

## Ray: 새로운 AI 애플리케이션을 위한 분산 프레임워크

Philipp Moritz\*, Robert Nishihara\*, Stephanie Wang, Alexey Tumanov, Richard Liaw,  
Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, Ion Stoica  
University of California, Berkeley

### 추상적인

차세대 AI 애플리케이션은 지속적으로 환경과 상호 작용하고 이러한 상호 작용을 통해 학습합니다. 이러한 응용 프로그램은 성능과 유연성 측면에서 새롭고 까다로운 시스템 요구 사항을 부과합니다. 이 백서에서는 이러한 요구 사항을 고려 하고 이를 해결하기 위한 분산 시스템인 Ray를 제시합니다.

Ray는 단일 동적 실행 엔진 에서 지원하는 작업 병렬 및 액터 기반 계산을 모두 표현할 수 있는 통합 인터페이스를 구현합니다. 성능 요구 사항을 충족하기 위해 Ray는 분산 스케줄러 와 분산 및 내결함성 저장소를 사용하여 시스템의 제어 상태를 관리합니다. 실험에서 우리는 초당 180만 작업 이상의 확장 전략과 여러 까다로운 강화 학습 응용 프로그램에 대해 기존의 특수 시스템보다 더 나은 성능을 보여줍니다.

예측에 사용합니다. 이러한 프레임워크는 종종 배치 설정에서 교육 시간을 줄이기 위해 오래된 특수 하드웨어(예: GPU 및 TPU)를 활용합니다. TensorFlow[7], MXNet[18] 및 PyTorch[46]를 예로 들 수 있습니다.

그러나 AI의 가능성은 고전 적인 감독 학습보다 훨씬 광범위합니다. 새로운 AI 애플리케이션은 점점 더 역동적인 환경에서 작동하고, 환경의 변화에 대응하고, 장기적인 목표를 달성하기 위해 일련의 조치를 취해야 합니다 [8, 43]. 그들은 수집된 데이터를 활용하는 것뿐만 아니라 가능한 조치의 공간을 탐색하는 것을 목표로 해야 합니다. 이러한 광범위한 요구 사항은 강화 학습(RL)의 패러다임 내에서 자연스럽게 구성됩니다. RL은 자연되고 제한된 피드백을 기반으로 불확실한 환경 내에서 지속적으로 작동하는 학습을 다룹니다 [56]. RL 기반 시스템은 Google의 AlphaGo가 인간 세계 챔피언을 이기는 것과 같은 놀라운 결과를 이미 얻었으며 [54] 대화 시스템, UAV [42] 및 로봇 조작[25, 60] 으로 진출하기 시작했습니다.

### 1. 소개

지난 20년 동안 많은 조직에서 점점 더 많은 양의 데이터를 수집했으며 이를 활용하는 것을 목표로 삼았습니다. 이로 인해 배치 [20, 64, 28], 스트리밍 [15, 39, 31] 및 그래프 [34, 35, 24] 처리 시스템을 포함하여 분산 데이터 분석을 위한 수많은 프레임워크가 개발되었습니다. 이러한 프레임 워크의 성공으로 조직은 대규모 데이터 세트를 비즈니스 또는 과학적 전략의 핵심 부분으로 분석할 수 있게 되었으며 "빅 데이터" 시대를 열었습니다.

보다 최근에는 데이터 중심 애플리케이션의 범위가 보다 복잡한 AI(인공 지능) 또는 ML(기계 학습) 기술을 포함하도록 확장되었습니다 [30].

패러다임 사례는 데이터 포인트에 레이블이 수반되고 데이터 포인트를 레이블에 매핑하는 주력 기술이 심층 신경망에 의해 제공되는 지도 학습의 경우입니다. 이러한 심층 네트워크의 복잡성으로 인해 심층 신경망 교육에 중점을 둔 또 다른 프레임 작업이 발생했습니다.

RL 응용 프로그램의 중심 목표는 시간 경과에 따라 효과적인 성능(예: 게임 승리 또는 드론 조종)을 생성하는 정책(환경 상태에서 작업 선택으로의 매핑)을 학습하는 것입니다. 대규모 애플리케이션에서 효과적인 정책을 찾으려면 세 가지 주요 기능이 필요합니다. 첫째, RL 방법은 종종 시뮬레이션에 의존하여 정책을 평가합니다. 시뮬레이션을 통해 다양한 행동 순서 선택을 탐색하고 이러한 선택의 장기적인 결과에 대해 배울 수 있습니다.

둘째, 감독 학습과 마찬가지로 RL 알고리즘은 시뮬레이션 또는 물리적 환경과의 상호 작용을 통해 생성된 데이터를 기반으로 정책을 개선하기 위해 분산 교육을 수행해야 합니다. 셋째, 정책은 문제를 제어하기 위한 솔루션을 제공하기 위한 것이므로 대화식 패쇄 루프 및 개방 루프 제어 시나리오에서 정책을 제공하는 것이 필요합니다.

이러한 특성은 새로운 시스템 요구 사항을 주도합니다. RL용 시스템은 세분화된 계산(예: 현실 세계와 상호 작용할 때 밀리초 단위로 작업을 렌더링하고 방대한 수의 시뮬레이션을 수행)을 지원해야 합니다.

시뮬레이션), 시간(예: 시뮬레이션은 밀리초 또는 몇 시간이 걸릴 수 있음)과 리소스 사용(예: 교육용 GPU 및 시뮬레이션용 CPU) 모두에서 이질성을 지원해야 하며 시뮬레이션 또는 상호 작용의 결과로 동적 실행을 지원해야 합니다. 환경과 함께 향후 계산을 변경할 수 있습니다. 따라서 밀리초 수준의 대기 시간에서 초당 수백만 개의 이기종 작업을 처리하는 동적 계산 프레임워크가 필요합니다.

빅 데이터 워크로드 또는 지도 학습 워크로드용으로 개발된 기존 프레임워크는 RL에 대한 이러한 새로운 요구 사항을 충족하지 못합니다. Map Reduce [20], Apache Spark [64] 및 Dryad [28]와 같은 대량 동기식 병렬 시스템은 세분화된 시뮬레이션 또는 정책 제공을 지원하지 않습니다. CIEL [40] 및 Dask [48]와 같은 작업 병렬 시스템은 분산 교육 및 서버를 거의 지원하지 않습니다. Naiad [39] 및 Storm [31]과 같은 스트리밍 시스템에서도 마찬가지입니다. TensorFlow [7] 및 MXNet [18]과 같은 분산형 딥 러닝 프레임워크는 자연스럽게 시뮬레이션 및 서버를 지원하지 않습니다. 마지막으로 TensorFlow Serving [6] 및 Clipper [19]와 같은 모델 제공 시스템은 교육과 시뮬레이션을 모두 지원하지 않습니다.

원칙적으로는 여러 기존 시스템(예: 분산 교육용 Horovod [53], 서버용 Clipper [19], 시뮬레이션용 CIEL [40])을 결합하여 중단 간 솔루션을 개발할 수 있지만 실제로는 응용 프로그램 내에서 이러한 구성 요소의 긴밀한 결합으로 인해 접근 방식을 유지할 수 없습니다. 결과적으로 오늘날 연구자와 실무자는 전문 RL 애플리케이션을 위한 일회용 시스템을 구축합니다 [58, 41, 54, 44, 49, 5]. 이 접근 방식은 기본적으로 스케줄링, 내결함성 및 데이터 이동과 같은 표준 시스템 과제를 각 애플리케이션에 밀어 넣음으로써 분산 애플리케이션 개발에 막대한 시스템 엔지니어링 부담을 부과합니다.

본 논문에서는 RL 애플리케이션을 위한 시뮬레이션, 훈련 및 서버를 가능하게 하는 범용 클러스터 컴퓨팅 프레임워크인 Ray를 제안한다. 이러한 워크로드의 요구 사항은 시뮬레이션과 같은 경량 및 상태 비저장 컴퓨팅에서 교육과 같은 장기 실행 및 상태 저장 컴퓨팅에 이르기까지 다양합니다.

이러한 요구 사항을 충족하기 위해 Ray는 작업 병렬 및 액터 기반 계산을 모두 표현할 수 있는 통합 인터페이스를 구현합니다. 작업을 통해 Ray는 시뮬레이션을 효율적이고 동적으로 로드 밸런싱하고 대규모 입력 및 상태 공간(예: 이미지, 비디오)을 처리하고 장애로부터 복구할 수 있습니다. 대조적으로 행위자는 Ray가 모델 교육과 같은 상태 저장 계산을 효율적으로 지원하고 공유된 가변 상태를 클라이언트(예: 매개 변수 서버)에 노출할 수 있도록 합니다. Ray는 확장성이 뛰어나고 내결함성이 있는 단일 동적 실행 엔진 위에 행위자와 작업 추상화를 구현합니다.

성능 요구 사항을 충족하기 위해 Ray는 일반적으로 기존 프레임워크 [64, 28, 40]에서 중앙 집중화된 두 가지 구성 요소를 배포합니다. (1) 작업 스케줄러 및 (2) a

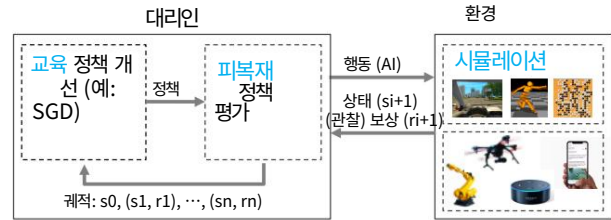


그림 1: RL 시스템의 예.

계산 계보 및 데이터 개체의 디렉토리를 유지 관리하는 메타데이터 저장소입니다. 이를 통해 Ray는 밀리초 수준의 대기 시간으로 초당 수백만 개의 작업을 예약할 수 있습니다. 또한 Ray는 작업 및 액터에 대한 계보 기반 내결함성 과 메타데이터 저장소에 대한 복제 기반 내결함성을 제공합니다.

Ray는 RL 애플리케이션 컨텍스트에서 제공, 교육 및 시뮬레이션을 지원하지만 다른 컨텍스트에서 이러한 워크로드에 대한 솔루션을 제공하는 시스템을 대체하는 것으로 간주되어야 한다는 의미는 아닙니다. 특히 Ray는 Clipper [19] 및 TensorFlow Serving [6]과 같은 서비스 시스템을 대체하는 것을 목표로 하지 않습니다. 이러한 시스템은 모델 관리, 테스트 및 모델 구성을 포함하여 모델 배포에서 더 광범위한 문제를 해결하기 때문입니다. 마찬가지로 Ray는 유연성에도 불구하고 Spark [64]와 같은 일반적인 데이터 병렬 프레임 작업을 대체할 수 없습니다. 현재 이러한 프레임워크가 제공하는 풍부한 기능과 API(예: straggler mitigation, 쿼리 최적화)가 부족하기 때문입니다.

우리는 다음과 같은 기여를 합니다.

- 새로운 RL 응용 프로그램의 필수 구성 요소인 교육, 시뮬레이션 및 서비스를 통합하는 최초의 분산 프레임 작업을 설계하고 구축합니다.
- 이러한 워크로드를 지원하기 위해 동적 작업 실행 엔진 위에 액터 및 작업 병렬 추상화를 통합합니다.
- 확장성과 내결함성을 달성하기 위해 우리는 제어 상태가 분할된 메타데이터 저장소에 저장되고 다른 모든 시스템 구성 요소가 상태 비저장인 시스템 설계 원칙을 제안합니다.
- 확장성을 확보하기 위해 상향식 디스크를 제안합니다. 공물 스케줄링 전략.

## 2 동기 및 요구 사항

RL 시스템의 기본 구성 요소를 고려 하고 Ray의 주요 요구 사항을 구체화하는 것으로 시작합니다. 그림 1에 표시된 것처럼 RL 설정에서 에이전트는 환경과 반복적으로 상호 작용합니다. 에이전트의 목표는 보상을 최대화하는 정책을 학습하는 것입니다. 정책은

```
// env 와 상호 작용하여 정책을 평가합니다 . (eg, simulator)
rollout(policy, environment): trajectory = [] state =
    environment.initial_state() while (not
        environment.has_terminated()): action =
        policy.compute(state) // Serving state, reward =
        environment.step(action) // 시뮬레이션 궤적.append(state,
            reward)
```

복귀 궤적

// 수렴 될 때 까지 정책 을 반복적 으로 개선 합니다 .

-----

```
// k 롤아웃 궤적을 생성하여 정책을 평가합니
다.append(rollout(policy, environment)) // 정책 개선
정책 = policy.update(trajectories) // 교육 반환 정책
```

그림 2: 정책 학습을 위한 일반적인 RL 유사 코드.

환경 상태에서 행동 선택으로의 매핑. 환경, 에이전트, 상태, 작업 및 보상 에 대한 정확한 정의는 애플리케이션별로 다릅니다.

