简介

下一代 AI 应用程序将不断与环境交互，并从这些交互中学习。这些应用程序在性能和灵活性方面提出了新的和苛刻的系统要求。在本文中，我们考虑这些要求并提出 Ray，一个分布式系统来解决它们。Ray 实现了一个动态任务图计算模型，支持任务并行和基于 actor 的编程模型。为了满足人工智能应用的性能要求，我们提出了一种架构，使用共享式存储系统和新颖的自下而上的分布式调度器，在逻辑上集中了系统的控制状态。在我们的实验中，我们展示了亚毫秒的远程任务延迟和线性吞吐量，每秒超过 180 万个任务。我们凭经验验证了 Ray 加速了具有挑战性的基准测试，同时也是适合新兴的一类强化学习应用和算法的高性能表现。

1 介绍

人工智能目前正在成为一系列真实世界应用的主力技术。但是，迄今为止，这些应用程序主要基于一个相当局限的监督式学习模式，在这种模式下，一个模型在线性训练，并被部署来在线服务预测。随着领域的成熟，有必要考虑比标准监督式学习更广泛的设置。机器学习应用程序不是做出和服务于单一预测，而是必须越来越多地在动态环境中运行，对环境变化作出反应，并采取一系列行动以实现一个目标。这些更广泛的要求自然地在强化学习（RL）的范式内进行，该学习在不确定的环境中学习持续运行。基于 RL 的应用程序已经取得显著的结果，例如 Google 的 AlphaGo 击败人类世界冠军，并正在寻找到自动驾驶汽车，无人机和机器人操纵方式。

有三个特点将 RL 应用与传统的监督学习应用区分开来。首先，他们经常严重依赖模拟来探索状态和发现行动的后果。模拟器可以编码计算机游戏的规则，诸如机器人的物理系统的牛顿动态，或者虚拟环境的混合动力学。这通常需要大量的计算;例如，一个现实的应用程序可能会执行数以百万计的模拟。其次，RL 应用程序的图计算是异构的并且是动态演化的。仿真花费的时间从几毫秒到几分钟，仿真的结果可以确定未来仿真的参数。第三，许多 RL 应用程序，例如机器人控制或自动驾驶，需要迅速采取行动以应对不断变化的环境。此外，要选择最佳的操作，应用程序可能需要实时执行更多的模拟。总之，我们需要一个支持异构和动态图计算的计算框架，同时以毫秒级别的延迟处理每秒数百万个任务。

现有的集群计算框架没有充分满足这些要求。MapReduce，Apache Spark，Dryad，Dask 和 CIEL 不支持通用 RL 应用程序所需的吞吐量和等待时间，而 TensorFlow，Naiad，MPI 和 Canary 通常假设静态计算图。

在本文中，我们提出了一个满足这些要求的集群计算框架 Ray。为了支持这些应用程序强加的异构和动态工作负载，Ray 实现了一个动态的任务图计算模型，类似于 CIEL。但是，除了 CIEL 提供的任务并行抽象外，Ray 还在此执行模型之上提供了一个 actor 编程抽象。Actor 抽象使得 Ray 能够支持有状态的组件，比如第三方模拟器。

为了在支持动态计算图的同时实现严格的性能目标，Ray 采用了可横向扩展的新型分布式体系结构。架构基于两个关键的想法。首先，我们将系统的所有控制状态存储在一个全局控制存储器中，这使得系统中的所有其他组件都是无状态的。因此，每个组件都可以轻松地水平扩展，并在发生故障时重新启动。反过来，全局控制存储可以通过共享进行扩展，并通过复

制实现容错。

其次，我们介绍一种新的自下而上的分布式调度器，其任务由工作节点和驱动程序提交给本地调度器（每个节点有一个本地调度器）。本地调度程序可以选择本地调度任务或将任务转发到已备份的全局调度程序。这通过允许本地决策来减少任务延迟，并且通过减轻全局调度器的负担来增加系统吞吐量。我们做出如下贡献：

1. 我们为新兴 AI 应用指定系统需求：支持（a）异构并行计算，（b）动态任务图，（c）高吞吐量和低延迟调度，以及（d）透明容错。
2. 除了任务并行编程抽象之外，我们还提供动态任务图计算模型之上的 actor 抽象。
3. 我们提出了一个可水平扩展的体系结构来满足上述要求，并构建实现此体系结构的集群计算系统 Ray。

2 动机和需求

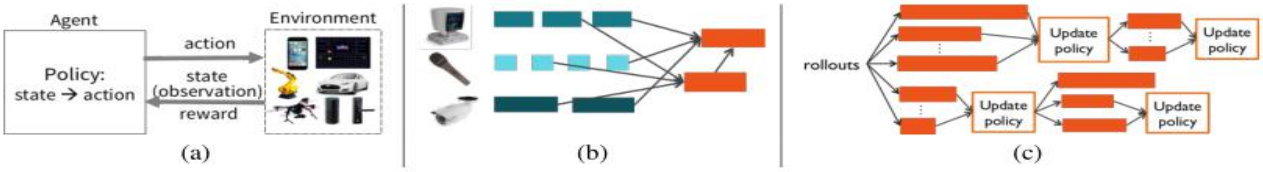


Figure 1: (a) An RL system. (b) Task graph for processing sensor inputs. (c) Task graph for learning policy.

Application	Environment	Agent	State	Action	Reward
Go (game)	Board and opponent	Player	Board position	Place a piece	Game outcome
Atari Pong	Program	Player	Sequence of video frames	Control joystick	Pong score
Robot moving an object	Physical world	Control program	Object and robot pose	Actuate joints	Object moved

Table 1: Example RL applications.

```
// generate a trajectory under a given policy
rollout(policy, environment):
  trajectory ← []
  state ← environment.initial_state()
  while (not environment.has_terminated()):
    action ← policy.compute(state)
    state, reward ← environment.step(action)
    trajectory.append(state, reward)
  return trajectory

// learn a policy in a given environment
train_policy(environment):
  policy ← initial_policy()
  while (policy has not converged):
    trajectories ← []
    // generate k rollouts and use them to update policy
    for i from 1 to k:
      trajectories.append(rollout(policy, environment))
    policy = policy.update(trajectories)
  return policy
```

Figure 2: Pseudocode for a typical RL training application.

虽然 Ray 可以支持各种工作负载，因为它提供了任务并行和参与者抽象，但我们专注于本文中的加强学习（RL）工作负载，因为它们是新兴 AI 应用的代表，也是 Ray 设计的主要驱动因素。在这里，我们考虑一个简单的 RL 应用程序来说明 Ray 的关键要求。

RL 系统由与环境重复交互的代理组成（参见 Figure 1(a)）。Agent 的目标是学习最大化回报的策略。策略是从环境状态到采取行动的映射。环境，状态，Agent，行动和回报的定义是特定于应用程序的 (Table 1)。

Figure 2 展示了 agent 学习策略所使用的伪代码示例。典型的程序包括两个步骤：（1）评估当前的政策;（2）改进政策。为了评估策略，伪代码调用 rollout（环境，策略）来生成一组 rollout，其中 rollout 是通过使用 environment.step（action）与环境交互收集的状态和回报的轨迹。根据当前策略和环境状态通过 policy.compute（state）计算操作。随着轨迹的产生，train_policy（）使用已完成的轨迹通过 policy.update（轨迹）改进当前的策略。重复这个过程直到策略收敛。

虽然很简单，但该应用程序说明了新兴 AI 应用程序的关键要求。我们将这些要求分为三类。

灵活性。 系统的灵活性通常根据其可支持的工作负载的多样性来衡量。我们考虑灵活性的两个方面：并发执行任务的异构性和执行图计算的一般性和动态性。

并行，异构的任务。并行任务可能在三个方面是异构的：

1. 功能。以机器的情况，评估环境的状态（例如，environment.step（action））涉及处理多个传感器（例如视频，麦克风和雷达）的输入。这需要并行运行多个任务，每个都执行不同的计算（见 Figure 1（b））。
2. 持续时间。计算轨迹花费的时间可能会有很大差异（参照 rollout(policy, environment)）。例如，在游戏的情况下，可能只需要几个动作（动作）就会输掉游戏，或者可能需要数百动作来获胜。
3. 资源类型。通过评估策略（例如，policy.compute（状态））来计算动作在很多情况下通过深度神经网络来实现，所述深度神经网络通常需要使用 GPU。另一方面，大多数其他应用的计算也需使用 CPU。

