，并向客户端公开共享可变状态（例如参数服务器）。Ray 在单一高度可扩展且容错的动态执行引擎之上实现了角色和任务抽象。

为满足性能需求，Ray 分布了在现有框架中通常集中的两个组件 [? ? ?]：(1) 任务调度器和 (2) 维护计算谱系和数据对象目录的元数据存储。这使 Ray 能够以毫秒级延迟每秒调度数百万个任务。此外，Ray 为任务和角色提供基于谱系的容错，为元数据存储提供基于复制的容错。

虽然 Ray 在 RL 应用上下文中支持服务、训练和仿真，但这并不意味着它应该被视为替代在其他上下文中提供这些工作负载解决方案的系统。特别是，Ray 不旨在替代服务系统，如 Clipper [?] 和 TensorFlow Serving [?]，因为这些系统解决了部署模型的更广泛挑战，包括模型管理、测试和模型组合。

图 1: Ray 中实现图 ??示例的 Python 代码。注意 @ray.remote 表示远程函数和角色。远程函数和角色方法的调用返回期货（futures），可以传递给后续的远程函数或角色方法以编码任务依赖关系。每个角色都有一个环境对象 self.env，在其所有方法之间共享。

同样，尽管其灵活性，Ray 不能替代通用数据并行框架，如 Spark [?]，因为它目前缺乏这些框架提供的丰富功能和 API（例如，落后者缓解、查询优化）。

我们做出以下**贡献**：

- 我们设计并构建了第一个统一训练、仿真和服务的分布式框架——新兴 RL 应用的必要组件。
- 为支持这些工作负载，我们在动态任务执行引擎之上统一了角色和任务并行抽象。
- 为实现可扩展性和容错性，我们提出了一种系统设计原则，其中控制状态存储在分片元数据存储中，所有其他系统组件都是无状态的。
- 为实现可扩展性，我们提出了自下而上的分布式调度策略。

2 编程和计算模型

Ray 实现了一个动态任务图计算模型，即，它将应用程序建模为在执行过程中演变的依赖任务图。在此模型之上，Ray 同时提供了角色和任务并行编程抽象。这种统一区别于相关系统，如 CIEL（仅提供任务并行抽象）和 Orleans [?] 或 Akka [?]（主要提供角色抽象）。

2.1 编程模型

任务。*任务*表示在无状态工作节点上执行远程函数。当调用远程函数时，立即返回表示任务结果的 future。Futures 可以使用 **ray.get**() 获取，并在不等待其结果的情况下作为参数传递给其他远程函数。这允许用户表达并行性，同时捕获数据依赖关系。表 ??展示了 Ray 的 API。

远程函数操作不可变对象，并且应该是 *无状态*和无副作用的：它们的输出完全由输入决定。这意味着幂等性，通过函数在失败时重新执行简化了容错。

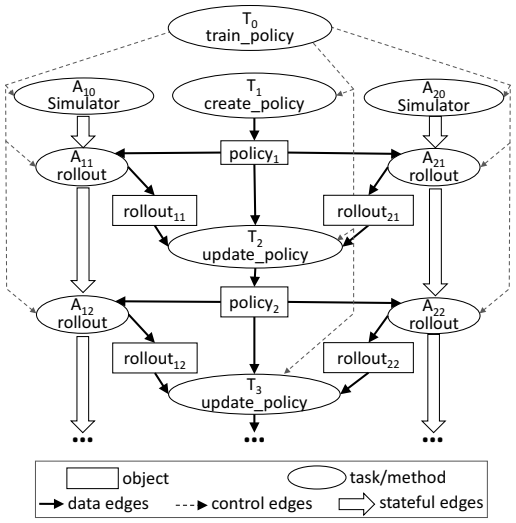


图 2: 对应于图 1中调用 train_policy.remote() 的任务图。远程函数调用和角色方法调用对应于任务图中的任务。图中显示了两个角色。每个角色的方法调用（标记为 A_{1i} 和 A_{2i} 的任务）之间有有状态的边，表示它们共享可变的角色状态。从 train_policy 到它调用的任务之间存在控制边。要并行训练多个策略，我们可以多次调用 train_policy.remote()。

角色。 角色表示有状态计算。每个角色公开可以远程调用并按顺序执行的方法。方法执行类似于任务，它远程执行并返回 future，但不同之处在于它在有状态工作节点上执行。角色的句柄可以传递给其他角色或任务，使它们能够调用该角色上的方法。