정책을 학습하기 위해 에이전트는 일반적으로 (1) 정책 평가 및 (2) 정책 개선의 두 단계 프로세스를 사용합니다. 정책을 평가하기 위해 에이전트는 환경(예: 환경 시뮬레이션)과 상호 작용하여 궤적을 생성합니다. 여기서 궤적은 현재 정책에 의해 생성된 일련의 (상태, 보상) 튜플로 구성됩니다. 그런 다음 에이전트는 이러한 궤적을 사용하여 정책을 개선합니다. 즉, 보상을 최대화하는 기술기 방향으로 정책을 업데이트합니다. 그림 2는 에이전트가 정책을 배우기 위해 사용하는 의사 코드의 예를 보여줍니다. 이 의사 코드는 궤적을 생성하기 위해 롤아웃 (환경, 정책)을 호출 하여 정책을 평가합니다. train policy() 는 이러한 궤적을 사용하여 policy.update(trajectories) 를 통해 현재 정책을 개선합니다. 이 프로세스 는 정책이 수렴될 때까지 반복됩니다.

-

따라서 RL 애플리케이션을 위한 프레임워크는 교육, 제공 및 시뮬레이션을 위한 효율적인 자원을 제공해야 합니다 (그림 1). 다음으로 이러한 워크로드에 대해 간략하게 설명합니다.

교육에는 일반적으로 정책을 업데이트하기 위해 종종 분산 환경에서 확률적 경사 하강법(SGD)을 실행하는 것이 포함됩니다. 분산형 SGD는 일반적으로 allreduce 집계 단계 또는 매개변수 서버[32]에 의존합니다.

Serving은 학습된 정책을 사용하여 환경의 현재 상태를 기반으로 작업을 렌더링합니다. 서빙 시스템은 대기 시간을 최소화하고 초당 결정을 최대화하는 것을 목표로 합니다. 확장을 위해 로드는 일반적으로 정책을 제공하는 여러 노드 간에 균형을 이룹니다.

마지막으로, 대부분의 기존 RL 애플리케이션은 시뮬레이션을 사용하여 정책을 평가합니다. 현재 RL 알고리즘은 그렇지 않습니다.

물리적 세계와의 상호 작용에서 얻은 데이터에만 의존할 만큼 충분히 효율적입니다. 이러한 시뮬레이션 은 복잡도가 매우 다양합니다. 몇 ms (예: 체스 게임의 이동 시뮬레이션)에서 몇 분(예: 자율 주행 자동차의 실제 환경 시뮬레이션)이 소요될 수 있습니다.

서로 다른 시스템에서 훈련과 서빙을 별도로 처리할 수 있는 지도 학습과 달리 RL에서는 이러한 세 가지 워크로드가 모두 단일 애플리케이션에 긴밀하게 결합되어 있으며 이들 사이에는 엄격한 대기 시간 요구 사항이 있습니다. 현재 이러한 워크로드 결합을 지원하는 프레임워크는 없습니다. 이론적으로 여러 전문 프레임워크를 함께 연결하여 전체 기능을 제공할 수 있지만 실제로는 RL의 맥락에서 시스템 간의 데이터 이동 및 대기 시간으로 인한 데이터 이동이 금지됩니다. 그 결과 연구자와 실무자 들은 그들만의 일회성 시스템을 구축해 왔습니다.

이러한 상황에서는 학습, 제공 및 시뮬레이션을 효율적으로 지원할 수 있는 RL을 위한 새로운 분산 프레임워크의 개발이 필요합니다. 특히 이러한 프레임워크는 다음 요구 사항을 충족해야 합니다.

세분화된 이기종 계산. 계산 기간은 밀리초(예: 작업 수행)에서 몇 시간(예: 복잡한 정책 교육)까지 다양할 수 있습니다. 또한 교육에는 이기종 하드웨어(예: CPU, GPU 또는 TPU)가 필요한 경우가 많습니다.

유연한 계산 모델. RL 애플리케이션에는 상태 비저장 및 상태 저장 계산이 모두 필요합니다. 상태 비저장 컴퓨팅 은 시스템의 모든 노드에서 실행될 수 있으므로 필요한 경우 로드 밸런싱 및 데이터로의 컴퓨팅 이동을 쉽게 달성할 수 있습니다. 따라서 상태 비저장 컴퓨팅은 이미지나 비디오에서 기능을 추출하는 것과 같은 세분화된 시뮬레이션 및 데이터 처리에 적합합니다. 대조적으로 상태 저장 계산은 매개 변수 서버 구현, GPU 지원 데이터에 대한 반복 계산 수행 또는 상태를 노출하지 않는 타사 시뮬레이터 실행에 적합합니다.

동적 실행. RL 응용 프로그램의 여러 구성 요소는 계산이 끝나는 순서 (예: 시뮬레이션이 끝나는 순서)가 항상 미리 알려지지 않고 계산 결과가 향후 계산을 결정할 수 있기 때문에 동적 실행이 필요합니다(예: 시뮬레이션은 더 많은 시뮬레이션을 수행해야 하는지 여부를 결정합니다.)

마지막으로 두 가지 의견을 제시합니다. 첫째, 대규모 클러스터에서 높은 활용도를 달성하려면 이러한 프레임워크가 다음을 처리해야 합니다. 돌초당 수백만 개의 작업. 심층 신경 궤, 이러한 프레임워크는 망 또는 복잡한 시뮬레이터를 처음부터 구현하기 위한 것이 아닙니다. 대신 기존 시뮬레이터 [13, 11, 59] 및 딥 러닝 프레임워크[7, 18, 46, 29]와의 원활한 통합이 가능해야 합니다.

5ms 단일 코어 작업과 200개의 32코어 노드 클러스터를 가정합니다. 이 클러스터는 (1s/5ms) × 32 × 200 = 1.28M 작업/초를 실행할 수 있습니다.

|                                                        |                                                                                             |
|--------------------------------------------------------|---------------------------------------------------------------------------------------------|
| 이름                                                     | 설명 함수 f를 원                                                                                  |
| 선물 = f.remote(args)                                    | 격으로 실행합니다. f.remote()는 객체 또는 미래를 입력으로 취하고 하나 이상의 미래를 반환할 수 있습니다. 이것은 논블로킹입니다.               |
| objects = ray.get(futures)                             | 하나 이상의 미래와 관련된 값을 반환합니다. 이것은 차단입니다.                                                         |
| ready futures = ray.wait(futures, k, timeout) 해당 작업이   | 완료된 미래를 반환합니다. k가 완료되었거나 제한 시간이 만료되었습니다.                                                    |
| 액터 = Class.remote(args) 미래 = actor.method.remote(args) | Class 클래스를 원격 액터로 인스턴스화하고 이에 대한 핸들을 반환합니다. 원격 액터에서 메서드를 호출하고 하나 이상의 퓨처를 반환합니다. 둘 다 비 차단입니다. |

표 1: 레이 API

3 프로그래밍 및 계산 모델

Ray는 동적 작업 그래프 계산 모델을 구현합니다. 즉, 실행 중에 진화하는 종속 작업의 그래프로 응용 프로그램을 모델링합니다. 이 모델 위에 Ray는 액터와 작업 병렬 프로그래밍 추상화를 모두 제공합니다. 이러한 통합은 작업 병렬 추상화만 제공하는 CIEL과 같은 관련 시스템 및 주요 액터 추상화를 제공하는 Orleans [14] 또는 Akka[1]와 차별화됩니다.

3.1 프로그래밍 모델

작업. 태스크는 상태 비저장 작업자에 대한 원격 기능 실행을 나타냅니다. 원격 함수가 호출되면 작업 결과를 나타내는 future가 즉시 반환됩니다. 퓨처는 ray.get()을 사용하여 검색할 수 있으며 결과를 기다리지 않고 다른 원격 함수에 인수로 전달할 수 있습니다. 이를 통해 사용자는 데이터 종속성을 캡처하면서 병렬성을 표현할 수 있습니다.

표 1은 Ray의 API를 보여줍니다. 원격 함수는 불변 개체에서 작동하며 상태 비저장 및 부작용이 없어야 합니다. 출력은 입력에 의해서만 결정됩니다. 이는 실행 시 함수 재실행을 통해 내결함성을 단순화하는 멍등성을 의미합니다.

배우. 행위자는 상태 저장 계산을 나타냅니다. 각 액터는 원격으로 호출할 수 있고 순차적으로 실행되는 메서드를 노출합니다. 메서드 실행은 원격으로 실행되고 퓨처를 반환한다는 점에서 작업과 유사하지만 상태 저장 작업자에서 실행된다는 점에서 다릅니다. 행위자에 대한 핸들을 다른 행위자나 작업에 전달할 수 있으므로 해당 행위자에 대한 메서드를 호출할 수 있습니다.

| 작업(상태 비저장)           | 액터(상태 저장)            |
|----------------------|----------------------|
| 세분화된 로드 밸런싱          | 거친 로드 밸런싱            |
| 객체 지역성 지원            | 열악한 지역성 지원           |
| 소규모 업데이트에 대한 높은 오버헤드 | 소규모 업데이트에 대한 낮은 오버헤드 |
| 효율적인 장애 처리           | 체크포인트로 인한 오버헤드       |

표 2: 태스크 대 행위자 트레이드오프.

표 2는 작업 및 액터의 속성을 요약한 것입니다. 작업은 각 작업이 해당 입력을 저장하는 노드에서 예약될 수 있으므로 작업 세분성, 입력 데이터 지역성, 체크포인트 및 복구가 필요하지 않으므로 낮은 복구 오버헤드에서 로드 인식 스케줄링을 활용하여 세분화된 로드 밸런싱을 가능하게 합니다. 중간 상태. 대조적으로 액터는 이러한 업데이트가 일반적으로 직렬화 및 역직렬화를 필요로 하는 외부 상태가 아닌 내부에서 수행되기 때문에 훨씬 더 효율적인 세분화된 업데이트 날짜를 제공합니다. 예를 들어 액터는 매개 변수 서버 [32] 및 GPU 기반 반복 계산(예: 교육)을 구현하는 데 사용할 수 있습니다. 또한 액터를 사용하여 직렬화하기 어려운 타사 시뮬레이터 및 기타 불투명 핸들을 래핑할 수 있습니다.

이중 및 유연성에 대한 요구 사항 (섹션 2)을 충족하기 위해 세 가지 방법으로 API를 확장합니다. 첫째, 시간이 다른 동시 작업을 처리하기 위해 ray.get() 과 같은 모든 결과를 기다리는 대신 첫 번째 k개의 사용 가능한 결과를 기다리는 ray.wait() 를 도입합니다. 둘째, 리소스가 다른 작업을 처리하기 위해 개발자가 리소스 요구 사항을 지정하여 Ray 스케줄러가 리소스를 효율적으로 관리할 수 있도록 합니다. 셋째, 유연성을 향상시키기 위해 중첩된 원격 기능을 활성화합니다. 즉, 원격 기능이 다른 원격 기능을 호출할 수 있습니다. 이것은 또한 다중 프로세스가 분산 방식으로 원격 기능을 호출할 수 있도록 하므로 높은 확장성을 달성하는 데 중요합니다(섹션 4).

3.2 계산 모델

Ray는 동적 작업 그래프 계산 모델 [21]을 사용하는데, 원격 기능과 행위자 메서드의 실행은 입력이 사용 가능해지면 시스템에 의해 자동으로 트리거됩니다. 이 섹션에서는 계산 그래프(그림 4)가 사용자 프로그램(그림 3)에서 구성되는 방법을 설명합니다. 이 프로그램은 표 1의 API를 사용하여 그림 2의 의사 코드를 구현합니다.

액터를 먼저 무시하면 계산 그래프에는 데이터 개체와 원격 함수 호출 또는 작업의 두 가지 유형의 노드가 있습니다. 데이터 에지와 제어 에지 의 두 가지 유형의 에지도 있습니다. 데이터 에지는 데이터를 캡처합니다.

```

@ray.remote def
create_policy(): # 임의로 정책을
초기화합니다. 반환 정책

@ray.remote(num_gpus=1) class
시뮬레이터(객체): def __init__(self):

# 환경을 초기화합니다. self.env = 환경()

def rollout(self, policy, num_steps): 관찰 = [] 관찰 =
self.env.current_state() for

_ in range(num_steps): action
= policy(observation) 관찰 =
self.env.step(action) observes.append(observation)

반환 관찰

@ray.remote(num_gpus=2) def
update_policy(정책, *롤아웃):
# 정책을 업데이트합니다. 반환 정책

@ray.remote def
train_policy(): # 정책을 만듭니다
다. policy_id =
create_policy.remote()
# 10명의 액터를 만듭니다.
simulators = [Simulator.remote() for # 100단계의 _ 범위 내 (10)]
교육을 수행합니다. in range(100): for # 각 액터에 대해
[s.rollout.remote(policy_id) for s in simulators] rollout_ids =
train_policy.remote()

시뮬레이터의 s]
# 롤아웃으로 정책을 업데이트합니다. policy_id =
update_policy.remote(policy_id, *rollout_ids) return
ray.get(policy_id)

```

그림 3: Ray에서 그림 2의 예를 구현하는 Python 코드. @ray.remote는 원격 함수 및 액터를 나타냅니다. 원격 함수 및 행위자 메서드 호출 반환

미래는 작업 종속성을 인코딩하기 위해 후속 원격 함수 또는 행위자 메서드에 전달될 수 있습니다. 각 액터에는 모든 메서드 간에 공유되는 환경 객체 self.env가 있습니다.