请注意，这些需求并非对当今许多流行的群集计算框架实现的批量同步并行（BSP）模型满足。使用 BSP，同一阶段内的所有任务通常执行相同的计算（尽管在不同的数据分区上）并且花费大致相同的时间量。

动态任务图。考虑 train_policy（）函数。尽管未在 Figure 2 中显示，但只要部分 rollouts 完成（而不是等待所有）并启动新的 rollouts 以维护执行 rollouts（如图 Figure 1(c)所示）。这使得执行图形具有动态性，因为我们无法预测 rollouts 的完成顺序或哪些 rollouts 将用于特定策略更新。

性能。 在机器人与物理环境进行交互的情况下，我们需要推断环境的状态并在几毫秒内计算新的行为。同样，模拟也可能需要几毫秒的量级。因此，我们需要能够在不到一毫秒的时间内安排任务。否则，调度开销可能会很大。鉴于拥有数万个内核的集群很常见，我们需要能够每秒安排数十万甚至数百万个任务。考虑一个由 100 台服务器组成的集群，每台服务器都有 32 个内核，假设每个任务需要 5ms 执行。为了充分利用这个集群，我们需要安排 640K 任务/秒。

易于开发。 由于编写并行应用程序并不重要，因为 ML 开发人员更倾向于专注于其应用程序而不是系统编程，所以简化开发对于此类系统的成功至关重要。
确定性重新运行和容错。确定性地重新运行作业的能力大大简化了调试。透明的容错功能可以避免用户明确处理故障。它还使用户能够使用便宜的可抢占资源（例如 AWS 上的现货实例），从而在公共云中运行时节省大量成本。
现有算法的简单并行化。这涉及提供一个简单的 API 并支持现有的语言，工具和库。首先，我们需要为 Python 提供支持，因为 Python 是 AI 开发人员的首选语言。其次，我们需要提供与广泛的可用第三方库的紧密集成。这些库包括模拟器，如 OpenAI，DeepMind 实验室，Mujoco 物理模拟器以及 TensorFlow，Theano，PyTorch 和 Caffe 等深度学习框架。 正如我们将看到的，这需要用类似 actor 的抽象来扩展任务并行模型以包装这些第三方服务。

3 编程和计算模型

Name	Description
$futures\_list = \textbf{f.remote}(args)$	Execute function $f()$ remotely. $f()$ takes either object values or futures as arguments, and returns a list of futures. This is a non-blocking call.
$obj\_list = \textbf{ray.get}(futures\_list)$	Return the values associated with a list of futures. This is a blocking call.
$futures\_list\_done = \textbf{ray.wait}(futures\_list, k, timeout)$	Given a list of futures, return the futures whose corresponding tasks have completed as soon as either k of the tasks have completed or the timeout expires.
$actor = \textbf{Class.remote}(args)$ $futures\_list = actor.\textbf{method.remote}(args)$	Instantiate class *Class* as a remote actor, and return a reference to it. Call a method on the remote actor and return a list of futures. This is a non-blocking call.

Table 2: Ray API

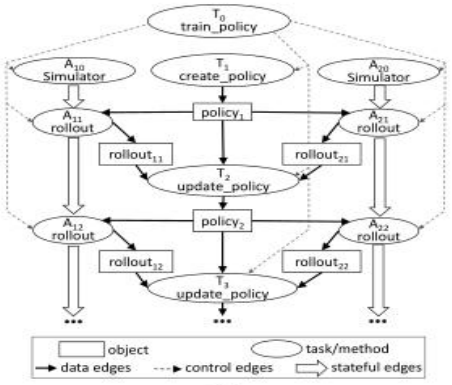
```
@ray.remote
def create_policy():
    # Initialize the policy randomly.
    return policy

@ray.remote(num_gpus=1)
class Simulator(object):
    def __init__(self):
        # Initialize the environment.
        self.env = Environment()
    def rollout(self, policy, num_steps):
        observations = []
        observation = self.env.current_state()
        for _ in range(num_steps):
            action = compute(policy, observation)
            observation = self.env.step(action)
            observations.append(observation)
        return observations
```

```
@ray.remote(num_gpus=2)
def update_policy(policy, *rollouts):
    # Update the policy.
    return policy

@ray.remote
def train_policy():
    # Create a policy.
    policy_id = create_policy.remote()
    # Create 10 actors.
    simulators = [Simulator.remote() for _ in range(10)]
    # Do 100 steps of training.
    for _ in range(100):
        # Perform one rollout on each actor.
        rollout_ids = [s.rollout.remote(policy) for s in simulators]
        # Update the policy with the rollouts.
        policy_id = update_policy.remote(policy_id,
                                         *rollout_ids)
    return ray.get(policy_id)
```

(a)



(b)

Figure 3:（a）在 Ray 中执行图 2 中的示例的 Python 代码。请注意，@ray.remote 表示远程函数和 actor。远程函数和 actor 方法的调用返回 future，这可以传递给随后的远程函数或 actor 方法来编码任务依赖关系。每个参与者都有一个环境对象 self.env 在其所有方法之间共享。（b）与 train_policy.remote()的调用相对应的任务图。远程函数调用和 actor 方法调用对应于任务图中的任务。该图显示了两种 actor。每个 actor（标记为 A1i 和 A2i 的任务）的方法调用在它们之间具有有状态的边，表示它们共享可变的 actor 状态。从 train_policy 到它调用的任务都有控制边。为了同时训练多个策略，我们可以多次调用 train_policy.remote()。

3.1 编程模型和 API

Ray 的核心是提供一个任务并行编程模型。Table 2 显示了 Ray 的 API。当调用远程函数时，将立即返回表示任务结果的 future。可以使用 ray:get()来检索 future，将来可以作为参数传

递到另一个远程函数。这允许用户在捕获数据依赖性时表达并行性。
远程函数对不可变对象进行操作，并且预期无状态和副作用：它们的输出完全由它们的输入决定。这意味着幂等性，它通过在失败时重新执行函数来简化容错。
为了满足第 2 节给出的异构性，灵活性和开发简便性的要求，我们用四种方法来增强任务并行编程模型。

首先，为了处理具有不同持续时间的并发任务，我们引入了 ray:wait()。这个调用需要一系列 futures，并在超时后或至少有 k 个可用时返回其结果可用的子集。相反，ray:get()会阻塞除非所有的 future 可用。 这对于 RL 应用来说是非常有益的，因为仿真可能具有广泛不同的持续时间，但由于引入的不确定性而使得容错复杂化。

其次，为了处理资源异构任务，我们使开发人员能够指定资源需求，以便 Ray 调度程序可以高效地管理资源。为远程函数指定的资源仅在函数执行期间分配。

第三，为了提高灵活性，我们启用了嵌套远程功能，这意味着远程功能可以调用其他远程功能。这对于实现高可伸缩性（见第 4 节）也很关键，因为它使多个进程能够并行调用远程功能（否则驱动程序将成为任务调用的瓶颈）。

最后，也是最重要的一点，为了便于开发和提高效率，我们通过使用 Actor 抽象来增强我们的编程模型。我们在开发无状态任务时遇到的一个限制是无法包含第三方模拟器，这些模拟器不公开其内部状态。为了解决这个限制，Ray 以 actor 的形式为有状态组件提供基本支持。在 Ray 中，actor 是一个有状态的进程，它公开了一组可以作为远程函数调用并且可以连续执行这些方法的方法。