表 ??总结了任务和角色的属性。任务通过在任务粒度上利用负载感知调度实现细粒度负载均衡，输入数据局部性，因为每个任务可以在存储其输入的节点上调度，以及低恢复开销，因为不需要检查点和恢复中间状态。相比之下，角色提供更高效的细粒度更新，因为这些更新在内部而非外部状态上执行，外部状态通常需要序列化和反序列化。例如，角色可用于实现参数服务器 [?] 和基于 GPU 的迭代计算（如训练）。此外，角色可用于包装难以序列化的第三方模拟器和其他不透明句柄。

为满足异质性和灵活性需求（第 ?? 节），我们以三种方式增强 API。首先，为处理具有异质持续时间的并发任务，我们引入 **ray**.**wait**()，它等待前 k 个可用结果，而不像 **ray**.**get**() 那样等待所有结果。其次，为处理资源异质任务，我们使开发人员能够指定资源需求，使 Ray 调度器能够高效管理资源。第三，为提高灵活性，我们启用嵌套远程函数，这意味着远程函数可以调用其他远程函数。这对实现高可扩展性（第 ?? 节）也至关重要，因为它使多个进程能够以分布式方式调用远程函数。

2.2 计算模型

Ray 采用动态任务图计算模型 [?]，其中远程函数和角色方法的执行在其输入可用时由系统自动触发。在本节中，我们描述如何从用户程序（图 1）构建计算图（图 2）。该程序使用表 ??中的 API 实现图 ??中的伪代码。

首先忽略角色，计算图中有两类节点：数据对象和远程函数调用（或任务）。还有两类边：数据边和控制边。数据边捕获数据对象和任务之间的依赖关系。更准确地说，如果数据对象 D 是任务 T 的输出，我们从 T 到 D 添加一条数据边。同样，如果 D 是 T 的输入，我们从 D 到 T 添加一条数据边。控制边捕获嵌套远程函数（第 2.1 节）导致的计算依赖关系：如果任务 T_1 调用任务 T_2，则我们从 T_1 到 T_2 添加一条控制边。

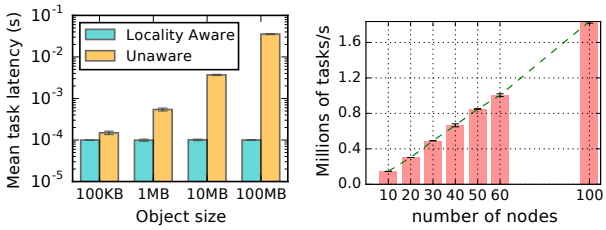
角色方法调用也表示为计算图中的节点。它们与任务相同，但有一个关键区别。为捕获同一角色上后续方法调用之间的状态依赖，我们添加第三类边：有状态边。如果方法 M_j 在同一角色上紧接方法 M_i 之后被调用，则我们从 M_i 到 M_j 添加一条有状态边。因此，在同一角色对象上调用的所有方法形成一个由有状态边连接的链（图 2）。这个链捕获了这些方法被调用的顺序。

有状态边帮助我们将角色嵌入到原本无状态的任务图中，因为它们捕获了共享角色内部状态的连续方法调用之间的隐式数据依赖。有状态边还使我们能够维护谱系。与其他数据流系统 [?] 一样，我们跟踪数据谱系以实现重建。通过明确地将有状态边包含在谱系图中，我们可以轻松重建丢失的数据，无论是由远程函数还是角色方法产生的（第 ?? 节）。

3 评估

在我们的评估中，我们研究以下问题：

1. Ray 如何满足第 ?? 节中列出的延迟、可扩展性和容错要求？（第 3.1 节）
2. 使用 Ray 的 API 编写的分布式原语（例如，allreduce）会带来哪些开销？（第 3.1 节）



(a) Ray 局部性调度　　(b) Ray 可扩展性

图 3: (a) 任务利用局部性感知放置。1000 个具有随机对象依赖性的任务被调度到两个节点之一。使用局部性感知策略，任务延迟保持独立于任务输入大小，而不是增长 1-2 个数量级。(b) 利用 GCS 和自下而上分布式调度器实现近乎线性的可扩展性。Ray 在 60 个节点上达到每秒 100 万个任务的吞吐量。由于成本原因，省略了 $x \in \{70, 80, 90\}$。