데이터 개체와 작업 간의 종속성. 보다 정확하게는 데이터 개체 D가 작업 T의 출력인 경우 T에서 D로 데이터 에지를 추가합니다. 마찬가지로 D가 T에 대한 입력인 경우 D에서 T로 데이터 에지를 추가합니다. 제어 에지는 계산을 캡처합니다. 중첩된 원격 기능의 결과인 종속성 (섹션 3.1): 작업 T1이 작업 T2를 호출하면 T1에서 T2로 제어 에지를 추가합니다.

행위자 메서드 호출도 계산 그래프에서 노드로 표시됩니다. 하나의 주요 차이점을 제외하고 작업과 동일합니다. 동일한 행위자에 대한 후속 메서드 호출에서 상태 종속성을 캡처하기 위해 세 번째 유형의 에지인 상태 저장 에지를 추가합니다. Mj 메서드가 동일한 액터의 Mi 메서드 바로 다음에 호출되면 Mi에서 Mj로 상태 저장 에지를 추가합니다. 따라서 모든

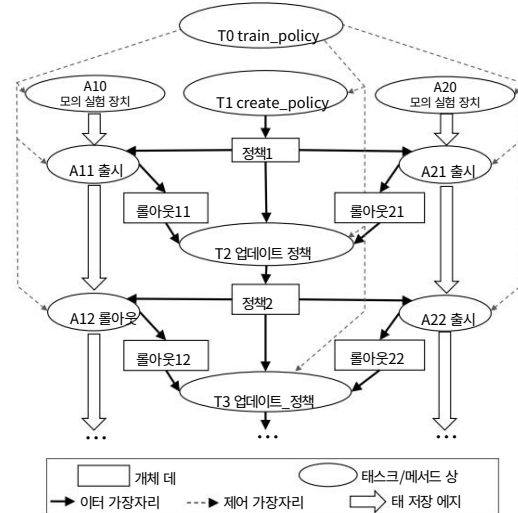


그림 4: 그림 3의 train\_policy.remote() 호출에 해당하는 작업 그래프. 원격 함수 호출 및 행위자 메서드 호출은 작업 그래프의 작업에 해당합니다. 그림은 두 배를 보여줍니다. 각 액터(A1i 및 A2i 레이블이 지정된 작업)에 대한 메서드 호출에는 변경 가능한 액터 상태를 공유함을 나타내는 상태 저장 에지가 있습니다. 열차 정책에서 호출하는 작업에 이르는 제어 에지가 있습니다. 여러 정책을 병렬로 교육하려면 train\_policy.remote()를 여러 번 호출할 수 있습니다.

동일한 액터 개체에서 호출된 메서드는 상태 저장 에지로 연결된 체인을 형성합니다 (그림 4). 이 체인은 이러한 메서드가 호출된 순서를 캡처합니다.

상태 저장 에지는 액터의 내부 상태를 공유하는 연속 메서드 호출 사이의 암시적 데이터 종속성을 캡처하기 때문에 다른 상태 비저장 작업 그래프에 액터를 포함하는 데 도움이 됩니다. 상태 저장 에지는 또한 계보를 유지할 수 있게 해줍니다. 다른 데이터 흐름 시스템 [64]에서와 마찬가지로 재구성을 가능하게 하기 위해 데이터 계보를 추적합니다. 계보 그래프에 상태 저장 에지를 명시적으로 포함함으로써 원격 함수 또는 행위자 방법(섹션 4.2.3)으로 생성된 손실된 데이터를 쉽게 재구성할 수 있습니다.

#### 4 아키텍처

Ray의 아키텍처는 (1) API를 구현하는 애플리케이션 계층과 (2) 높은 확장성과 내결함성을 제공하는 시스템 계층으로 구성됩니다.

##### 4.1 애플리케이션 계층

애플리케이션 계층은 세 가지 유형의 프로세스로 구성됩니다.

- 드라이버: 사용자 프로그램을 실행하는 프로세스.
- 작업자: 드라이버 또는 다른 사람이 호출한 작업(원격 기능)을 실행하는 상태 비저장 프로세스

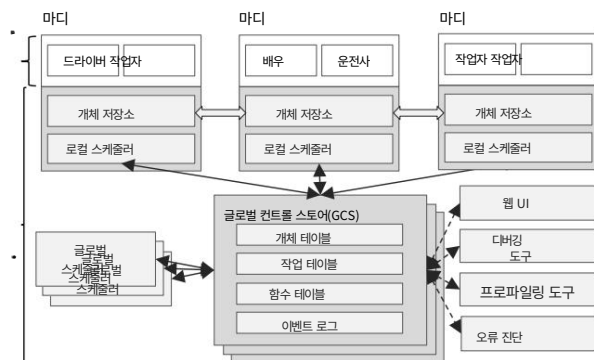


그림 5: Ray의 아키텍처는 애플리케이션 계층과 시스템 계층의 두 부분으로 구성됩니다. 애플리케이션 계층은 3장에서 설명한 API 및 계산 모델을 구현하고, 시스템 계층은 성능 및 내결함성 요구 사항을 충족하기 위해 작업 스케줄링 및 데이터 관리를 구현합니다.

노동자, 작업자는 자동으로 시작되며 시스템 계층에서 서명된 작업으로 시작됩니다. 원격 함수가 선언되면 함수가 자동으로 모든 작업자에게 게시됩니다. 작업자는 작업 간에 로컬 상태를 유지하지 않고 작업을 순차적으로 실행합니다.

- 행위자: 호출될 때 자신이 호출하는 메서드만 실행하는 상태 저장 프로세스입니다. 작업자와 달리 액터는 작업자 또는 드라이버에 의해 명시적으로 인스턴스화됩니다. 작업자와 마찬가지로 액터는 각 메서드가 이전 메서드 실행의 결과 상태에 따라 달라진다는 점을 제외하면 메서드를 순차적으로 실행합니다.

## 4.2 시스템 계층

시스템 계층은 글로벌 제어 저장소, 분산 스케줄러 및 분산 개체 저장소의 세 가지 주요 구성 요소로 구성됩니다. 모든 구성 요소는 수평 확장 가능하고 내결함성이 있습니다.

### 4.2.1 글로벌 컨트롤 스토어(GCS)

글로벌 제어 저장소(GCS)는 시스템의 전체 제어 상태를 유지하며 이는 우리 디자인의 고유한 기능입니다. 핵심적으로 GCS는 pub sub 기능이 있는 키-값 저장소입니다. 우리는 샤딩을 사용하여 확장성을 달성하고 샤딩 체인 복제 [61]를 사용하여 내결함성을 제공합니다.

GCS와 그 설계의 주된 이유는 초당 수백만 개의 작업을 동적으로 생성할 수 있는 시스템의 내결함성과 낮은 대기 시간을 유지하는 것입니다.

노드 장애 시 내결함성을 위해서는 계보 정보를 유지할 수 있는 솔루션이 필요합니다. 기존 계보 기반 솔루션 [64, 63, 40, 28]은 거친 병렬 처리에 중점을 두므로 단일 노드(예: 마스터, 드라이버)를 사용하여 성능에 영향을 주지 않고 계보를 저장할 수 있습니다. 그러나 이 설계는 시뮬레이션과 같은 세분화된 동적 워크로드에 맞게 확장할 수 없습니다. 그러므로,

내구성 있는 계보 스토리지를 다른 시스템 구성 요소와 분리하여 각각 독립적으로 확장할 수 있습니다.

짧은 대기 시간을 유지하려면 실행할 위치를 선택하고 다른 노드에서 원격 입력을 검색하는 것과 관련된 태스크 디스패치를 포함하는 태스크 스케줄링의 오버헤드를 최소화해야 합니다. 기존의 많은 데이터 흐름 시스템 [64, 40, 48]은 개체 위치와 크기를 중앙 집중식 스케줄러에 저장하여 이들을 결합합니다. 이는 스케줄러가 병목 현상이 아닐 때 자연스러운 설계입니다. 그러나 Ray가 목표로 하는 규모와 세분성은 중앙 집중식 스케줄러를 중요한 경로에서 벗어나게 유지해야 합니다. 각 개체 전송에 스케줄러를 포함하는 것은 allreduce와 같은 분산 교육에 중요한 프리미티브(통신 집약적이고 대기 시간에 민감함)에 대해 엄청난 비용이 많이 듭니다. 따라서 스케줄러가 아닌 GCS에 개체 메타데이터를 저장하여 작업 예약에서 작업 발송을 완전히 분리합니다.

요약하면 GCS는 시스템의 모든 구성 요소가 상태 비저장이 되도록 하므로 Ray의 전체 설계를 크게 단순화합니다. 이는 내결함성 지원을 단순화할 뿐만 아니라 (즉, 장애 시 구성 요소가 단순히 다시 시작하고 GCS에서 계보를 읽음) 모든 구성 요소가 다음을 통해 필요한 상태를 공유하므로 분산 개체 저장소 및 스케줄러를 종속적으로 쉽게 확장할 수 있습니다. GCS. 추가적인 이점은 디버깅, 프로파일링 및 시각화 도구를 쉽게 개발할 수 있다는 것입니다.

### 4.2.2 상향식 분산 스케줄러

섹션 2에서 설명한 것처럼 Ray는 초당 수백만 개의 작업을 동적으로 예약해야 하며 작업은 몇 밀리초 정도 걸릴 수 있습니다. 우리가 알고 있는 클러스터 스케줄러 중 어느 것도 이러한 요구 사항을 충족하지 않습니다.

Spark [64], CIEL [40] 및 Dryad [28]와 같은 대부분의 클러스터 컴퓨팅 프레임워크는 중앙 집중식 스케줄러를 구현하여 지역성을 제공할 수 있지만 대기 시간은 수십 ms입니다. 작업 도용 [12], Sparrow [45] 및 Canary [47]와 같은 분산 스케줄러는 높은 확장성을 달성할 수 있지만 데이터 지역성을 고려하지 않거나 [12] 작업이 독립적인 작업에 속한다고 가정 [45] 하거나 계산 그래프는 알려져 있습니다[47].

위의 요구 사항을 충족하기 위해 글로벌 스케줄러와 노드별 로컬 스케줄러로 구성된 2단계 계층적 스케줄러를 설계합니다. 전역 스케줄러의 과부하를 방지하기 위해 노드에서 생성된 작업은 먼저 노드의 로컬 스케줄러에 제출됩니다. 로컬 스케줄러는 노드가 과부하되지 않는 한 (즉, 로컬 작업 대기열이 사전 정의된 임계값을 초과하지 않는 한) 작업의 요구 사항을 충족할 수 없는 경우(예: GPU 부족) 작업을 로컬로 예약합니다. 로컬 스케줄러가 작업을 로컬로 예약하지 않기로 결정하면 이를 전역 스케줄러로 전달합니다. 이 스케줄러는 먼저 로컬에서(즉, 스케줄링 계층 구조의 앞) 작업 스케줄링을 시도하므로 이를 상향식 스케줄러라고 합니다.

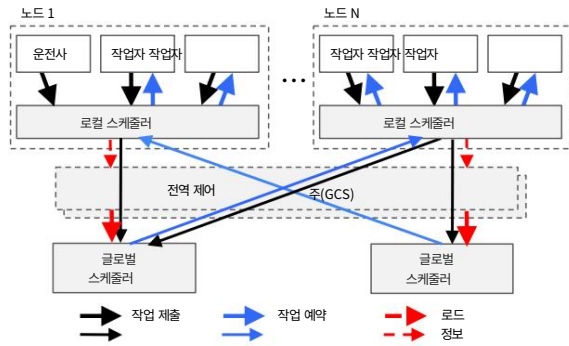


그림 6: 상향식 분산 스케줄러. 작업은 드라이버와 작업자에서 로컬 스케줄러로 상향식으로 제출되고 필요한 경우에만 전역 스케줄러로 전달됩니다(섹션 4.2.2).

각 화살표의 두께는 요청 속도에 비례합니다.

전역 스케줄러는 각 노드의 부하와 작업의 제약 조건을 고려하여 일정 결정을 내립니다. 보다 정확하게는 글로벌 스케줄러는 작업에서 요청한 유형의 리소스가 충분한 노드 집합을 식별하고 이러한 노드 중에서 가장 낮은 예상 대기 시간을 제공하는 노드를 선택합니다. 주어진 노드에서 이 시간은 (i) 작업이 해당 노드에서 대기할 예상 시간 (즉, 작업 대기열 크기 x 평균 작업 실행) 및 (ii) 작업 원격 입력의 예상 전송 시간의 합입니다. (즉, 원격 입력의 총 크기를 평균 대역폭으로 나눈 값). 전역 스케줄러는 하트비트를 통해 각 노드의 대기열 크기와 노드 리소스 가용성, GCS에서 작업 입력 위치 및 크기를 가져옵니다. 또한 글로벌 스케줄러는 단순 지수 평균을 사용하여 평균 작업 실행 및 평균 전송 대역폭을 계산합니다. 글로벌 스케줄러에 병목 현상이 발생하면 GCS를 통해 동일한 정보를 공유하는 더 많은 복제본을 인스턴스화할 수 있습니다.