3.2 计算模型

Ray 采用动态任务图计算模型，在该模型中，系统在输入可用时自动触发远程函数和 actor 模型方法的执行。在本节中，我们将描述如何从用户程序（Figure 3(a)）构建图计算（Figure (b)）。该程序使用 Table 2 中的 API 来实现 Figure 2 中的伪代码。

首先忽略角色，图计算中有两种类型的节点：数据对象和远程函数调用或任务。还有两种类型的边缘：数据边缘和控制边缘。数据边缘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，则我们添加从 T 到 D 的数据边缘。同样，如果 D 是 T 的输入，我们添加从 D 到 T 的数据边缘。控制边捕获嵌套远程函数产生的计算依赖性（第 3.1 节）：如果任务 T1 调用任务 T2，则我们添加从 T1 到 T2 的控制边。

Actor 方法调用也被表示为图计算中的节点。它们与具有一个关键区别的任务完全相同。为了在同一个 actor 上的后续方法调用中捕获状态依赖，我们添加第三种类型的边缘：一个有状态的边缘。如果方法 Mj 在同一个 actor 上的方法 Mi 之后被调用，那么我们添加一个从 Mi 到 Mj 的有状态边。因此，在同一个 actor 对象上调用的所有方法形成一个由状态边连接的链（Figure 3(b)）。该链捕获这些方法被调用的顺序。

有状态的边帮助我们将 actor 嵌入到另一个无状态的任务图中，因为它们捕获连续方法调用之间共享 actor 内部状态的隐式数据依赖关系。有状态的边缘也使我们能够保持血统。和其

他数据流系统一样，我们追踪数据血缘以实现重建。通过在血缘图中明确包含有状态边，我们可以轻松地重建丢失的数据，无论是由远程函数还是由 actor 方法产生（第 4.2.3 节）。

4 系统结构

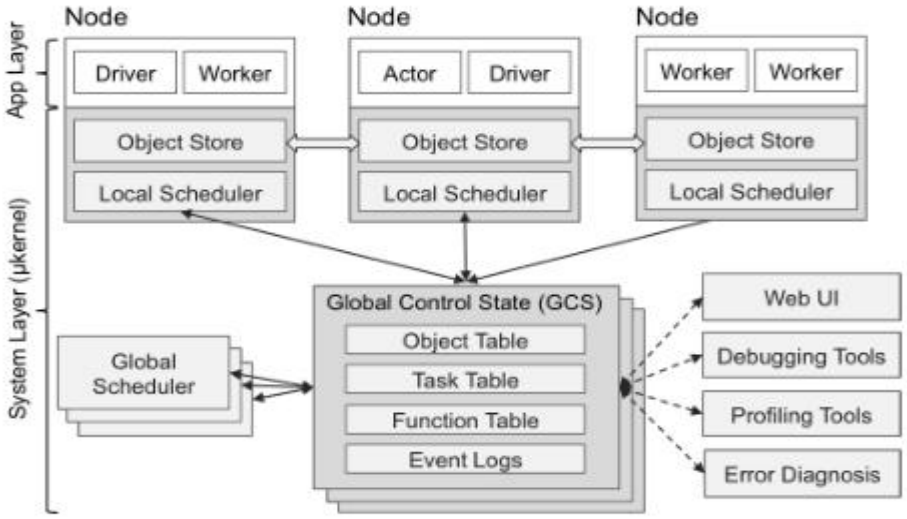


Figure 4:Ray 的体系结构由两部分组成：应用程序层和系统层。 应用层实现第 3 节中描述的 API 和计算模型，系统层实现任务调度和数据管理以满足性能和容错要求。

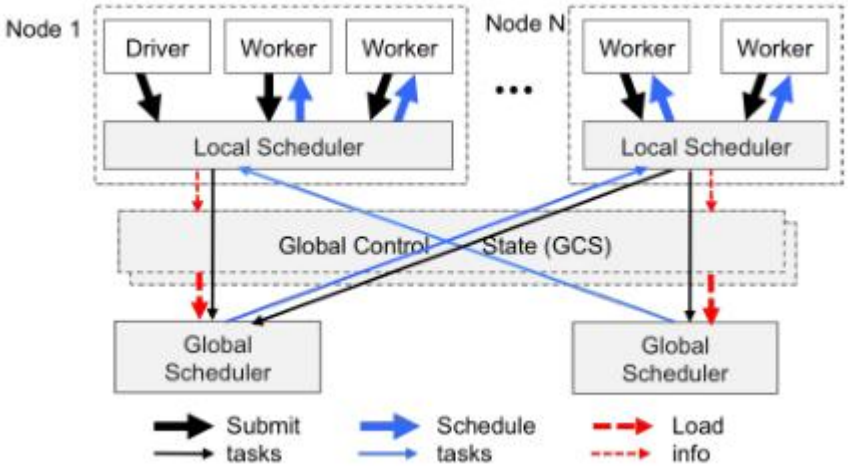


Figure 5: 自下而上的分布式调度器。任务从 driver 和 worker 自下而上地提交给本地调度程序，并仅在需要时才转发给全局调度程序（第 4.2.2 节）。每个箭头的厚度与其请求率成正比。

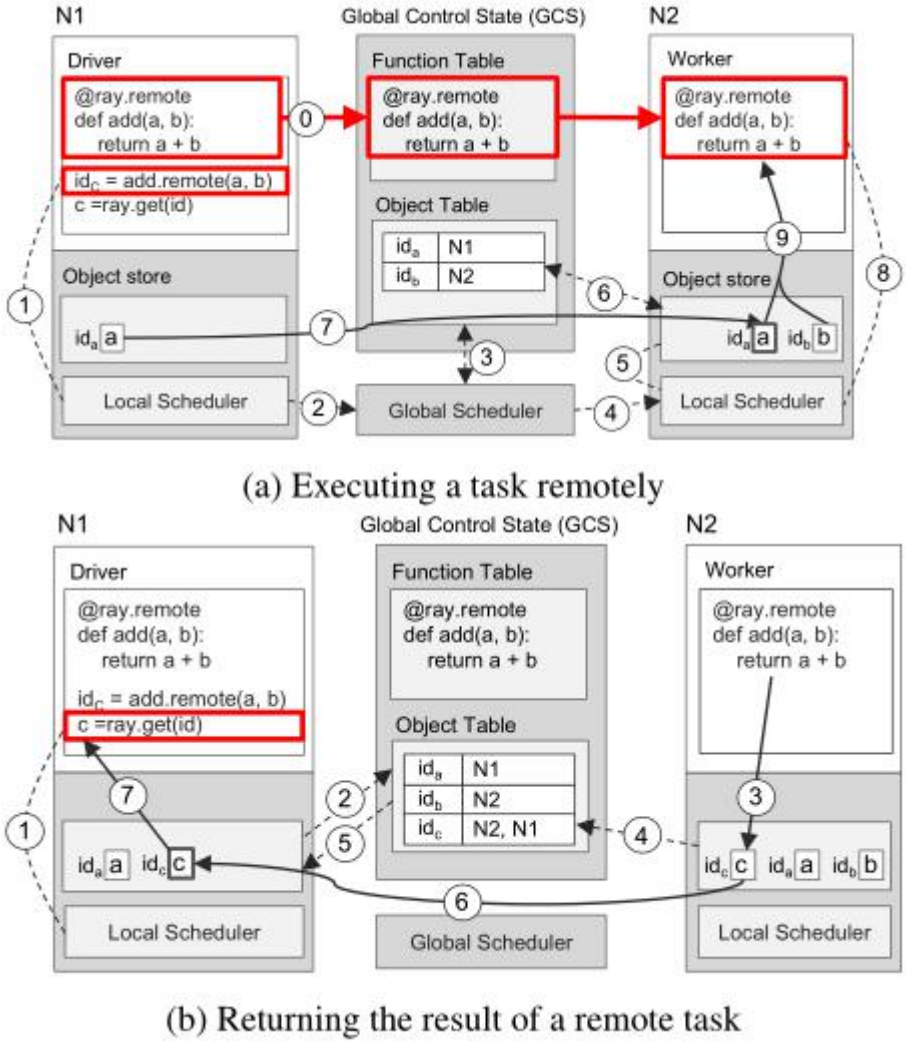


Figure 6: 输入 a 和 b 并返回 c 的端到端示例。 实线是数据平面操作，虚线是控制平面操作。（a）函数 add()由节点 1(N1)在 GCS 上注册，在 N1 上调用，并在 N2 上执行。（b）N1 使用 ray.get()获取 add()的结果。c 的对象表条目在步骤 4 中创建，并在将 c 复制到 N1 后步骤 6 中更新。