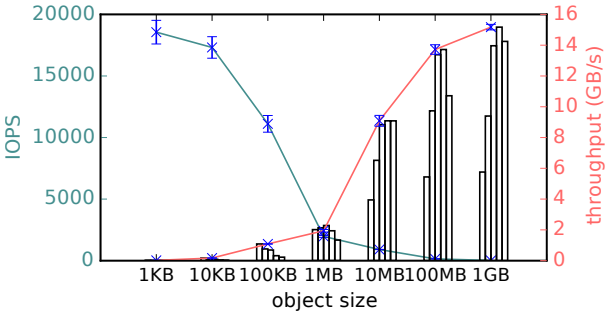


图 4: 对象存储写入吞吐量和 IOPS。从单个客户端，大型对象的吞吐量超过 15GB/s（红色），小型对象的 IOPS 达到 18K（青色），在 16 核实例上（m4.4xlarge）。它使用 8 个线程复制大于 0.5MB 的对象，1 个线程复制小对象。条形图报告了使用 1、2、4、8、16 个线程的吞吐量。结果是 5 次运行的平均值。

3. 在 RL 工作负载的背景下，Ray 与专门用于训练、服务和仿真的系统相比如何？（第 ?? 节）
4. 与定制系统相比，Ray 对 RL 应用有哪些优势？（第 ?? 节）

所有实验都在 Amazon Web Services 上运行。除非另有说明，我们使用 m4.16xlarge CPU 实例和 p3.16xlarge GPU 实例。

3.1 微基准测试

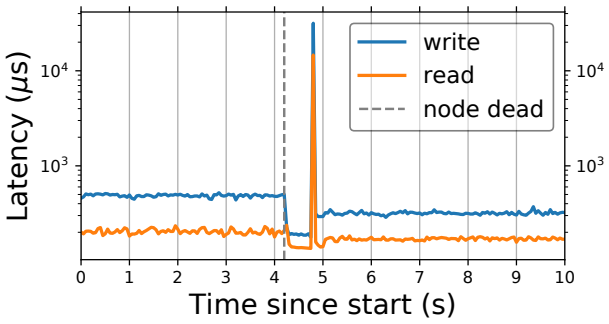
局部性感知任务放置。细粒度负载均衡和局部性感知放置是 Ray 中任务的主要优势。角色一旦放置，就无法将其计算移动到大型远程对象，而任务可以。在图 3a中，没有数据局部性感知放置的任务（就像角色方法的情况一样），在 10-100MB 输入数据大小时延迟增加 1-2 个数量级。Ray 通过共享对象存储统一任务和角色，允许开发人员使用任务进行例如，对模拟角色产生的输出进行昂贵的后处理。

端到端可扩展性。全局控制存储（GCS）和自下而上分布式调度器的主要优势之一是能够水平扩展系统以支持高吞吐量的细粒度任务，同时保持容错和低延迟任务调度。在图 3b中，我们在一个尴尬并行的空任务工作负载上评估这种能力，在 x 轴上增加集群大小。我们观察到任务吞吐量几乎完美的线性增长。Ray 在 60 个节点上超过每秒 100 万个任务的吞吐量，并继续线性扩展到 100 个节点上每秒超过 180 万个任务。最右边的数据点显示 Ray 可以在不到一分钟（54 秒）内处理 1 亿个任务，变异性最小。如预期的那样，增加任务持续时间会按照平均任务持续时间比例降低吞吐量，但整体可扩展性仍然保持线性。虽然许多现实工作负载可能由于对象依赖性和应用程序并行性的固有限制而表现出更有限的可扩展性，但这证明了我们的整体架构在高负载下的可扩展性。

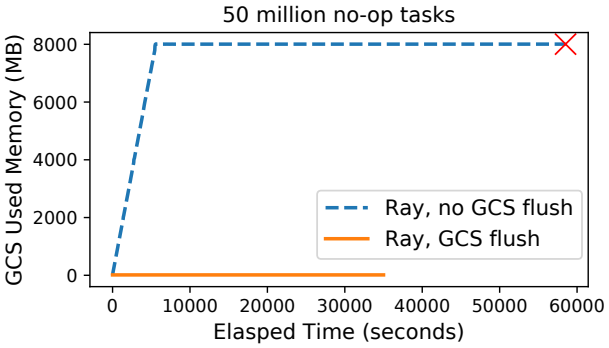
对象存储性能。为评估对象存储的性能（第 ?? 节），我们跟踪两个指标：IOPS（小对象）和写入吞吐量（大对象）。在图4中，随着对象大小增加，单个客户端的写入吞吐量超过 15GB/s。对于较大的对象，memcpy 主导对象创建时间。对于较小的对象，主要开销在于客户端和对象存储之间的序列化和 IPC。