이를 통해 스케줄러 아키텍처의 확장성이 매우 높아집니다.

#### 4.2.3 메모리 내 분산 개체 저장소

작업 대기 시간을 최소화하기 위해 메모리 내 분산 스토리지 시스템을 구현하여 모든 작업 또는 상태 비저장 계산의 입력 및 출력을 저장합니다. 각 노드에서 공유 메모리를 통해 객체 저장소를 구현합니다. 이를 통해 동일한 노드에서 실행 중인 작업 간에 무복사 데이터 공유가 가능합니다. 데이터 형식으로 Apache Arrow[2]를 사용합니다.

작업의 입력이 로컬이 아닌 경우 입력은 실행 전에 로컬 개체 저장소에 복제됩니다. 또한 작업은 출력을 로컬 개체 저장소에 기록합니다. 복제는 핫 데이터 개체로 인한 잠재적인 병목 현상을 제거하고 작업이 로컬 메모리에서/로 데이터를 읽고 쓰지만 하므로 작업 실행 시간을 최소화합니다. 이는 많은 AI 애플리케이션에서 공유하는 프로파일링 컴퓨팅 바인딩 워크로드의 처리량을 증가시킵니다. 짧은 대기 시간을 위해 개체를 메모리에 완전히 유지하고 다음과 같이 제거합니다.

LRU 정책을 사용하여 디스크에 필요합니다.

Spark [64] 및 Dryad [28]와 같은 기존 클러스터 컴퓨팅 프레임워크와 마찬가지로 객체 저장소는 불변 데이터로 제한됩니다. 이는 복잡한 일관성 프로토콜의 필요성을 제거하고(개체가 업데이트되지 않음) 내결함성 지원을 단순화합니다. 노드 장애의 경우 Ray는 계보 재실행을 통해 필요한 개체를 복구합니다. GCS에 저장된 계보는 초기 실행 중 상태 비저장 작업과 상태 저장 액터를 모두 추적합니다. 전자를 사용하여 저장소의 개체를 재구성합니다.

단순화를 위해 객체 저장소는 분산된 객체를 지원하지 않습니다. 즉, 각 객체는 단일 노드에 맞습니다. 큰 행렬이나 트리와 같은 분산 개체는 애플리케이션 수준에서 퓨처 컬렉션으로 구현할 수 있습니다.

#### 4.2.4 구현

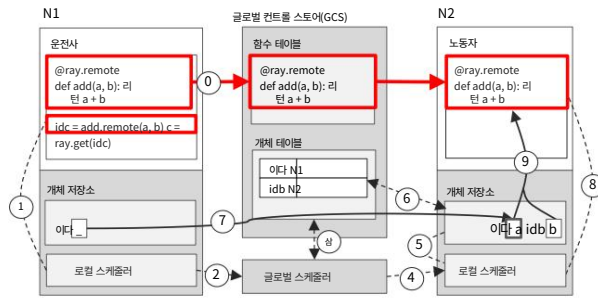
Ray는 University of California, Berkeley에서 개발된 활성 오픈 소스 프로젝트입니다. Ray는 Python 환경과 완벽하게 통합되며 간단히 `pip install ray`를 실행하여 쉽게 설치할 수 있습니다. 구현은  $\approx 40K$  코드 라인(LoC), 시스템 계층의 경우 C++에서 72%, 애플리케이션 계층의 경우 Python에서 28%로 구성됩니다. GCS는 완전히 단일 키 작업으로 샤드당 하나의 Redis [50] 키-값 저장소를 사용합니다. GCS 테이블은 확장을 위해 개체 및 작업 ID로 분할되며 모든 분할은 내결함성을 위해 체인 복제 [61]됩니다. 로컬 및 글로벌 스케줄러를 모두 이벤트 중심의 단일 스레드 프로세스로 구현합니다. 내부적으로 로컬 스케줄러는 로컬 개체 메타데이터, 입력을 기다리는 작업, 작업자에게 보낼 준비가 된 작업에 대해 캐시된 상태를 유지합니다. 서로 다른 개체 저장소 간에 큰 개체를 전송하기 위해 여러 TCP 연결에서 개체를 스트라이프합니다.

### 4.3 모든 것을 하나로 모으기

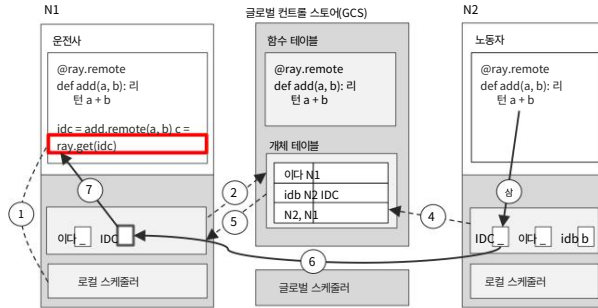
그림 7은 스칼라 또는 행렬일 수 있는 두 객체 a와 b를 추가하고 결과 c를 반환하는 간단한 예제를 통해 Ray가 종단 간 작동하는 방식을 보여줍니다. 원격 함수 `add()`는 초기화 시 자동으로 GCS에 등록되고 시스템의 모든 작업자에게 배포됩니다(그림 7a의 0단계).

그림 7a는 `add.remote(a,b)`를 호출하는 드라이버에 의해 트리거되는 단계별 작업을 보여줍니다. 여기서 a와 b는 각각 노드 N1과 N2에 저장됩니다. 드라이버는 `add(a,b)`를 로컬 스케줄러에 제출하고(1단계) 글로벌 스케줄러로 전달합니다(2단계). 다음으로 글로벌 스케줄러는 `add(a,b)`의 위치를 찾습니다. s 인수를 GCS에 저장하고(3단계) 인수 b를 저장하는 노드 N2에서 작업을 예약하기로 결정합니다(4단계). 노드 N2의 로컬 스케줄러는 로컬 개체 저장소에 `add(a,b)`의 인수가 포함되어 있는지 확인합니다(5단계). 이후

† <https://github.com/ray-project/ray> ‡ N1은 작업 일정을 로컬로 결정할 수도 있습니다.



(a) 원격으로 작업 실행



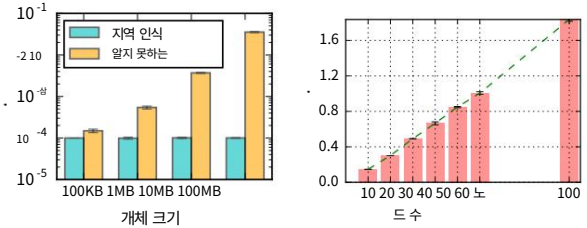
(b) 원격 작업 결과 반환

그림 7: a와 b를 더하고 c를 반환하는 종단 간 예. 실선은 데이터 플레인 작업이고 점선은 컨트롤 플레인 작업입니다. (a) 함수 add()는 노드 1 (N1)에 의해 GCS에 등록되고 N1에서 호출되고 N2에서 실행됩니다. (b) N1은 ray.get()을 사용하여 add()의 결과를 얻습니다. c에 대한 개체 테이블 항목은 4단계에서 생성되고 c가 N1에 복사된 후 6단계에서 업데이트됩니다.

지역 상점에는 개체 a가 없으므로 GCS에서 a의 위치를 찾습니다 (6단계). a가 N1에 저장되어 있음 을 알게 되면 N2의 객체 저장소는 이를 로컬로 복제합니다(7단계). 이제 add()의 모든 인수가 로컬에 저장되므로 로컬 작업자(8단계)에서 add()의 로컬 스케줄러가 공유 메모리를 통해 인수에 액세스합니다(9단계).

그림 7b는 각각 N1에서 ray.get()을 실행 하고 N2에서 add()를 실행하여 트리거 되는 단계별 작업을 보여줍니다. ray.get(idc)가 호출되면 드라이버는 add()가 반환한 미래의 idc를 사용하여 값 c에 대한 로컬 개체 저장소를 확인합니다 (1단계). 로컬 개체 저장소는 c를 저장하지 않으므로 GCS에서 해당 위치를 찾습니다. 이때 c가 아직 생성되지 않았으므로 c에 대한 항목이 없습니다. 결과적으로 N1의 객체 저장소는 c의 항목이 생성될 때 트리거될 객체 테이블에 콜백을 등록합니다(2단계). 한편 N2에서 add()는 실행을 완료하고 결과 c를 로컬 객체 저장소에 저장하고(3단계) c의 항목을 GCS에 추가합니다 (4단계). 결과적으로 GCS는 c의 항목으로 N1의 개체 저장소에 대한 콜백을 트리거합니다 (5단계). 다음으로 N1은 N2에서 c를 복제하고 (6단계) c를 ray.get()으로 반환하여 (7단계) 최종적으로 작업을 완료합니다.

이 예제에는 많은 수의 RPC가 포함되지만



(a) 레이 로컬리티 스케줄링

(b) 광선 확장성

그림 8: (a) 작업은 지역 인식 배치를 활용합니다. 임의의 개체 종속성이 있는 1000개의 작업이 두 노드 중 하나에 예약됩니다. 지역 인식 정책을 사용하면 작업 대기 시간이 1-2배 증가하는 대신 작업 입력 크기와 독립적으로 유지됩니다. (b) GCS 및 상향식 분산 스케줄러를 활용한 선행에 가까운 확장성. Ray는 60개 노드로 초당 100만 작업 처리량에 도달합니다.  $x \in \{70, 80, 90\}$  비용 때문에 생략.

대부분의 경우 이 숫자는 훨씬 작습니다. 대부분의 작업이 로컬로 예약되고 GCS 응답이 글로벌 및 로컬 스케줄러에 의해 캐시되기 때문입니다.

## 5 평가

평가에서 다음 질문을 연구합니다.

1. Ray는 섹션 2에 나열된 대기 시간, 확장성 및 연결함성 요구 사항을 얼마나 잘 충족합니까? (섹션 5.1)
2. Ray의 API를 사용하여 작성된 분산 프리미티브(예: allreduce)에 부과되는 오버헤드는 무엇입니까? (섹션 5.1)
3. RL 워크로드의 맥락에서 Ray는 훈련, 서빙 및 시뮬레이션을 위한 특수 시스템과 어떻게 비교 됩니까? (섹션 5.2)
4. 맞춤형 시스템과 비교할 때 Ray가 RL 애플리케이션에 제공하는 이점은 무엇입니까? (섹션 5.3)

모든 실험은 Amazon Web Services에서 실행되었습니다. 달리 명시되지 않는 한 우리는 m4.16xlarge CPU를 자세와 p3.16xlarge GPU 인스턴스에 사용합니다.

### 5.1 마이크로벤치마크

지역 인식 작업 배치, 세분화된 로드 밸런싱 및 지역 인식 배치는 Ray 작업의 주요 이점입니다. 일단 배치된 액터는 계산을 대형 원격 개체로 이동할 수 없지만 작업은 가능합니다.

그림 8a에서 데이터 지역성 인식 없이 배치된 작업 (액터 방법의 경우)은 10-100MB 입력 데이터 크기에서 대기 시간이 1-2배 증가합니다.

Ray는 공유 개체 저장소를 통해 작업과 액터를 통합하여 개발자가 시뮬레이션 액터가 생성한 출력에 값비싼 후처리와 같은 작업을 사용할 수 있도록 합니다.

종단간 확장성. 주요 이점 중 하나는

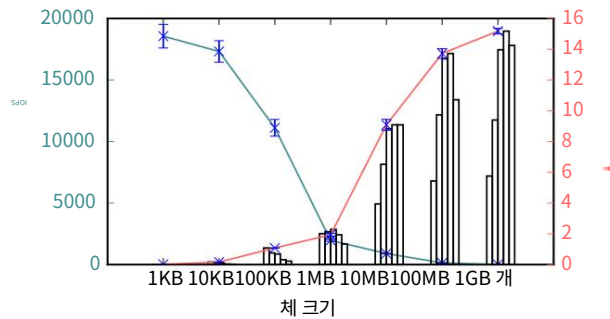


그림 9: 개체 저장소 쓰기 처리량 및 IOPS. 단일 클라이언트에서 처리량은 16코어 인스턴스(m4.4xlarge)에서 대형 객체의 경우 15GB/s(빨간색), 소형 객체의 경우 18K IOPS(청록색)를 초과합니다. 8개의 스레드를 사용하여 0.5MB보다 큰 개체를 복사하고 작은 개체의 경우 1개의 스레드를 사용합니다. 막대 그래프는 1, 2, 4, 8, 16개의 스레드가 있는 처리량을 보고합니다. 결과는 5회 실행에 대한 평균입니다.