4.1 应用层

应用程序层由三部分组成：

1.驱动程序：执行用户程序的进程。

2.Worker（工作对象）：执行由驱动程序或其他工作对象调用的任务（远程函数）的无状态进程。系统层自动启动工作对象并分配任务。当声明远程函数时，该函数会自动发布给所有工作对象。worker 连续执行任务。

3.Actor：一个有状态的进程，当它被调用时执行它暴露的方法。与 worker 不同，actor 由 worker 或 driver 明确实例化。和 worker 一样，actor 连续执行方法。

请注意，worker 是没有状态的，因为它们不会在任务中保持本地状态。假设执行确定性任务，调用具有相同参数的相同远程函数将返回相同的结果，而不管它是否在同一个 worker

上执行。 相比之下，actor 是一个有状态的过程，方法调用的结果可能依赖于该 actor 执行的先前方法。

4.2 系统层

系统层使我们能够满足性能和容错目标，如第 2 节所述，采用一种架构，其中每个组件均可水平扩展和容错。该层由三个主要组件组成：全局控制存储区，分布式调度程序和分布式对象存储区。

4.2.1 全局控制存储区

其核心是我们的系统利用全局控制存储（GCS），该存储将所有最新的元数据和控制状态信息存储在系统中。这包括（1）每个任务的规范，（2）每个远程函数的代码，（3）图计算，（4）所有对象的当前位置，（5）每个调度事件。GCS 还提供发布订阅基础设施以促进组件之间的通信。

通过以集中方式存储和管理整个控制状态，GCS 使每个其他组件都成为无状态。这不仅简化了对容错的支持（即失败时，组件重新启动并从 GCS 读取最新状态），而且还可以轻松地对每个其他组件进行横向扩展，因为组件的副本或碎片共享的所有状态可通过 GCS 访问。

为了扩展 GCS，我们使用分片。由于我们可以将伪随机 ID 实际上与 GCS 中的每个数据条目（例如对象，任务，函数）相关联，因此跨多个分片平衡负载相对容易。为了提供容错功能，我们为每个分片使用动态复制副本。

集中系统控制信息使我们能够在 GCS 之上轻松构建调试，分析和可视化工具。我们迄今为止制作的简约工具在我们的开发中已经证明是有用的。

4.2.2 自下而上的分布式调度器

许多现有的集群计算框架（如 Apache Spark，CIEL，Dryad 和 Hadoop）都实现了集中式调度程序。虽然这简化了设计，但它损害了可扩展性。

有几种方法可以提高调度可扩展性：（1）批量调度，其中调度程序批量向 worker 节点提交任务以分摊与任务提交相关的固定开销（例如 Drizzle）；（2）分层调度，其中全局调度器将任务图分割成跨每个节点的本地调度器（例如，Canary）；（3）并行调度，其中多个全局调度器在所有 worker 节点（例如，Sparrow）上同时调度任务。不幸的是，这些方法都不符合 Ray 的要求。批量调度仍然需要全局调度器来处理每个任务，这限制了其可伸缩性，分层调度假定任务图是事先已知的（即图是静态的），并行调度假定每个全局调度器调度独立作业。相比之下，我们需要高度可扩展的调度程序来处理动态任务图，该动态任务图可能由单个作业生成。

像现有的分层调度解决方案一样，我们采用全局调度器和各节点本地调度器。但是，与以前的解决方案不同，节点上创建的任务首先提交给节点的本地调度程序，而不是全局调度程序（Figure 5）。本地调度程序在本地调度任务，除非节点超载，或者它不能满足任务的要求（例如，缺少 GPU），或者任务的输入是远程的。如果本地调度程序不安排任务，它将任务发送到全局调度程序。为了确定负载，本地调度程序检查其任务队列的当前长度。如果长度

超过某个可配置的阈值，则认为本地节点过载。这个阈值的设置，当所有任务被移交给全局调度器时，可以使调度策略保持跨越连续集中，当所有任务在本地处理时，分散到各地。

每个本地调度程序定期向包含其负载信息的 GCS 发送心跳（例如每 100ms）。 GCS 记录此信息并将其转发给全局调度程序。接收到任务后，全局调度程序使用来自后续节点的最新负载信息以及任务输入的位置和大小（来自 GCS 的对象元数据）来决定将任务分配给哪个节点。如果全局调度程序成为瓶颈，我们可以实例化更多副本，并让每个本地调度程序随机选择一个副本来发送其任务。这使我们的调度程序体系结构高度可扩展。

4.2.3 内存分布式对象存储

为了尽量减少任务延迟，我们实施了内存分布式存储系统来存储每项任务的输入和输出。这使 workers 和 actors 能够有效地共享数据。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共享零拷贝数据。另外，我们使用 Apache Arrow，这是一种高效的内存布局，正在成为数据分析中事实上的标准。

如果任务的输入不是本地的，则在执行之前将输入复制到同一节点上的本地对象存储。任务还将所有输出写入本地对象存储。由于任务只能在本地内存中读写数据，因此复制消除了由于热数据对象而导致的潜在瓶颈并最大限度地减少了任务执行时间。这增加了计算绑定工作负载的吞吐量，这是许多 AI 应用程序共享的配置文件。

现有的集群计算框架（如 Apache Spark 和 Dryad），其对象存储局限于不可变数据，系统简化设计，通过避免需要复杂的一致性协议（避免并发更新），简化支持容错。

为了简单起见，我们的对象存储不支持分布式对象，也就是说，每个对象都适合单个节点。像大矩阵或树这样的分布式对象可以作为 future 的集合在更高的层次（例如，应用层）上实现。

对象重建。组件故障可能导致对象丢失，Ray 通过血统重新执行恢复。Ray 通过在执行过程中记录 GCS 中的任务依赖关系来跟踪血缘。这与 Apache Spark 和 CIEL 等其他集群计算系统采用的解决方案类似。此外，与这些系统一样，Ray 假定对象是不可变的，而运算符（即远程函数和 actor 方法）是确定性的。但是，与这些系统不同，Ray 增加了对有状态操作者（actor）重建的支持。通过将状态边直接集成到图计算中，我们可以为远程函数和 actor 使用相同的重构机制。

为了重建一个丢失的对象，我们沿着数据和有状态边向后执行，直到找到其输入全部出现在对象存储区中的任务。然后我们重复以这些输入为根的计算子图。考虑图 Figure 3（b）中的例子，并假设 rollout12 已经丢失。通过沿着数据和有状态边向后走，我们到达没有输入的 A10。因此，为了重建 rollout12，我们需要通过执行 A10 来重新实例化 actor，然后按顺序执行方法 A11 和 A12。

注意，对于其血缘包括有状态边缘的任何对象，重构将需要重新实例化该 actor（例如 A10）并重新执行可能长的有状态边缘链（例如，A11; A12 等）。由于 actor 通常用于包装具有有限生命周期的第三方模拟器，因此我们预计这些链条是有界的。然而，我们也发现 actor 对管理更一般的状态很有用。为了在这种情况下提高恢复时间，我们定期检查 actor 的状态，