GCS 容错。为了在提供强一致性和容错的同时保持低延迟，我们在 Redis 之上构建了一个轻量级链复制 [?] 层。图 5a模拟记录 Ray 任务到 GCS 并从 GCS 读取任务，其中键为 25 字节，值为 512 字节。客户端尽可能快地发送请求，同时最多有一个在途请求。故障会从客户端（收到明确错误或超时尽管重试）或链中的任何服务器（收到明确错误）报告给链主节点。总体而言，重新配置导致的最大客户端观察到的延迟不到 30ms（这包括故障检测和恢复延迟）。

GCS 刷新。 Ray 配备了定期将 GCS 内容刷新到磁盘的功能。在图5b中，我们顺序提交 5000 万个空任务并监控 GCS 内存消耗。如预期的那样，它随着跟踪的任务数量线性增长，最终达到系统的内存容量。此时，系统变得停滞，工作负载无法在合理的时间内完成。通过定期 GCS 刷新，我们实现两个目标。首先，内存占用被限制在用户可配置的水平（在

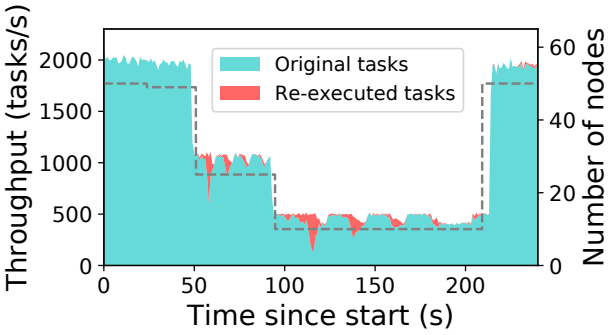


(a) GCS 读写延迟的时间线，从提交任务的客户端角度观察。链开始时有 2 个副本。我们手动触发重新配置，具体如下：在 $t \approx 4.2$ 秒时，一个链成员被杀死；紧接着，新的链成员加入，启动状态传输，并将链恢复到 2 路复制。尽管进行了重新配置，客户端观察到的最大延迟仍低于 30 毫秒。

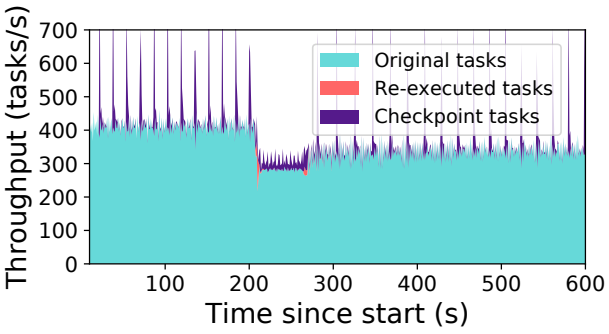


(b) Ray GCS 通过 GCS 刷新保持恒定的内存使用量。在没有 GCS 刷新的情况下，内存使用量达到最大容量并且工作负载在预定时间内无法完成 (用红色叉号表示)。

图 5: Ray GCS 容错和刷新。



(a) 任务重建



(b) 角色重建

图 6: Ray 容错机制。(a) 当节点被移除时，Ray 重建丢失的任务依赖（虚线），并在节点重新添加后恢复到原始吞吐量。每个任务耗时 100 毫秒，并依赖于先前提交的任务生成的对象。(b) 角色从其最后一个检查点重建。在 $t = 200$ 秒时，我们终止了 10 个节点中的 2 个，导致集群中的 2000 个角色中的 400 个在剩余节点上恢复（$t = 200$–270 秒）。

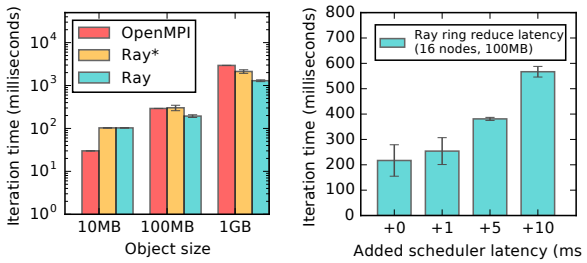
微基准测试中，我们采用激进策略，尽可能保持较低的内存消耗）。其次，刷新机制为长时间运行的 Ray 应用提供了一种自然的方式来将谱系快照到磁盘。