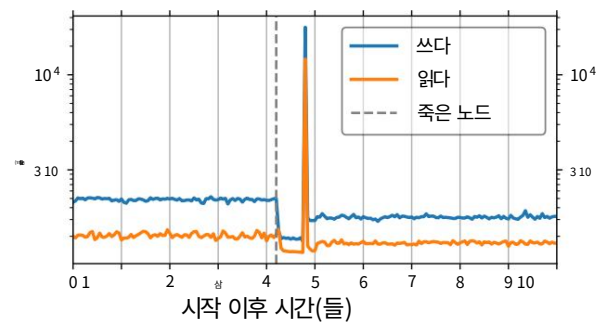
GCS(Global Control Store) 및 상향식 분산 스케줄러는 내결함성과 대기 시간이 짧은 작업 스케줄링을 유지하면서 세분화된 작업의 높은 처리량을 지원하도록 시스템을 수평으로 확장하는 기능입니다. 그림 8b에서 우리는 x축에서 클러스터 크기를 증가시키면서 비어 있는 작업의 병렬 워크로드에서 이 능력을 평가합니다. 작업 처리량이 점진적으로 증가함에 따라 거의 완벽한 선형성을 관찰합니다. Ray는 60개 노드에서 초당 100만 작업 처리량을 초과하고 100개 노드에서 초당 180만 작업 이상으로 선형 확장을 계속합니다. 가장 오른쪽 데이터 포인트는 Ray가 최소한의 가변성으로 1분(54초) 이내에 1억 개의 작업을 처리할 수 있음을 보여줍니다. 예상대로 작업 기간을 늘리면 평균 작업 기간에 비례하여 처리량이 감소하지만 전체 확장성은 선형으로 유지됩니다. 많은 현실적인 작업 부하가 개체 종속성 및 응용 프로그램 병렬 처리에 대한 본질적인 제한으로 인해 보다 제한된 확장성을 나타낼 수 있지만 이는 높은 부하에서 전체 아키텍처의 확장성을 보여줍니다.

개체 저장소 성능. 개체 저장소(4.2.3절)의 성능을 평가하기 위해 IOPS(소형 개체용) 및 쓰기 처리량(대형 개체용)의 두 가지 지표를 추적합니다. 그림 9에서 개체 크기가 증가함에 따라 단일 클라이언트의 쓰기 처리량이 15GB/s를 초과합니다.

더 큰 개체의 경우 memcpy가 개체 생성 시간을 지배합니다. 더 작은 개체의 경우 주요 오버헤드는 클라이언트와 개체 저장소 간의 직렬화 및 IPC에 있습니다.

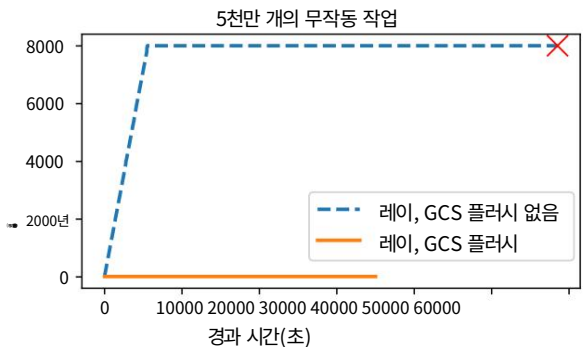
GCS 내결함성. 강력한 일관성과 내결함성을 제공 하면서 짧은 대기 시간을 유지하기 위해 Redis 위에 경량 체인 복제 [61] 계층을 구축합니다.

그림 10a는 키가 25바이트이고 값이 512바이트인 GCS에서 Ray 작업을 기록하고 GCS에서 작업을 읽는 것을 시뮬레이션합니다. 클라이언트는 한 번에 최대 하나의 진행 중인 요청을 포함하여 최대한 빨리 요청을 보냅니다. 실패는 클라이언트(명백한 오류를 수신했거나 재시도에도 불구하고 시간 초과됨)에서 체인 마스터로 보고되거나



(a) 작업을 제출하는 클라이언트에서 본 GCS 읽기 및 쓰기 대기 시간에 대한 타임라인. 체인은 2개의 복제본으로 시작합니다. 다음과 같이 재구성을 수동으로 트리거합니다.  $t \approx 4.2s$ 에서 체인 멤버가 죽습니다. 그 직후 새 체인 구성원이 합류하고 상태 전송을 시작하며 체인을 양방향 복제로 복원합니다.

재구성에도 불구하고 최대 클라이언트 관찰 대기 시간은 30ms 미만입니다.



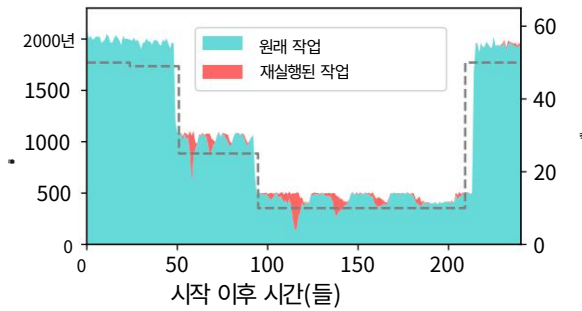
(b) Ray GCS는 GCS 플러시로 일정한 메모리 공간을 유지합니다. GCS 플러시가 없으면 메모리 공간이 최대 용량에 도달하고 워크로드가 미리 결정된 기간(빨간색 십자가로 표시됨) 내에 완료되지 않습니다.

그림 10: Ray GCS 내결함성 및 플러싱.

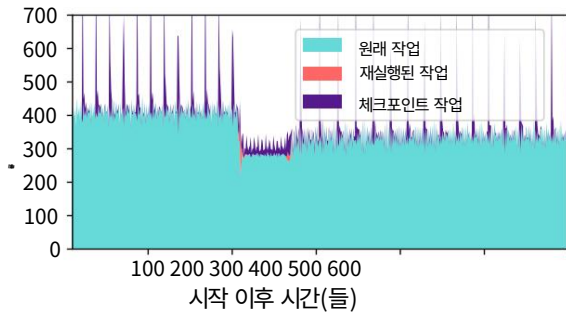
체인의 모든 서버에서(명시적 오류 수신). 전반적으로 재구성으로 인해 최대 30ms 미만의 클라이언트 관찰 지연이 발생했습니다(여기에는 오류 감지 및 복구 지연이 모두 포함됨).

GCS 플러싱. Ray는 주기적으로 GCS의 내용을 디스크로 플러싱할 수 있습니다. 그림 10b에서 우리는 5천만 개의 빈 작업을 순차적으로 제출하고 GCS 메모리 소비를 모니터링합니다. 예상대로 추적되는 작업의 수에 따라 선형적으로 증가하고 결국 시스템의 메모리 용량에 도달합니다. 이 시점에서 시스템이 중단되고 워크로드가 합리적인 시간 내에 완료되지 않습니다. 주기적인 GCS 플러싱을 통해 두 가지 목표를 달성합니다. 첫째, 메모리 풋프린트는 사용자가 구성할 수 있는 수준으로 제한됩니다(마이크로벤치마크에서는 소비된 메모리를 가능한 한 낮게 유지하는 공격적인 전략을 사용합니다). 둘째, 플러싱 메커니즘은 장기 실행 Ray 애플리케이션을 위해 디스크에 계보를 스냅샷하는 자연스러운 방법을 제공합니다.

작업 실패에서 복구. 그림 11a에서 우리는



(a) 작업 재구성



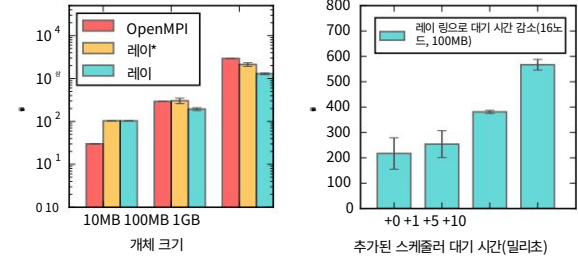
(b) 배우 재구성

그림 11: 레이 내결함성. (a) Ray는 노드가 제거될 때(점선) 손실된 작업 종속성을 재구성하고 노드가 다시 추가되면 원래 처리량으로 복구합니다. 각 작업은 100ms이며 이전에 제출된 작업에서 생성된 개체에 따라 다릅니다. (b) 액터는 마지막 체크포인트에서 재구성됩니다.  $t = 200s$  에서 10개 노드 중 2개를 종료하여 클러스터의 2000개 액터 중 400개를 나머지 노드에서 복구합니다( $t = 200-270s$ ).

내구성 있는 GCS 계보 스토리지를 사용하여 작업자 노드 장애로부터 투명하게 복구하고 탄력적으로 확장하는 Ray의 능력을 보여줍니다. m4.xlarge 인스턴스에서 실행되는 워크로드는 드라이버가 제출한 100ms 작업의 선형 체인으로 구성됩니다. 노드가 제거되면(25초, 50초, 100초) 로컬 스케줄러는 실행을 계속하기 위해 체인의 이전 결과를 재구성합니다. 모든 노드당 처리량은 전반적으로 안정적으로 유지됩니다.

액터 실패에서 복구. 행위자 메서드 호출을 종속성 그래프에서 직접 상태 저장 예지로 인코딩하면 그림 11a와 동일한 개체 재구성 메커니즘을 재사용하여 상태 저장 계산을 위한 투명한 내결함성을 제공할 수 있습니다. Ray는 사용자 정의 체크포인트 기능을 추가로 활용하여 액터의 재구성 시간을 제한합니다(그림 11b). 최소한의 오버헤드로 체크포인트를 사용하면 500개의 메서드만 재실행할 수 있는 반면 체크포인트 없이는 10,000개의 재실행이 가능합니다. 앞으로는 예를 들어 사용자가 상태를 변경하지 않는 메서드에 주석을 달 수 있도록 하여 액터 재구성 시간을 더욱 단축할 수 있기를 바랍니다.

올리두스. Allreduce는 분산 커뮤니케이션입니다.



(a) 레이 대 OpenMPI

(b) 레이 스케줄러 제거

그림 12: (a) 16개의 m4.16xlarge 노드에서 allreduce의 평균 실행 시간. 각 작업자는 별개의 노드에서 실행됩니다. Ray\*는 Ray를 전송용 스트레드 1개와 수신용 스트레드 1개로 제한합니다. (b) Ray의 짧은 대기 시간 스케줄링은 allreduce에 매우 중요합니다.

많은 기계 학습 워크로드에 중요한 기본 요소입니다.

여기에서 Ray가 기본적으로 기존 구현[53]과 일치하도록 충분히 낮은 오버헤드로 링 올리두스 [57] 구현을 지원할 수 있는지 평가합니다. 우리는 Ray가 ~200ms에 100MB, ~1200ms에 1GB에서 16개 노드에 걸쳐 allreduce를 완료하여 인기 있는 MPI 구현인 OpenMPI(v1.10)보다 각각 1.5배 및 2배 뛰어난 성능을 보인다는 것을 발견했습니다(그림 12a). Ray의 성능은 AWS의 노드 간 25Gbps 연결을 최대한 활용하여 네트워크 전송에 여러 스트레드를 사용하는 반면 OpenMPI는 단일 스트레드에서 데이터를 순차적으로 송수신합니다 [22].

더 작은 개체의 경우 OpenMPI는 더 낮은 오버헤드 알고리즘으로 전환하여 Ray를 능가합니다. 이 최적화는 향후 구현할 예정입니다.

Ray의 스케줄러 성능은 allreduce와 같은 `ing` 프리미티브를 구현하는 데 중요합니다. 그림 12b에서는 인위적인 작업 실행 지연을 삽입하고 몇 ms의 추가 대기 시간으로 성능이 거의 2배 떨어짐을 보여줍니다. Spark 및 CIEL과 같은 중앙 집중식 스케줄러가 있는 시스템은 일반적으로 수십 밀리 초의 스케줄러 오버헤드가 있어 [62, 38] 이러한 작업 부하를 비실용적으로 만듭니다. 링에 필요한 작업 수가 참여자 수에 따라 2차적으로 감소하기 때문에 일정 처리량도 병목 현상이 됩니다.

## 5.2 빌딩 블록

End-to-end 애플리케이션(예: AlphaGo [54])은 교육, 서빙 및 사물레이아웃의 긴밀한 결합이 필요합니다. 이 섹션에서는 이러한 워크로드 각각을 일반적인 RL 애플리케이션의 요구 사항을 설명하는 설정으로 격리합니다.

RL을 대상으로 하는 유연한 프로그래밍 모델과 이 프로그래밍 모델을 지원하도록 설계된 시스템으로 인해 Ray는 이러한 개별 워크로드에 대한 전용 시스템의 성능과 일치하거나 때로는 초과합니다.

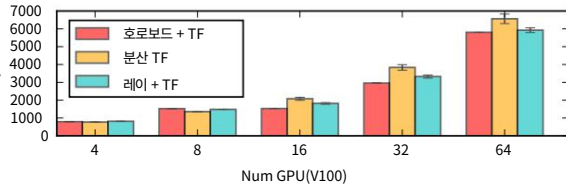


그림 13: ResNet-101 TensorFlow 모델의 훈련을 배포할 때 도달한 초당 이미지(공식 TF 벤치마크에서). 모든 실험은 25Gbps 이더넷으로 연결된 p3.16xlarge 인스턴스에서 실행되었으며 작업자는 Horovod [53]에서와 같이 노드당 4개의 GPU를 할당했습니다. 하드웨어 차이 및 최근 TensorFlow 성능 개선으로 인해 이전에 보고된 것과 일부 측정 편차가 있음을 확인합니다.