并允许 actor 从检查点恢复。

为了低延迟，我们将对象完全保存在内存中，并使用最近最少使用的释放策略将它们从内存中释放。

4.3 将所有聚集一起

Figure 6 展示了 Ray 如何通过一个简单的例子进行端对端的工作，该例子添加了两个对象 a 和 b，它们可以是标量或矩阵，并返回结果 c。远程函数 add()在初始化时自动向 GCS 注册，并分发给系统中的每个 worker（Figure 6（a）中的步骤 0）。

Figure 6（a）显示了 driver 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。driver 程序向本地调度器提交 add(a，b)（步骤 1），该调度器将其转发给全局调度器（步骤 2）.接下来，全局调度器查找 add(a，b)参数的位置在 GCS 中（步骤 3）并且决定在存储参数 b 的节点 N2 上调度任务（步骤 4）。节点 N2 处的本地调度器检查本地对象库是否包含 add(a，b)的参数（步骤 5）。由于本地存储没有对象 a，因此它在 GCS 中查找 a 的位置（步骤 6）。了解 a 存储在 N1 中，N2 的对象存储在本地复制（步骤 7）。由于 add()的所有参数现在都存储在本地，本地调度程序在本地 worker 上调用 add()（步骤 8），它通过共享内存访问参数（步骤 9）。

Figure 6（b）分别显示了执行 N1 时的 ray.get()和 N2 时的 add()触发的逐步操作。在 ray.get(idc)的调用之后，驱动程序使用 add()返回的 future idc（步骤 1）检查本地对象存储库中的值 c。由于本地对象存储不存储 c，它会在 GCS 中查找它的位置。此时，c 没有条目，因为 c 尚未创建。结果，N1 的对象存储注册了一个回调对象表，在创建 c 的入口时触发（步骤 2）。同时，在 N2 中，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 用 c 的输入触发对 N1 的对象存储的回调（步骤 5）。接下来，N1 从 N2 复制 c（步骤 6），并将 c 返回给 ray.get()（步骤 7），最终完成该任务。

虽然这个例子涉及大量的 RPC，但在很多情况下，这个数字要小得多，因为大多数任务都是在本地调度的，并且 GCS 响应由全局和本地调度器缓存。

5 实现

Ray 在 4 万代码行（LoC）中实现，在系统层的 C ++中为 72％，在应用程序层中的 Python 为 28％。对象存储库和我们的零拷贝序列化库已作为可独立于 Ray 使用的独立项目进行了分解。

自下而上的分布式调度程序（第 4.2.2 节）为 3.2KLoC，并将在我们改进 Ray 的调度策略时进行重大开发。在本节中，我们重点讨论实现实时 AI 应用程序所实现的性能目标的实现细节：（a）调度程序性能，（b）对象存储库性能和（c）端到端系统性能。

自下而上的分布式调度器。我们将本地和全局调度程序实现为事件驱动的单线程过程。在内部，本地调度程序维护本地对象元数据的缓存状态，等待输入的任务以及准备派发给 worker 的任务。随着对象依赖关系变得可用，任务已准备好用于分派。Worker 可用性触发了在节

点容量限制下尽可能多的任务分派。

本地调度程序定期向全局调度程序发送心跳（每 100ms），通过发布订户机制经过 GCS，其中包含调度队列长度和资源可用性。这使全局调度程序能够平衡各节点间的负载。
对象存储。Ray 的对象存储也被实现为单线程事件循环。它使用共享内存，因此同一节点上的 worker 无需复制即可读取数据。对象是不可变的。一个对象只有在 worker 完成创建后才可见。为了最大限度地减少对象创建开销，应用商店预分配一个大型内存映射文件池。我们使用类似 SIMD 的内存副本来最大化将数据从 worker 复制到对象存储的共享内存的吞吐量。我们也并行计算一个对象的内容哈希，它用于检测非确定性计算。Ray 使用 Apache Arrow 在序列化/反序列化 Python 对象时实现高性能。

全局控制存储。我们使用每个分片的一个 Redis 键值存储（Redis 中可以轻松地与其他键值存储交换）实施 Ray 的全局控制存储（GCS）。我们通过对象和任务 ID 对 GCS 表进行分片以扩展，并且我们复制每个分片以进行容错。随着我们扩大实验范围，我们在多个节点上分配碎片。尽管我们的 GCS 实现使用多个 Redis 服务器，但我们的性能和容错要求也可以通过像 RAMCloud 这样的现有系统来满足。

最后，Ray 的监视器跟踪系统组件的存活性并反映 GCS 中的组件故障。必要时，失败群集节点上的任务和对象被标记为丢失，并且随后使用血缘信息重建对象。

6 评估

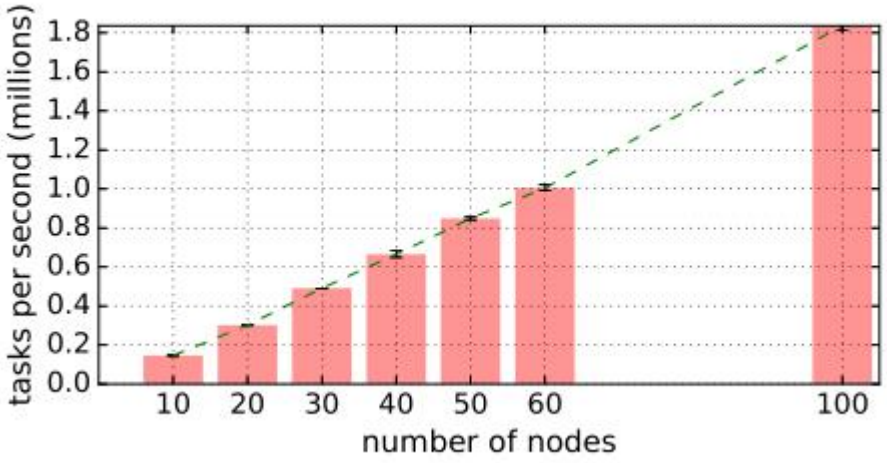


Figure 7: 利用 GCS 和自下而上的分布式调度程序，系统的端到端可扩展性以线性方式实现。Ray 利用 60 m4.16xlarge 节点达到每秒 100 万个任务的吞吐量并在一分钟内处理 1 亿个任务。