从任务故障恢复。在图 6a中，我们展示了 Ray 透明地从工作节点故障恢复并弹性扩展的能力，使用持久的 GCS 谱系存储。工作负载在 m4.xlarge 实例上运行，由驱动程序提交的 100ms 任务的线性链组成。随着节点被移除（在 25 秒、50 秒、100 秒），本地调度器重建链中先前的结果以继续执行。整体每节点吞吐量始终保持稳定。

从角色故障恢复。通过将角色方法调用编码为依赖图中的有状态边，我们可以重用与图 6a相同的对象重建机制，为有状态计算提供透明的容错。Ray 另外利用用户定义的检查点函数限制角色的重建时间（图6b）。在最小开销下，检查点使得只需重新执行 500 个方法，而不检查点则需要重新执行 10k 个方法。未来，我们希望进一步减少角色重建时间，例如，允许用户注释不改变状态的方法。

Allreduce。Allreduce 是许多机器学习工作负载重要的分布式通信原语。在这里，我们评估 Ray 是否能够原生支持足够低开销的环形 allreduce [?] 实现以匹配现有实现 [?]。我们发现 Ray 在 16 个节点上完成 100MB 的 allreduce 大约需要 ~200ms，1GB 需要 ~1200ms，出人意料地分别比 OpenMPI（v1.10）——一个流行的 MPI 实现——快 1.5× 和 2×（图 7a）。我们将 Ray 的性能归因于其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 则在单个线程上顺序发送和接收数据 [?]。对于较小的对象，OpenMPI 通过切换到较低开销的算法优于 Ray，这是我们计划在未来实现的优化。

Ray 的调度器性能对实现诸如 allreduce 之类的



(a) Ray 与 OpenMPI 比较　　(b) Ray 调度器消融研究

图 7: (a) 16 个 m4.16xl 节点上 allreduce 的平均执行时间。每个工作节点运行在不同节点上。Ray* 将 Ray 限制为 1 个线程发送和 1 个线程接收。(b) Ray 的低延迟调度对 allreduce 至关重要。

原语至关重要。在图 7b中，我们注入人工任务执行延迟，并显示即使只有几毫秒的额外延迟，性能也会下降近 2×。像 Spark 和 CIEL 这样具有中央调度器的系统通常具有数十毫秒的调度器开销 [? ?]，使这类工作负载不切实际。调度器吞吐量也成为瓶颈，因为环形 reduce 所需的任务数量随集群规模呈线性增长。

相关工作

动态任务图。 Ray 与 CIEL [?] 和 Dask [?] 密切相关。三者都支持具有嵌套任务的动态任务图并实现了 futures 抽象。CIEL 还提供基于谱系的容错，而 Dask 与 Ray 一样，完全集成了 Python。然而，Ray 在两个方面有所不同，这些差异对性能有重要影响。首先，Ray 用角色抽象扩展了任务模型。这对于分布式训练和服务中的高效有状态计算是必要的，以保持模型数据与计算的位置一致。其次，Ray 采用完全分布式和解耦的控制平面和调度器，而不是依赖存储所有元数据的单一主节点。这对于在不修改系统的情况下高效支持像 allreduce 这样的原语至关重要。在 16 个节点上处理 100MB 的峰值性能时，Ray 上的 allreduce（第3.1节）在 200ms 内提交 32 轮共 16 个任务。与此同时，Dask 报告在 512 个核心上的最大调度器吞吐量为 3k 任务/秒 [?]。使用集中式调度器，allreduce 的每一轮将至少产生 ∼5ms 的调度延迟，导致完成时间最多增加 2×（图7b）。即使使用分散式调度器，将控制平面信息与调度器耦合也会使后者处于数据传输的关键路径上，为 allreduce 的每一轮增加额外的往返时间。

数据流系统。 流行的数据流系统，如 MapReduce [?]、Spark [?] 和 Dryad [?] 在分析和 ML 工作负载中被广泛采用，但它们的计算模型对于细粒度和动态的仿真工作负载来说过于限制。Spark 和 MapReduce 实现了 BSP 执行模型，该模型假设同一阶段内的任务执行相同的计算并且花费大致相同的时间。Dryad 放宽了这一限制，但缺乏对动态任务图的支持。此外，这些系统都不提供角色抽象，也没有实现分布式可扩展的控制平面和调度器。最后，Naiad [?] 是一个数据流系统，为某些工作负载提供了改进的可扩展性，但仅支持静态任务图。