모든 실행에 OpenMPI 3.0, TF 1.8 및 NCCL2를 사용했습니다.

### 5.2.1 분산 훈련

Ray 액터 추상화를 활용하여 모델 복제본을 나타내는 데이터 병렬 동시식 SGD를 구현합니다.

모델 가중치는 모두 Ray API 위에 구현된 allreduce(5.1) 또는 매개변수 서버를 통해 동기화됩니다.

그림 13에서 우리는 각 실험에 대해 동일한 TensorFlow 모델과 합성 데이터 생성기를 사용하여 최첨단 구현 [53]에 대해 Ray(동시식) 매개변수 서버 SGD 구현의 성능을 평가합니다. TensorFlow 기반 시스템 과만 비교하여 딥 러닝 프레임워크 자체의 차이점이 아닌 Ray가 부과하는 오버헤드를 정확하게 측정합니다. 각 반복에서 모델 복제 액터는 그라디언트를 병렬로 계산하고 분할된 매개변수 서버로 그라디언트를 보낸 다음 다음 반복을 위해 매개변수 서버에서 합산된 그라디언트를 읽습니다.

그림 13은 Ray가 Horovod의 성능과 일치하고 분산된 TensorFlow의 10% 이내임을 보여줍니다(분산 복제 모드에서). 이는 Ray의 범용 API에서 이러한 특수 시스템에서 발견되는 것과 동일한 애플리케이션 수준 최적화를 표현할 수 있는 기능 때문입니다. 핵심 최적화는 단일 반복 내에서 기울기 계산, 전송 및 합산의 파이프라이닝입니다. 네트워크 전송과 GPU 계산을 겹치기 위해 맞춤형 TensorFlow 연산자를 사용하여 Tensor를 Ray의 객체 저장소에 직접 씁니다.

### 5.2.2 서빙

모델 제공은 중단 간 애플리케이션의 중요한 구성 요소입니다. Ray는 주로 동일한 동적 작업 그래프 내에서(예: Ray의 RL 애플리케이션 내에서) 실행되는 시뮬레이터에 모델을 제공하는 데 중점을 둡니다. 대조적으로 Clipper [19]와 같은 시스템은 외부 클라이언트에게 예측을 제공하는 데 중점을 둡니다.

이 설정에서 낮은 대기 시간은 높은 활용도를 달성하는 데 중요합니다. 이를 보여주기 위해 표 3에서

|                                                |               |  |
|------------------------------------------------|---------------|--|
| 시스템 소형 입력 대형 입력 Clipper 4400 ± 15states/sec    |               |  |
| 290 ± 1.3states/sec 6200 ± 21states/sec 6900 ± |               |  |
| 레이                                             | 150states/sec |  |

표 3: 전용 서빙 시스템인 Clipper [19]와 내장형 서빙 워크로드 2개에 대한 Ray의 처리량 비교.

잔여 네트워크와 완전히 연결된 작은 네트워크를 사용하여 평가하는 데 각각 10ms와 5ms가 걸립니다. 서버는 각각 4KB 및 100KB 크기의 상태를 64개의 배치로 보내는 클라이언트에 의해 쿼리됩니다.

Ray 액터를 사용하여 정책을 제공하는 것과 REST를 통해 오픈 소스 Clipper 시스템을 사용하여 달성한 서버 처리량. 여기에서 클라이언트 및 서버 프로세스는 모두 동일한 시스템(p3.8xlarge 인스턴스)에 함께 있습니다. 이것은 종종 RL 애플리케이션의 경우이지만 Clipper와 같은 시스템에서 처리하는 일반적인 웹 서비스 워크로드에는 해당되지 않습니다.

낮은 오버헤드 직렬화 및 공유 메모리 추상화로 인해 Ray는 큰 입력을 받아들이고 사용된 것과 유사한 더 비싼 잔여 네트워크 정책 모델에서 더 빠른 작은 완전히 연결된 정책 모델에 대해 훨씬 더 높은 처리량을 달성합니다. AlphaGo Zero에서는 입력이 더 적습니다.

### 5.2.3 시뮬레이션

RL에서 사용되는 시뮬레이터는 가변 길이("타임스텝")로 결과를 생성하는데, 훈련이 있는 빠박한 루프로 인해 사용 가능한 즉시 사용해야 합니다. 작업 이질성과 적시성 요구 사항으로 인해 BSP 스타일 시스템에서 시뮬레이션을 효율적으로 지원하기 어렵습니다.

시연을 위해 우리는 (1) 라운드 사이에 글로벌 장벽이 있는 3라운드 n 코어에서 3n 병렬 시뮬레이션 실행을 제출하는 MPI 구현을 (2) 시뮬레이션 결과를 동시에 수집 하면서 동일한 3n 작업을 실행하는 Ray 프로그래밍 모델을 사용합니다. 드라이버로 돌아갑니다. 표 4는 두 시스템 모두 확장성이 우수하지만 Ray는 최대 1.8배의 처리량을 달성한다는 것을 보여줍니다. 이는 세분화된 시뮬레이션 작업의 결과를 동적으로 생성하고 수집할 수 있는 프로그래밍 모델에 동기를 부여합니다.

|               |       |        |         |       |
|---------------|-------|--------|---------|-------|
| 시스템, 프로그래밍 모델 | 1 CPU | 16 CPU | 256 CPU |       |
| MPI, 대량 동기식   |       | 22.6K  | 208K    | 2.16M |
| Ray, 비동기 작업   |       | 22.3K  | 29만     | 4.03M |

표 4: OpenAI Gym [13]의 Pendulum-v0 시뮬레이터에 대한 초당 시간 단계. Ray는 대규모로 이기종 시뮬레이션을 실행할 때 더 나은 활용을 허용합니다.

<sup>5</sup>전문가는 장벽을 극복하기 위해 MPI의 비동기 프리미티브를 사용할 수 있지만 프로그램 복잡성이 증가함에도 불구하고 BSP를 시뮬레이터하기 위해 그러한 구현을 선택했습니다.

### 5.3 RL 애플리케이션

교육, 시뮬레이션 및 서빙 단계를 밀접하게 결합할 수 있는 시스템이 없으면 오늘날의 강화 학습 알고리즘은 일회성 솔루션으로 구현되어 예를 들어 다른 계산 구조가 필요하거나 다른 아키텍처. 결과적으로 Ray에서 두 가지 대표적인 강화 학습 응용 프로그램을 구현하여 이러한 알고리즘을 위해 특별히 구축된 사용자 지정 시스템을 일치시키고 심지어 수행할 수 있습니다. 주된 이유는 Ray의 프로그래밍 모델의 유연성 때문입니다. 이 모델은 맞춤형 시스템으로 포팅하기 위해 상당한 엔지니어링 노력이 필요한 응용 프로그램 수준의 최적화를 표현할 수 있지만 Ray의 동적 작업 그래프 실행 엔진에서 투명하게 지원됩니다.

#### 5.3.1 진화 전략

대규모 RL 워크로드에서 Ray를 평가하기 위해 진화 전략(ES) 알고리즘을 구현하고 참조 구현 [49]과 비교합니다. 데이터 공유를 위한 다중 처리 라이브러리. 이 알고리즘은 주기적으로 작업자 풀에 새 정책을 브로드캐스트하고 대략 10,000개의 작업 결과를 집계합니다 (각각 10~1,000개의 시뮬레이션 단계 수행).

그림 14a에서 볼 수 있듯이 Ray의 구현은 8192개의 코어로 확장됩니다. 사용 가능한 코어를 두 배로 늘리면 평균 완료 시간이 1.6배 빨라집니다. 반대로 특수 목적 시스템은 2048코어에서 완료되지 않으며 시스템 작업이 애플리케이션 드라이버의 처리 용량을 초과합니다. 이 문제를 방지하기 위해 Ray 구현은 액터의 집계 트리를 사용하여 최고 게시 결과(10분)보다 2배 이상 빠른 3.7분의 중간 시간에 도달합니다.

Ray를 사용한 직렬 구현의 초기 병렬화는 7줄의 코드만 수정하면 됩니다. 중첩 작업 및 액터에 대한 Ray의 지원으로 계층적 집계를 통한 성능 향상을 쉽게 실현할 수 있었습니다.

대조적으로, 참조 구현에는 작업자 간의 작업 및 데이터 통신을 위한 프로토콜 전용으로 수백 줄의 코드가 있었고 계층적 집계와 같은 최적화를 지원하려면 추가 엔지니어링이 필요했습니다.

#### 5.3.2 근접 정책 최적화

우리는 Ray에서 PPO(Proximal Policy Optimization) [51]를 구현하고 OpenMPI 통신 프리미티브를 사용하는 고도로 최적화된 참조 구현 [5]과 비교합니다. 알고리즘은 비용기식 분산-수집 방식으로 시뮬레이션 액터에 새 작업이 할당됩니다.

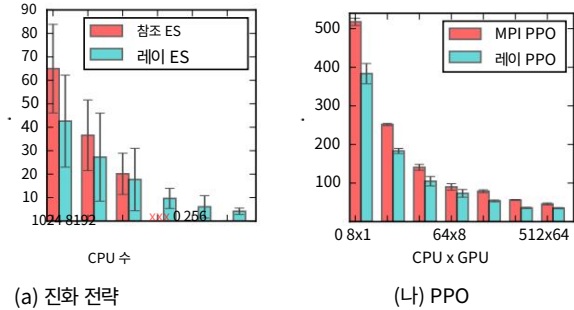


그림 14: Humanoid v1 작업에서 6000점에 도달하는 데 걸리는 시간 [13]. (a) Ray ES 구현은 8192개의 코어로 잘 확장되며 최고 게시 결과보다 2배 이상 빠른 3.7분의 평균 시간을 달성합니다. 특수 목적 시스템은 1024개 코어 이상으로 실행되지 못했습니다. ES는 이 벤치마크에서 PPO보다 빠르지 만 런타임 편차가 더 큼니다. (비)

Ray PPO 구현은 훨씬 적은 비용으로 더 적은 수의 GPU로 특수 MPI 구현 [5]을 능가합니다.

MPI 구현에는 CPU 8개마다 GPU 1개가 필요했지만 Ray 버전은 최대 8개의 GPU가 필요했습니다(8개 CPU당 GPU 1개 이하).

롤아웃을 드라이버에 반환합니다. 작업은 320000개의 시뮬레이션 단계가 수집될 때까지 제출됩니다(각 작업은 10~1000단계 생성). 정책 업데이트는 배치 크기가 32768인 SGD의 20단계를 수행합니다. 이 예의 모델 매개변수는 대략 350KB입니다.

이러한 실험은 p2.16xlarge(GPU) 및 m4.16xlarge(높은 CPU) 인스턴스를 사용하여 실행되었습니다.

그림 14b에서 볼 수 있듯이 Ray 구현은 GPU의 일부를 사용하면 모든 실험에서 최적화된 MPI 구현을 수행합니다. 그 이유는 Ray가 이질성을 인식하고 사용자가 태스크 또는 액터의 세분성에서 리소스 요구 사항을 표현하여 비대칭 아키텍처를 활용할 수 있도록 하기 때문입니다. 그런 다음 Ray 구현은 TensorFlow의 단일 프로세스 다중 GPU 지원을 활용하고 가능한 경우 개체를 GPU 메모리에 고정할 수 있습니다. 이 최적화는 단일 GPU 프로세스에 대한 롤아웃을 비용기식으로 수집해야 하기 때문에 MPI로 쉽게 포팅할 수 없습니다. 실제로 [5]에는 PPO의 두 가지 사용자 지정 구현이 포함되어 있습니다. 하나는 대규모 클러스터에 MPI를 사용하고 다른 하나는 GPU에 최적화되었지만 단일 노드로 제한됩니다. Ray는 두 시나리오 모두에 적합한 구현을 허용합니다.

리소스 이질성을 처리하는 Ray의 능력은 또한 PPO의 비용을 4.5배 감소시켰습니다 [4]. CPU 전용 작업은 더 저렴한 고성능 CPU 인스턴스에서 예약할 수 있기 때문입니다.

대조적으로, MPI 애플리케이션은 종종 모든 프로세스가 동일한 코드를 실행하고 동일한 리소스를 필요로 하는 대칭 아키텍처를 나타내므로 확장을 위해 CPU 전용 시스템을 사용할 수 없습니다. 또한 MPI 구현에는 장애를 투명하게 처리하지 않기 때문에 주문형 인스턴스가 필요합니다. 4배 저렴한 스팟 인스턴스를 가정하면 Ray의 내결함성 및 리소스 인식 스케줄링을 함께 사용하면 비용이 18배 절감됩니다.

## 6 관련 작업

동적 작업 그래프. Ray는 CIEL [40] 및 Dask [48]와 밀접한 관련이 있습니다. 세 가지 모두 중첩 작업이 있는 동적 작업 그래프를 지원 하고 미래 추상화를 구현합니다. CIEL은 또한 계보 기반 내결함성을 제공 하는 반면 Dask는 Ray와 마찬가지로 Python과 완전히 통합됩니다.