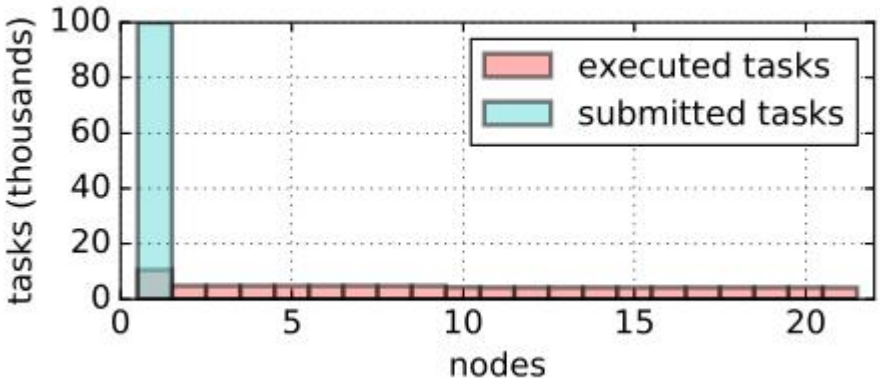


Figure 8: Ray 保持平衡负载。 第一个节点上的驱动程序提交 100K 个任务，这些任务由全局调度程序在 21 个可用节点之间重新平衡。

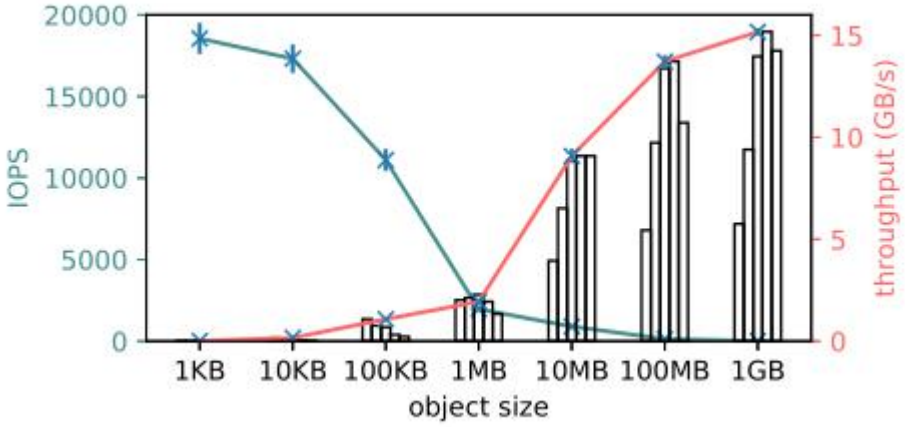


Figure 9: 对象存储写入吞吐量和 IOPS。对于 16 核心实例（m4.4xlarge）上的小型对象，单个客户端的吞吐量超过大型对象的 15GB / s（红色）和 18K IOPS（青色）。它使用 8 个线程来复制大于 0.5MB 的对象和 1 个用于小对象的线程。条形图以 1,2,4,8,16 线程报告吞吐量。结果在 5 次运行中取平均值。

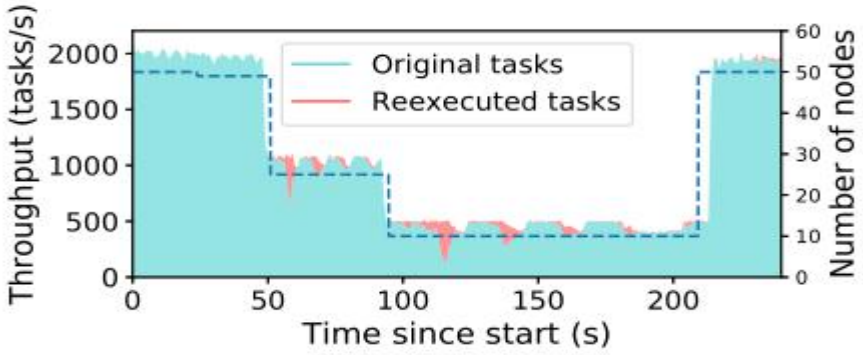


Figure 10: 完全透明的分布式任务容错。 虚线表示群集中的节点数量。 曲线显示新任务（青色）和重新执行的任务（红色）的吞吐量。Driver 不断提交和检索 10000 个任务。每个任务需要 100ms，并取决于前一轮中的任务。每个任务都有大小为 100KB 的输入和输出。

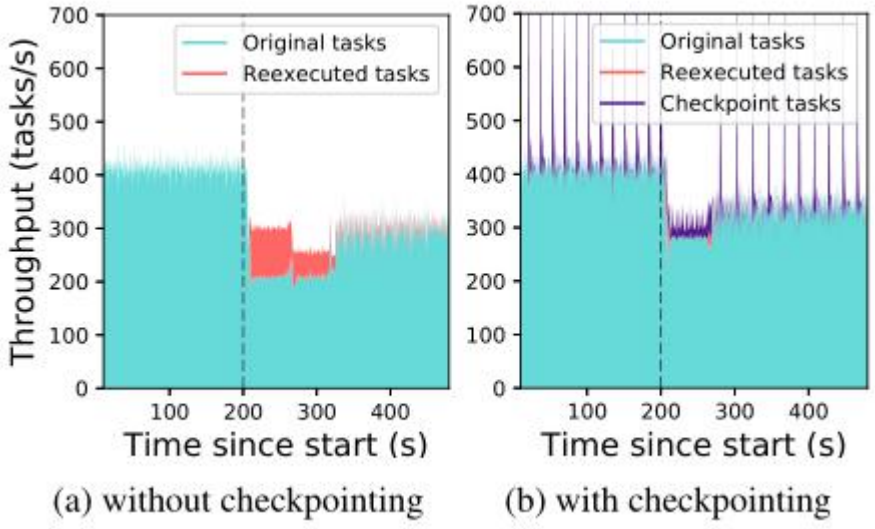


Figure 11: Actor 方法的完全透明容错。Driver 不断向集群中的参与者提交任务。在 t = 200s 时，我们杀死 10 个节点中的 2 个，导致群集中 2000 个 Actor 中的 400 个在其余节点上恢复。

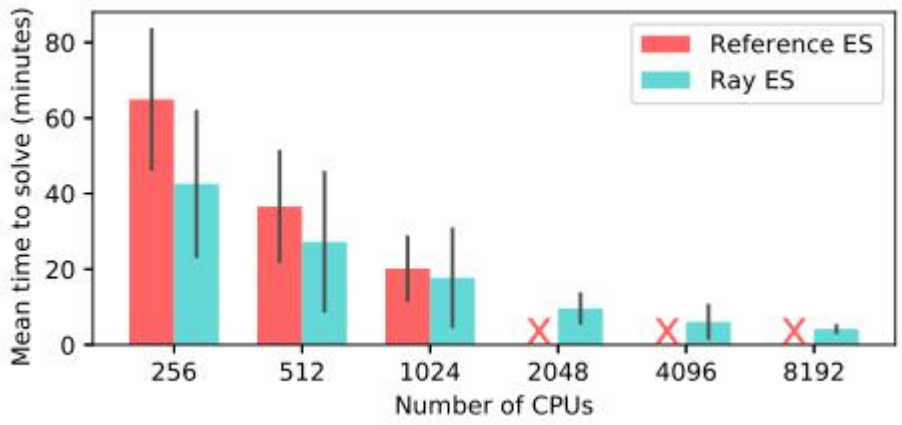


Figure 12: 在 Humanoid-v1 任务中达到 6000 分的时间。Ray ES 实现可以很好地扩展到 8192 个内核。专用系统无法运行超过 1024 个内核。使用 8192 个内核，我们的平均时间达到了 3.7 分钟，是最佳发布结果的两倍。在此基准测试中，ES 比 PPO（第 6.3.2 节）更快，但是运行时变化更大。

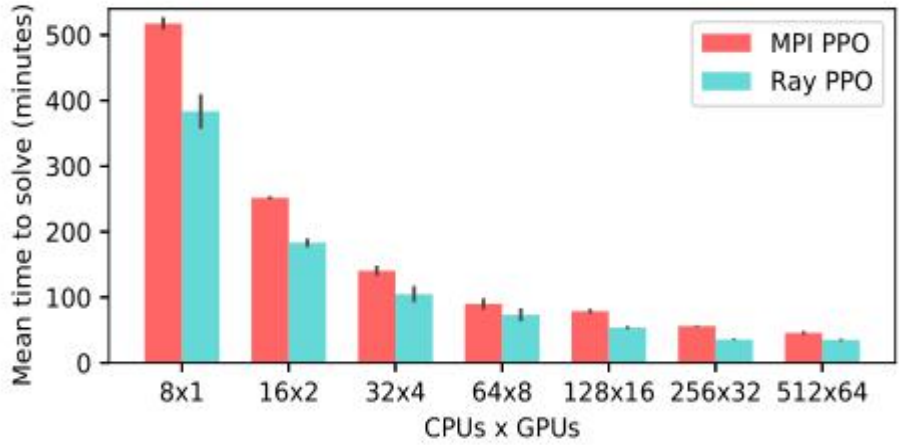


Figure 13: 在 Humanoid-v1 任务中达到 6000 分的时间。Ray PPO 实现的性能优于专用 MPI 实现，GPU 数量更少，成本也更低。MPI 的实现需要每 8 个 CPU 有 1 个 GPU，而 Ray 版本最多需要 8 个 GPU（每 8 个 CPU 不超过 1 个 GPU）。

在本节中，我们将展示三个关键点。首先，我们考察整个系统的可扩展性以及各个组件的性能（第 6.1 节）。其次，我们展示了鲁棒性和容错性（第 6.2 节）。第三，我们证明了 Ray 在强化学习应用方面非常合适，无论是在性能还是开发方面（第 6.3 节）。所有实验均在亚马逊网络服务上运行。具体的实例类型如下所示。