机器学习框架。 TensorFlow [?] 和 MXNet [?] 针对深度学习工作负载，高效利用 CPU 和 GPU。虽然它们对由线性代数操作组成的静态 DAG 的训练工作负载实现了出色的性能，但对于将训练与仿真和嵌入式服务紧密耦合所需的更一般计算支持有限。TensorFlow Fold [?] 提供了对动态任务图的一些支持，MXNet 也通过其内部 C++ API 提供支持，但都不完全支持在执行过程中根据任务进度、任务完成时间或故障修改 DAG 的能力。TensorFlow 和 MXNet 原则上通过允许程序员模拟低级消息传递和同步原语来实现通用性，但在这种情况下的陷阱和用户体验与 MPI 类似。OpenMPI [?] 可以实现高性能，但编程相对困难，因为它需要显式协调来处理异构和动态任务图。此外，它迫使程序员显式处理容错。

角色系统。 Orleans [?] 和 Akka [?] 是两个非常适合开发高可用性和并发分布式系统的角色框架。然而，与 Ray 相比，它们对从数据丢失中恢复的支持较少。要恢复有状态角色，Orleans 开发人员必须显式检查点角色状态和中间响应。Orleans 中的无状态角色可以复制以实现横向扩展，因此可以充当任务，但与 Ray 不同，它们没有谱系。同样，虽然 Akka 明确支持在故障期间持久化角色状态，但它不为 无状态计算（即任务）提供高效的容错。对于消息传递，Orleans 提供至少一次，Akka 提供最多一次语义。相比之下，Ray 提供透明的容错和精确一次语义，因为每个方法调用都记录在 GCS 中，且参数和结果都是不可变的。我们发现这些限制在实践中不会影响我们应用程序的性能。Erlang [?] 和 C++ Actor Framework [?]，两个其他基于角色的系统，对容错的支持同样有限。

全局控制存储和调度。 逻辑上集中控制平面的概念先前已在软件定义网络（SDN）[?]、分布式文件系统（例如 GFS [?]）、资源管理（例如 Omega [?]）和分

布式框架（例如 MapReduce [?]、BOOM [?]）等方面提出。Ray 从这些开创性工作中汲取灵感，但提供了显著改进。与 SDN、BOOM 和 GFS 相比，Ray 解耦了控制平面信息的存储（例如 GCS）和逻辑实现（例如调度器）。这允许存储和计算层独立扩展，这是实现我们可扩展性目标的关键。Omega 使用分布式架构，其中调度器通过全局共享状态进行协调。对于这种架构，Ray 添加了全局调度器以平衡本地调度器之间的负载，并以毫秒级（而非秒级）为目标进行任务调度。

Ray 实现了一个独特的分布式自下而上调度器，它可水平扩展，并能处理动态构建的任务图。与 Ray 不同，大多数现有集群计算系统 [? ? ?] 使用集中式调度器架构。虽然 Sparrow [?] 是分散的，但其调度器做出独立决策，限制了可能的调度策略，并且一个作业的所有任务都由同一全局调度器处理。Mesos [?] 实现了两级层次调度器，但其顶级调度器管理框架，而非单个任务。Canary [?] 通过让每个调度器实例处理任务图的一部分实现了令人印象深刻的性能，但不处理动态计算图。

Cilk [?] 是一种并行编程语言，其工作窃取调度器为动态任务图实现了可证明的高效负载均衡。然而，没有像 Ray 的全局调度器这样的中央协调器，这种完全并行的设计也难以扩展以支持分布式环境中的数据局部性和资源异构性。

致谢

本研究部分得到 NSF CISE Expeditions Award CCF-1730628 的支持，以及来自阿里巴巴、亚马逊网络服务、蚂蚁金服、Arm、CapitalOne、爱立信、Facebook、谷歌、华为、英特尔、微软、丰业银行、Splunk 和 VMware 的资助，以及 NSF grant DGE-1106400 的支持。我们感谢匿名评审人和我们的指导教授 Miguel Castro 提供的有思想性的反馈，这些反馈帮助提高了本文的质量。