그러나 Ray는 성능에 중요한 영향을 미치는 두 가지 측면에서 다릅니다. 먼저 Ray는 행위자 추상화로 작업 모델을 확장합니다. 이는 모델 데이터를 계산과 함께 배치하기 위해 분산 교육 및 서버에서 효율적인 상태 저장 계산에 필요합니다. 둘째, Ray는 모든 메타데이터를 저장하는 단일 마스터에 의존하는 대신 완전히 분산되고 분리된 제어 평면과 스케줄러를 사용합니다. 이는 시스템 수정 없이 allreduce와 같은 프리미티브를 효율적으로 지원하는 데 중요합니다. 16개 노드에서 100MB의 최고 성능에서 Ray(섹션 5.1)의 allreduce는 200ms에 16개 작업의 32개 라운드를 제출합니다. 한편, Dask는 512개의 코어에서 3k 작업/초의 최대 스케줄러 처리량을 보고합니다 [3].

중앙 집중식 스케줄러를 사용하면 allreduce의 각 라운드에서 최소 ~5ms의 스케줄링 지연이 발생하여 완료 시간이 최대 2배 더 나빠집니다(그림 12b). 분산형 스케줄러를 사용하더라도 컨트롤 플레인 정보를 스케줄러와 연결하면 스케줄러가 데이터 전송을 위한 중요한 경로에 남아 모든 축소 라운드마다 추가 왕복이 추가됩니다.

데이터 흐름 시스템. MapReduce [20], Spark [65] 및 Dryad [28]와 같은 인기 있는 데이터 흐름 시스템은 분석 및 ML 워크로드에 널리 채택되었지만 해당 계산 모델은 세분화된 동적 시뮬레이션 워크로드에 대해 너무 제한적입니다. Spark 및 MapReduce는 BSP 실행 모델을 구현하며, 동일한 단계 내의 작업이 동일한 계산을 수행하고 대략 동일한 시간이 소요된다고 가정합니다. Dryad는 이 제한을 완화하지만 동적 작업 그래프에 대한 지원이 부족합니다. 또한 이러한 시스템 중 어느 것도 액터 추상화를 제공하지 않으며 확장 가능한 분산 제어 평면 및 스케줄러를 구현하지도 않습니다. 마지막으로 Naiad [39]는 일부 워크로드에 대해 향상된 확장성을 제공하지만 정적 작업 그래프만 지원하는 데이터 흐름 시스템입니다.

기계 학습 프레임워크. TensorFlow [7] 및 MXNet [18]은 딥 러닝 워크로드를 대상으로 하며 CPU와 GPU를 모두 효율적으로 활용합니다. 선형 대수 연산의 정적 DAG로 구성된 교육 워크로드에 대해 뛰어난 성능을 달성 하지만 교육을 시뮬레이션 및 임베디드 서버와 긴밀하게 결합하는 데 필요한 보다 일반적인 계산에 대한 지원은 제한적입니다. TensorFlow Fold [33]는 내부 C++ API를 통해 MXNet뿐만 아니라 동적 작업 그래프에 대한 일부 지원을 제공 하지만 작업 진행, 작업 완료 시간 또는 결함에 대한 응답으로 실행 중에 DAG를 수정하는 기능을 완전히 지원하지 않습니다. TensorFlow 및 MXNet은 원칙적으로 프로그램을 허용함으로써 일반성을 달성합니다.

mer는 낮은 수준의 메시지 전달 및 동기화 프리미티브를 시뮬레이트하지만 이 경우 합성과 사용자 경험은 MPI와 유사합니다. OpenMPI [22]는 높은 성능을 달성 할 수 있지만 이기종 및 동적 작업 그래프를 처리하기 위해 명시적인 조정이 필요하기 때문에 프로그래밍하기가 상대적으로 어렵습니다. 또한 프로그래머가 내결함성을 명시적으로 처리하도록 합니다.

액터 시스템. Orleans [14]와 Akka [1]는 가용성이 높고 분산 시스템을 개발하는 데 매우 적합한 두 액터 프레임워크입니다. 그러나 Ray에 비해 데이터 손실 복구 지원이 적습니다. 상태 저장 액터를 복구하려면 Orleans 개발자는 액터 상태 및 중간 응답을 명시적으로 검사해야 합니다. Orleans의 상태 비저장 액터는 확장을 위해 복제할 수 있으므로 작업으로 작동할 수 있지만 Ray와 달리 계보가 없습니다. 유사하게, Akka는 실패에 걸려 지속되는 액터 상태를 명시적으로 지원하지 않지만 상태 비저장 컴퓨팅(예: 작업)에 대한 효율적인 내결함성을 제공하지 않습니다. 메시지 전달의 경우 Orleans는 at-least-once를 제공하고 Akka는 at-most-once 시맨틱을 제공합니다.

대조적으로, Ray는 각 메시지 호출이 GCS에 기록되고 인수와 결과가 모두 불변이므로 투명한 내결함성과 정확히 한 번 의미 체계를 제공합니다. 실제로 이러한 제한 사항은 애플리케이션의 성능에 영향을 미치지 않습니다. 다른 액터 기반 시스템인 Erlang [10]과 C++ 액터 프레임워크 [17]는 마찬가지로 내결함성을 지원하는 데 제한이 있습니다.

글로벌 제어 저장소 및 스케줄링. 컨트롤 플레인을 논리적으로 중앙 집중화한다는 개념은 소프트웨어 정의 네트워크(SDN) [16], 분산 파일 시스템(예: GFS [23]), 리소스 관리(예: Omega [52]) 및 분산 프레임에서 이전에 제안되었습니다. 작동합니다(예: MapReduce [20], BOOM [9]). Ray는 이러한 선구적인 노력에서 영감을 얻었지만 상당한 개선을 제공합니다. SDN, BOOM 및 GFS와 달리 Ray는 논리 구현(예: 스케줄러)에서 컨트롤 플레인 정보(예: GCS)의 저장을 분리합니다. 이를 통해 스토리지 및 계산 계층 모두 독립적으로 확장할 수 있으며 이는 확장성 목표를 달성하는 데 핵심입니다. Omega는 스케줄러가 전역 공유 상태를 통해 조정하는 분산 아키텍처를 사용합니다. 이 아키텍처에 Ray는 전역 스케줄러를 추가하여 로컬 스케줄러 간에 부하를 분산하고 두 번째 수준이 아닌 ms 수준의 작업 스케줄링을 대상으로 합니다.

Ray는 수평 확장이 가능하고 동적으로 구성된 작업 그래프를 처리할 수 있는 고유한 분산형 상향식 스케줄러를 구현합니다. Ray와 달리 대부분의 기존 클러스터 컴퓨팅 시스템 [20, 64, 40]은 중앙 집중식 스케줄러 아키텍처를 사용합니다. Sparrow [45]는 분산 형이지만 스케줄러는 독립적인 결정을 내리고 가능한 스케줄링 정책을 제한하며 작업의 모든 작업은 동일한 글로벌 스케줄러에 의해 처리됩니다. Mesos [26]는 2단계 계층 스케줄러를 구현하지만 최상위 스케줄러는 개별 작업이 아닌 프레임워크를 관리합니다.

Canary [47]는 각 스케줄러 인스턴스가 작업 그래프의 일부를 처리하도록 함으로써 인상적인 성능을 달성 하지만 동적 계산 그래프는 처리하지 않습니다.

Cilk [12]는 작업 휴지기 스케줄러가 동적 작업 그래프를 위한 효율적인 로드 밸런싱을 달성하는 병렬 프로그래밍 언어입니다. 그러나 Ray의 글로벌 스케줄러와 같은 중앙 조정자가 없기 때문에 이 완전 병렬 설계는 분산 환경에서 데이터 지역성과 리소스 이질성을 지원하도록 확장하기 어렵습니다.

## 7 토론 및 경험

Ray를 구축하는 것은 긴 여정이었습니다. 분산 교육 및 시뮬레이션을 수행하기 위해 2년 전에 Spark 라이브러리로 시작했습니다. 그러나 BSP 모델의 상대적 경직성, 작업당 높은 오버헤드, 행위자 추상화의 부족으로 인해 우리는 새로운 시스템을 개발하게 되었습니다. 약 1년 전에 Ray를 출시한 이후 수백 명의 사용자가 Ray를 사용했고 여러 회사에서 프로덕션 환경에서 실행하고 있습니다. 여기에서 Ray를 개발하고 사용한 경험과 일부 초기 사용자 피드백에 대해 논의합니다.

API. API를 설계하면서 미니멀리즘을 강조했습니다. 처음에는 기본 작업 추상화로 시작했습니다.

나중에 우리는 이기종 기간이 있는 돌아옴을 수용하기 위해 wait() 프리미티브를 추가했고 타사 시뮬레이터를 수용하고 비용이 많이 드는 초기화의 오버헤드를 상각하기 위해 액터 추상화를 추가했습니다. 결과 API는 상대적으로 낮은 수준이지만 강력하고 사용하기 쉬운 것으로 입증되었습니다. 우리는 이미 이 API를 사용하여 A3C [36], PPO [51], DQN [37], ES [49], DDPG [55] 및 유인원 X [27]. 대부분의 경우 이러한 알고리즘을 Ray로 포팅하는 데 수십 줄의 코드만 필요했습니다. 초기 사용자 피드백을 기반으로 일정 결정을 알릴 수 있는 더 높은 수준의 프리미티브 및 라이브러리를 포함하도록 API를 개선하는 것을 고려하고 있습니다.

제한 사항. 워크로드 일반성을 감안할 때 특수화 된 최적화는 어렵습니다. 예를 들어 계산 그래프에 대한 완전한 지식 없이 일정 결정을 내려야 합니다. Ray의 일정 최적화에는 더 복잡한 런타임 프로파일링이 필요할 수 있습니다. 또한 각 작업의 계보를 저장하려면 GCS에서 저장 비용을 제한하는 가비지 수집 정책을 구현해야 하며, 이는 우리가 적극적으로 개발하고 있는 기능입니다.

결함 허용. AI 애플리케이션에 내결함성이 정말로 필요한지 묻는 질문을 자주 받습니다. 결국 많은 AI 알고리즘의 통계적 특성으로 인해 실패한 돌아옴을 간단히 무시할 수 있습니다. 우리의 경험에 비추어 볼 때 우리의 대답은 "그렇다"입니다. 첫째, 오류를 무시하는 기능은 응용 프로그램을 훨씬 쉽게 작성하고 추론할 수 있도록 합니다. 둘째, 결정론적 재생을 통한 내결함성의 특정 구현은 대부분의 오류를 쉽게 재현할 수 있으므로 디버깅을 크게 단순화합니다. 이는 확률론으로 인해 AI al

고리들은 디버깅하기 어려운 것으로 악명이 높습니다. 셋째, 내결함성은 AWS의 스팟 인스턴스와 같은 저렴한 리소스에서 실행할 수 있으므로 비용을 절약하는 데 도움이 됩니다. 물론 이것은 약간의 간접비를 수반합니다. 그러나 우리는 이 오버헤드가 대상 워크로드에 대해 최소임을 확인했습니다.

GCS 및 수평적 확장성. GCS는 Ray 개발 및 디버깅을 극적으로 단순화했습니다. 이를 통해 내부 구성 요소 상태를 수동으로 노출하지 않고 Ray 자체를 디버깅하는 동안 전체 시스템 상태를 쿼리할 수 있습니다. 또한 GCS는 애플리케이션 수준 디버깅에 사용되는 타임라인 시각화 도구의 백엔드이기도 합니다.

GCS는 Ray의 수평적 확장성에도 중요한 역할을 했습니다. 섹션 5에서는 GCS가 병목 상태가 될 때마다 더 많은 샤드를 추가하여 확장할 수 있었습니다. GCS는 또한 단순히 더 많은 복제본을 추가하여 글로벌 스케줄러를 확장할 수 있도록 했습니다. 이러한 장점으로 인해 제어 상태를 중앙 집중화하는 것이 미래 분산 시스템의 핵심 설계 구성 요소가 될 것이라고 믿습니다.

## 8 결론

오늘날 어떤 범용 시스템도 훈련, 제공 및 시뮬레이션의 긴밀한 루프를 효율적으로 지원할 수 없습니다. 이러한 핵심 구성 요소를 표현하고 새로운 AI 응용 프로그램의 요구 사항을 충족하기 위해 Ray는 단일 동적 작업 그래프에서 작업 병렬 및 행위자 프로그래밍 모델을 통합하고 전역 제어 저장소 및 상향식 분산 스케줄러로 지원되는 확장 가능한 아키텍처를 사용합니다. 이 아키텍처에 의해 동시에 달성되는 프로그래밍 유연성, 높은 처리량 및 낮은 대기 시간은 리소스 요구 사항, 기간 및 기능이 다양한 작업을 생성하는 새로운 인공 지능 작업 부하에 특히 중요합니다. 우리의 평가는 초당 최대 180만 작업의 선형 확장성, 투명한 내결함성 및 여러 최신 RL 작업 부하에 대한 상당한 성능 향상을 보여줍니다. 따라서 Ray는 미래의 AI 애플리케이션 개발을 위해 유연성, 성능 및 사용 용이성의 강력한 조합을 제공합니다.