6.1 可扩展性和性能

端到端的可扩展性。全局控制存储（GCS）的主要优势之一是可以横向扩展系统。我们在这一节评估这种能力。在 Figure 7 中，我们测试了一个困难的并行工作负载，增加了 x 轴上的簇大小。我们观察到逐渐增加的任务吞吐量接近完美的线性。Ray 在 60 个节点上每秒吞吐量超过 100 万个任务，并且在 100 个节点上继续线性扩展到每秒 180 万个任务以上。最右边的数据点显示 Ray 可以在不到一分钟（54 秒）内处理 1 亿个磁道任务。可变性（用黑色误差线显示）很小。正如预期的那样，随着任务持续时间的增加，吞吐量会按比例降低平均任务持续时间，但整体可扩展性保持线性。

全球调度程序的主要职责是在整个系统中保持均衡的负载。在 Figure 8 中，单个节点上提交的 100K 个任务在可用资源之间进行重新平衡。请注意，负载源的节点处理更多任务，因为它在将任务转发到全局调度程序之前最大化本地节点的利用率。

对象存储的性能。我们对比对象存储性能的两个度量标准：Figure 9 中的 IOPS（针对小对象）和写吞吐量（针对大对象）。随着对象大小的增加，单个客户端的写入吞吐量达到 15GB / s。对于较大的对象，从客户端复制对象支配对象创建的时间。对于较小的对象，完成时间由客户端和对象存储之间的序列化开销和 IPC 决定。对象存储峰值为 18K IOPS，相当于每次操作花费 56µs。

6.2 容错

从对象故障中恢复。在 Figure 10 中，我们演示了 Ray 从 worker 节点故障中透明恢复并弹性扩展的能力。驱动程序提交几轮任务，其中每个任务都依赖于前一轮中的任务。当 Worker 节点被杀死（在 25s，50s，100s）时，幸存的本地调度器会自动触发重建丢失的对象。在重建期间，Driver 最初提交的任务不运行，因为它们的依赖不能得到满足。但是，总体任务吞吐量保持稳定，充分利用可用资源，直到重建丢失的依赖关系。此外，随着更多节点在 210s 时加回系统，Ray 能够完全恢复到其初始吞吐量。

从 actor 失败中恢复。接下来，我们演示 Ray 的透明恢复失去的 actor 的能力。通过将每个参与者的方法调用编码到依赖关系图中，我们可以重用与 Figure 10 中相同的对象重构机制。Figure 11a 中的工作负载演示了没有中间参与者状态被保存的极端情况。之前对每个失去 actor 的方法调用必须连续重新执行（t = 210-330s）。失去的 actor 会自动在可用节点间重新分配，吞吐量在重建后会完全恢复。

为了改善长期存活 actor 的重建时间，我们提供了中间角色状态的透明检查点。Figure 11b 显示了相同的工作负载，但每 10 个方法调用每个角色的自动检查点任务。初始吞吐量与没有检查点的吞吐量相当。节点失效后，大部分重建都是通过执行检查点任务来完成的，从而重建 actor 的状态（t = 210-270s）。结果，只有 500 个任务需要重新执行，而新方法调用分别停顿了 60 秒，而重新执行了 10K 次，而没有检查点的 120 秒。未来，我们希望进一步减少 actor 重建时间，例如，通过允许用户注释为只读方法。

来自 GCS 备份的开销。为了使 GCS 容错，我们复制每个数据库碎片。当客户端写入 GCS 的其中一个分片时，它会将写入复制到所有副本。对于通过减少 GCS 碎片数量来人为地减少使 GCS 成为瓶颈的工作负载，双向复制的开销不到 10％。在大多数实际的工作量中，减速是无法察觉的。

6.3 强化学习应用

鉴于第 2 节中描述的强化学习应用程序的多样化和苛刻的要求，今天的强化学习算法在特殊用途 ad-hoc 系统之上实现，这些特殊用途 ad-hoc 系统通常需要大量的工程设计开发工作，而不会推广到其他算法。

在本节中，我们在 Ray 中实现了两种强化学习算法，并表明我们能够匹配或优于专门为这些算法构建的专用系统的性能。此外，使用 Ray 在集群上分发这些算法需要在算法的串行实现中仅更改几行代码。另外，我们在一个对延迟敏感的设置中测试 Ray，在该设置中，Ray 用于在不同的实时要求下控制模拟机器。

6.3.1 进化策略

为了在大规模的 RL 工作负载上评估 Ray，我们实现了演化策略（ES）算法，并与参考实现进行了比较，该参考实现是为该算法构建的专用系统。它使用一系列 Redis 服务器作为消息总线，并依靠低级多处理库来共享数据。

如 Figure 12 所示，在 Ray 之上的一个简单的实现是可扩展的，扩展到 8192 个物理内核，而专用系统在 1024 个内核之后停止运行。Ray 实施的中位时间为 3.7 分钟，是最佳公布结果（10 分钟）的两倍。Ray 的实现也大大简化了开发。使用 Ray 并行化串行实现需要修改 7 行代码。相比之下，参考实现需要数百行代码来开发用于在工作人员之间传递任务和数据的自定义协议，并且不能容易地适应不同的算法或通信模式。我们在 B.1 节中包含说明这一点的伪代码。

6.3.2 近端策略优化

为了在单节点和小型集群 RL 工作负载上评估 Ray，我们在 Ray 中实现了近端策略优化（PPO），并与使用 OpenMPI 通信原语的高度优化的参考实现进行了比较。所有实验均使用 p2.16xlarge（GPU）和 m4.16xlarge（高 CPU）实例，每个实例都有 32 个物理核心。

Ray 的 API 可以轻松利用异构资源将成本降低 4.5 倍。Ray 任务和参与者可以指定不同的资源需求，从而允许在廉价的高 CPU 实例上调度纯 CPU 任务。相比之下，MPI 应用程序通常具有对称体系结构，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，防止仅使用 CPU 的机器进行横向扩展。

如 Figure 13 所示，Ray 实现在所有实验（部分 D 中列出的超参数）中用一部分 GPU 执行优化的 MPI 实现。和 ES 一样，我们能够使用 Ray 对 PPO 进行并行化，并且对串行程序的结构

进行最小限度的更改。

6.3.3 控制模拟机器

我们展示 Ray 可以通过实时控制模拟机器来满足软实时要求。Ray 驱动程序运行模拟的机器，并以固定的时间步长从 1 毫秒变为 30 毫秒，以模拟不同的实时需求。驱动程序提交的任务将使用线下训练的策略来计算要采取的操作。但是，只有在相关时间段内 driver 收到这些动作（否则先前的动作重复），才会采取行动。真实机器的延迟预算约为 10 毫秒，我们发现，即使我们运行模拟的速度比实时更快（使用 3 毫秒的时间步长），Ray 也能够产生稳定的行为。表 3 显示了没有足够快到达机器的任务的比例。

Time budget (ms)	30	20	10	5	3	2	1
% actions dropped	0	0	0	0	0.4	40	65
Stable walk?	Yes	Yes	Yes	Yes	Yes	No	No

Table 3: Low-latency robot simulation results

7 相关工作

动态任务图。 Ray 与 CIEL 密切相关。它们都支持具有嵌套任务的动态任务图，实现 future 抽象，并提供基于线性的容错。但是，它们在两个重要方面有所不同。首先，Ray 通过 actor 抽象扩展任务模型。其次，Ray 采用完全分布式的控制平面和调度器，而不是依靠单个主控。此外，Ray 还添加了 ray:wait()方法，采用内存中（而不是基于文件）对象存储，并扩展了现有的编程语言（Python），而 CIEL 提供了自己的脚本语言（Skywriting ）。 Ray 还与 Dask 密切相关，Dask 支持动态任务图形，包括一个等待原语，并在 Python 环境中使用 future 抽象。但是，Dask 使用集中调度程序，不提供类似 actor 的抽象，也不提供容错功能。

数据流系统。流行的数据流系统（如 MapReduce，Spark 和 Dryad）广泛采用分析和 ML 工作负载，但其计算模型更具限制性。Spark 和 MapReduce 实现了 BSP 执行模型，该模型假定同一阶段内的任务执行相同的计算并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，这些系统都没有提供 actor 抽象，也没有实现分布式可扩展控制平面和调度器。最后，Naiad 是一个数据流系统，可为某些工作负载提供改进的可扩展性，但仅支持静态任务图。

Actor 系统。Orleans 提供了一个虚拟的基于 actor 的抽象。Actor 是永久的，它们的状态持续在调用之中。为了扩展，Orleans 还允许 actor 的多个实例在 actor 以不可变状态操作或没有状态时并行运行。这些无状态的 actor 可以担任 Ray 的任务。但是，与 Ray 不同的是，Orleans 开发者必须明确检查点角色状态和中间响应。此外，Orleans 还提供了至少一次的语义。相比之下，Ray 提供了透明容错和一次语义，因为每个方法调用都记录在 GCS 中，并且参数和结果都是不可变的。我们发现实际上这些限制不会影响我们应用程序的性能。

Erlang 和 C ++ Actor Framework（CAF）是另外两个基于 actor 的系统，它们还要求应用程序明确处理容错。而且，Erlang 的全局状态存储不适合共享大型对象，如 ML 模型，而 CAF 不支持数据共享。

全局控制状态和调度。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）中，先前已经提出了逻辑集中控制平面的概念，Ray 从这些开拓性努力中汲取灵感，但提供了重大改进。与耦合控制平面数据和计算的 SDN，BOOM 和 GFS 相比，Ray 将控制平面信息（例如，GCS）的存储与逻辑实现（例如调度器）分开解耦。这允许存储层和计算层独立扩展，这对于实现我们的可扩展性目标至关重要。Omega 使用分布式体系结构，调度程序通过全局共享状态进行协调。对于这种体系结构，Ray 添加了全局调度程序来平衡跨本地调度程序的负载，并针对 ms 级而非二级任务调度。

Ray 实现了一个可水平扩展的独特的自下而上分布式调度程序，并且可以处理动态构建的任务图。与 Ray 不同，大多数现有的集群计算系统使用集中式调度程序体系结构。虽然 Sparrow 是分散式的，但它的调度程序会做出独立决定，限制可能的调度策略，并且所有任务都由同一个全局调度程序处理。Mesos 实现了一个两级分层调度器，但是它的顶级调度器可能是一个瓶颈。Canary 通过让每个调度程序实例处理任务图的一部分来实现令人印象深刻的性能，但不处理动态计算图。

机器学习框架。 TensorFlow 和 MXNet 针对深度学习工作负载，并有效利用 CPU 和 GPU。虽然它们对由线性代数运算的静态 DAG 组成的工作负载实现了出色的性能，但它们对更一般的工作负载的支持有限。TensorFlow Fold 通过其内部 C ++ API 为动态任务图和 MXNet 提供了一些支持，但它们都不支持在执行期间修改 DAG 以响应任务进度、任务完成时间或故障的能力。原则上 TensorFlow 和 MXNet 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，但是这种情况下的陷阱和用户体验与 MPI 的类似。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，它迫使程序员明确处理容错。

8 讨论和经验

自从几个月前我们发布 Ray 以来，已有超过一百万的人下载并使用它。在这里，我们讨论了我们开发和使用 Ray 的经验，以及我们从早期用户那里收到的一些反馈。

API。在设计 API 时，我们强调极简主义。最初我们从基本任务抽象开始。后来，我们添加了 wait()基元来适应具有不同持续时间的 rollouts，并且 actor 抽象可以容纳第三方模拟器，并分摊昂贵的初始化开销。虽然由此产生的 API 相对初级，但它已被证明功能强大且易于使用。实际上，有些团队报告指示开发人员首先编写串行实现，然后使用 Ray 将其并行化。

为了说明这一点，接下来我们简要描述我们的两种其他算法的经验：异步优势 Actor Critic（A3C）和超参数搜索.A3C 是一种最先进的 RL 算法，利用异步策略更新来显著提高训练时间以前的算法。为了扩展这个算法，我们使用一个简单的分层方案，其中 A3C 的多个实例被并行训练并周期性地聚合以形成改进的模型。在 Ray 中实现分层 A3C 非常简单，需要 20 行 Python 代码才能扩展非分层版本。此外，这种简单的扩展将相同硬件的性能提高了 30％。

我们能够使用 Ray 在大约 30 行 Python 代码中实现最先进的超参数搜索算法。Ray 对嵌套任务的支持非常关键，因为多个实验必须并行运行，并且每个实验通常在内部使用并行。 wait()原型允许我们按照完成的顺序处理实验结果并自适应地启动新实验。Actor 抽象允许我们暂

停并恢复基于其他实验进展的有状态实验（参见第 B.3 节）。相反，大多数现有的实现必须等待所有的实验完成，这导致资源利用效率低下。

Ray 的 API 工作仍然在进行中。基于早期的用户反馈，我们正在考虑增强 API 以包含更高级别的基元，例如简单聚合和映射。这些也可以通知 Ray 系统层的调度决策（4.2 节）。

限制。鉴于工作负载普遍性，专业优化很难。例如，我们必须在不完全知道图计算的情况下做出调度决策。Ray 中的计划优化可能需要更复杂的运行时概要分析。此外，为每个任务存储线性血缘，需要实施垃圾回收策略来减少 GCS 中的存储成本，这是我们正在积极开发的一项研究。

容错。我们经常被问到 AI 应用程序是否真的需要容错功能。毕竟，由于许多人工智能算法的统计性质，人们可以简单地忽略失败的展示。根据我们的经验，我们的答案是不合格的“是”。首先，忽略失败的能力使得应用程序更容易编写和推理。其次，我们通过确定性重新运行，能够大大简化调试过程，因为它使我们能够轻松地重现大多数错误。这一点尤其重要，因为由于它们的随机性，人工智能算法出人意料地难以调试。第三，容错功能有助于节省资金，因为它允许我们使用 AWS 上的现货实例等廉价资源运行。而且，随着工作负载的扩大，我们期望容错变得更加重要。当然，这是以一些开销为代价的。但是，我们发现这种开销对于我们的目标工作负载来说是最小的。

GCS 和水平伸缩性。 GCS 极大地简化了 Ray 开发和调试。所有其他组件的基本故障处理和水平缩放都需要不到一周时间才能实施。GCS 使我们能够在调试时查询整个系统状态（而不必手动公开内部组件状态）。这帮助我们发现了许多错误并且了解系统行为。

GCS 有助于 Ray 的水平可伸缩性。在 6.1 节中报告的实验中，只要 GCS 成为瓶颈，我们就可以通过添加更多的分片来扩展结果。通过简单地添加更多副本，GCS 还可以使全局调度程序进行扩展。虽然目前我们正在手动配置 GCS 分片和全局调度器的数量，但我们计划在未来开发自适应算法。由于这些优势，我们认为集中控制状态将成为未来分布式系统的关键设计组件。

9 结论

新兴的 AI 应用程序提出了挑战性的计算需求。为了满足这些需求，Ray 引入了全局控制存储和自下而上的分布式调度程序。这个架构实现动态任务图执行，而且反过来支持任务并行和 actor 编程模型。这种编程灵活性对于 RL 工作负载尤为重要，因为 RL 工作负载在资源需求、持续时间和功能方面产生各种任务。我们的评估表明，每秒可执行 1M 任务的线性可伸缩性，透明的容错能力以及对若干当代 RL 工作负载的显着性能改进。 因此，Ray 为开发未来的 AI 应用程序提供了灵活性、性能和易用性。