## 9 감사의 말

이 연구는 부분적으로 NSF CISE Expeditions Award CCF-1730628과 Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk 및 VMware와 NSF는 DGE-1106400을 부여합니다. 이 논문의 질을 높이는 데 도움이 된 사려 깊은 피드백을 주신 익명의 검토자와 목자 Miguel Castro에게 감사드립니다.

참조

[1] 아카. <https://akka.io/>.

[2] 아파치 애로우. <https://arrow.apache.org/>.

[3] Dask 벤치마크. <http://matthewrocklin.com/blog/work/2017/07/03/scaling>.

[4] EC2 인스턴스 요금. <https://aws.amazon.com/ec2/pricing/on-demand/>.

[5] OpenAI Baselines: 강화 학습 알고리즘의 고품질 구현. <https://github.com/openai/gym> 기준.

[6] TensorFlow 서빙. <https://www.tensorflow.org/>.

[7] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. TensorFlow: 대규모 기계 학습을 위한 시스템입니다. OSDI(운영 체제 설계 및 구현) 에 관한 제12회 USENIX 심포지엄 진행 중 . 미국 조지아 주 사바나 (2016).

[8] AGARWAL, A., BIRD, S., COZOWICZ, M., HOANG, L., LANG FORD, J., LEE, S., LI, J., MELAMED, D., OSHRI, G., RIBAS, O., SEN, S. 및 SLIVKINS, A. 다중 세계 테스트 결정 서비스. arXiv 프리프린트 arXiv:1606.03966 (2016).

[9] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, JM, AND SEARS, R. BOOM 분석: 클라우드를 위한 데이터 중심의 선언적 프로그래밍 탐색. 컴퓨터 시스템에 관한 제5차 유럽 회의 절차 ( 2010), ACM, pp. 223–236.

[10] ARMSTRONG, J., VIRDING, R., WIKSTROM, C., AND WILLIAMS, M. ERLANG의 동시 프로그래밍.

[11] BEATTIE, C., LEIBO, JZ, TEPLYASHIN, D., WARD, T., H., LEFRANCQ, A., KUTTLER VALDES, V., SADIK, A., GREEN, S. 및 WIKSTROM, C. arXiv 프리프린트 arXiv:1612.03801 (2016).

[12] BLUMOFF, RD 및 LEISERSON, CE 작업 도움으로 다중 스레드 계산 스케줄링. J. ACM 46, 5( 1999년 9월), 720–748.

[13] BROCKMAN, G., CHEUNG, V., PETTERSSON, L., SCHNEIDER, J., SCHULMAN, J., TANG, J., ZAREMBA, W. OpenAI 체육관. arXiv 프리프린트 arXiv:1606.01540 (2016).

[14] BYKOV, S., GELLER, A., KLIOT, G., LARUS, JR, PANDYA, R., AND THELIN, J. Orleans: 모두를 위한 클라우드 컴퓨팅. 클라우드 컴퓨팅에 관한 제2회 ACM 심포지엄 절차 ( 2011), ACM, p. 16.

[15] CARBONE, P., EWEN, S., FORA, G., HARIDI, S., RICHTER, S., AND TZOUMAS, K. Apache Flink의 상태 관리: 일관된 상태 저장 분산 스트림 처리. 절차 VLDB 기부. 10, 12(2017년 8월), 1718–1729.

[16] CASADO, M., FREEDMAN, MJ, PETTIT, J., LUO, J., MCKEOWN, N. 및 SHENKER, S. Ethane: 작업을 장악하고 있습니다. SIGCOMM 컴퓨팅. 공동. 37, 4(2007년 8월), 1–12.

[17] Charousset, D., SCHMIDT, TC, HIESGEN, R. 및 WAHLISCH M. Native Actors: 분산된 이기종 환경을 위한 확장 가능한 소프트웨어 플랫폼입니다 . 액터, 에이전트 및 탈중앙 제어 기반 프로그래밍에 관한 2013 년 워크숍 진행 (2013), ACM, pp. 87–96.

[18] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C. 및 ZHANG, Z. MXNet: 이기종 분산 시스템을 위한 유연하고 효율적인 기계 학습 라이브러리입니다 . 기계 학습 시스템에 대한 NIPS 워크숍 (LearningSys'16)(2016)에서.

[19] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, MJ, GONZALEZ, JE 및 STOICA, I. Clipper: 대기 시간이 짧은 온라인 예측 제공 시스템. 네트워크 시스템 설계 및 구현에 관한 14회 USENIX 심포지엄(NSDI 17) (보스턴, 매사추세츠, 2017), USENIX 협회, pp. 613–627.

[20] DEAN, J., AND GHEMAWAT, S. MapReduce: 대규모 클러스터에서 단순화된 데이터 처리. 공동. ACM 51, 1(2008년 1월), 107–113.

[21] DENNIS, JB 및 MISUNAS, DP 기본 데이터 흐름 프로세서를 위한 예비 아키텍처 . In Proceedings of the 2nd Annual Symposium on Computer Architecture (New York, NY, USA, 1975), ISCA '75, ACM, pp. 126–132.

[22] GABRIEL, E., FAGG, GE, BOSILCA, G., ANGSKUN, T., DON GARRA, JJ, SQUYRES, JM, SAHAY, V., KAMBADUR, P., BARRETT, B., LUMSDAINE, A., CASTAIN, RH, DANIEL, DJ, GRAHAM, RL 및 WOODALL, TS Open MPI: 차세대 MPI 구현의 목표, 개념 및 설계. Proceedings , 11차 유럽 PVM/ MPI 사용자 그룹 회의 (헝가리 부다페스트, 2004년 9월), pp. 97–104.

[23] GHEMAWAT, S., GOBIOFF, H. 및 LEUNG, S.-T. 구글 파일 시스템. 29–43.

[24] GONZALEZ, JE, XIN, RS, DAVE, A., CRANKSHAW, D., FRANKLIN, MJ, AND STOICA, I. GraphX: 분산 데이터 흐름 프레임워크에서 그래프 처리. 운영 체제 설계 및 구현에 관한 11회 USENIX 회의 절차 (Berkeley, CA, USA, 2014), OSDI'14, USENIX 협회 , pp. 599–613.

[25] GU\*, S., HOLLY\*, E., LILLICRAP, T. 및 LEVINE, S. 비동기 오프 정책 업데이트를 통한 로봇 조작을 위한 심층 강화 학습 . IEEE International Conference on Robotics and Automation(ICRA 2017)(2017)에서.

[26] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, AD, KATZ, R., SHENKER, S. 및 STOICA, I. Mesos: 데이터 센터에서 세분화된 리소스 공유를 위한 플랫폼입니다. 네트워크 작업 시스템 설계 및 구현에 관한 제8회 USENIX 회의 절차 (Berkeley, CA, USA, 2011), NSDI'11, USENIX 협회, pp. 295–308.

[27] HORGAN, D., QUAN, J., BUDDEN, D., BARTH-MARON, G., HESSEL, M., VAN HASSELT, H., AND SILVER, D. 분산된 우선 순위 경험 재생. 학습 표현에 관한 국제 회의 (2018).

[28] ISARD, M., BUDI, M., YU, Y., BIRRELL, A. 및 FETTERLY, D. Dryad: 순차 빌딩 블록의 분산 데이터 병렬 프로그램 . 컴퓨터 시스템 2007에 관한 제2회 ACM SIGOPS/EuroSys 유럽 회의 절차(뉴욕, 뉴욕, 미국, 2007), EuroSys '07, ACM, pp. 59–72.

[29] JIA, Y., SHELHAMER, E., DONAHUE, J., KARAYEV, S., LONG, J., GIRSHICK, R., GUADARRAMA, S., AND DARRELL, T. Caffe: 빠른 기능을 위한 컨벌루션 아키텍처 임베딩. arXiv 프리프린트 arXiv:1408.5093 (2014).

[30] JORDAN, MI, AND MITCHELL, TM 기계 학습: 추세, 관점 및 전망. 사이언스 349, 6245(2015), 255–260.

[31] LEIBIUSKY, J., EISBRUCH, G. 및 SIMONASSI, D. Storm 시작하기. 오라일리 미디어, Inc., 2012.

[32] LI, M., ANDERSEN, DG, PARK, JW, SMOLA, AJ, AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, EJ, 및 SU, B.-Y. 매개 변수 서버로 분산 기계 학습을 확장합니다 . 운영 체제 설계 및 구현에 관한 제11회 USENIX 회의 절차(미국 캘리포니아주 버클리, 2014년), OSDI'14, pp. 583–598.

[33] LOOKS, M., HERRESHOFF, M., HUTCHINS, D., AND Norvig, P. 동적 계산 그래프를 사용한 딥 러닝. arXiv 프리프린트 arXiv:1702.02181 (2017).

[34] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C. 및 HELLERSTEIN, J. GraphLab: 병렬 머신 러닝을 위한 새로운 프레임 워크. 인공 지능의 불확실성에 관한 제26차 회의 절차( 미국 버지니아 주 알링턴, 2010), UAI'10, pp. 340–349.

[35] MALEWICZ, G., AUSTERN, MH, BIK, AJ, DEHNERT, JC, HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: 대규모 그래프 처리 시스템 . 데이터 관리에 관한 2010 ACM SIGMOD 국제 회의 진행(뉴욕, 뉴욕, 미국, 2010), SIGMOD '10, ACM, pp. 135–146.

[36] MNIH, V., BADIA, AP, MIRZA, M., GRAVES, A., LILLICRAP, TP, HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Async 심층 강화 학습을 위한 시간적 방법. 기계 학습에 관한 국제 회의(2016) 에서 .

[37] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, AA, VENESS, J., BELLEMARE, MG, GRAVES, A., RIEDMILLER, M., FIDJELAND, AK, OSTROVSKI, G., 외. 심층 강화 학습을 통한 인간 수준 의 제어. 자연 518, 7540 (2015), 529–533.

[38] MURRAY, D. 데이터 종속 제어 흐름을 지원하는 분산 실행 엔진 . 케임브리지 대학교, 2012.

[39] MURRAY, DG, MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P. 및 ABADI, M. Naiad: 적시 데이터 흐름 시스템. 운영 체제 원칙에 관한 제24회 ACM 심포지엄 절차 (뉴욕, 뉴욕, 미국, 2013), SOSP '13, ACM, pp. 439–455.

[40] MURRAY, DG, SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: 분산 데이터 흐름 컴퓨팅을 위한 범용 실행 엔진. 네트워크 시스템 설계 및 구현에 관한 제8회 USENIX 회의 절차 (Berkeley, CA, USA, 2011), NSDI'11, USENIX 협회, pp. 113–126.

[41] NAIR, A., SRINIVASAN, P., BLACKWELL, S., ALCICEK, C., FEARON, R., MARIA, AD, PANNEERSHELVAM, V., SULEYMAN, M., BEATTIE, C., PETERSEN, S., LEGG, S., MNIH, V., KAVUKCUOGLU, K., AND SILVER, D. 심층 강화 학습을 위한 대규모 병렬 방법, 2015.

[42] NG, A., COATES, A., DIEL, M., GANAPATHI, V., SCHULTE, J., TSE, B., BERGER, E., AND LIANG, E. 강화를 통한 자동 반전 리튬터 비형 학습. 실험적 로봇 공학 IX (2006), 363–372.

[43] NISHIHARA, R., MORITZ, P., WANG, S., TUMANOV, A., PAUL, W., SCHLEIER-SMITH, J., LIAW, R., NIKNAMI, M., JORDAN, MI, AND STOICA, I. 실시간 기계 학습: 누락된 조각. 운영 체제의 핫 토픽에 대한 워크숍 (2017)에서.

[44] OPENAI. OpenAI Dota 2 1v1 봇. <https://openai.com/the-international/> , 2017.

[45] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M. 및 STOICA, I. Sparrow: 분산, 낮은 대기 시간 스케줄링. 운영 체제 원칙에 관한 제24차 ACM 심포지엄 절차 (뉴욕, 뉴욕, 미국, 2013), SOSP '13, ACM, pp. 69–84.

[46] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L. 및 LERER, A. PyTorch에서 자동 차별화.

[47] QU, H., MASHAYEKHI, O., TEREI, D., AND LEVIS, P. Canary: 고성능 클라우드 컴퓨팅을 위한 스케줄링 아키텍처. arXiv 프리프린트 arXiv:1602.01412 (2016).

[48] ROCKLIN, M. Dask: 차단된 알고리즘 및 작업 스케줄링을 사용한 병렬 계산 . 14th Python in Science Conference 진행 (2015), K. Huff 및 J. Bergstra, Eds., pp. 130 – 136.

[49] SALIMANS, T., HO, J., CHEN, X., AND SUTSKEVER, I. 강화 학습에 대한 확장 가능한 대안으로서의 진화 전략. arXiv 프리프린트 arXiv:1703.03864 (2017).

[50] SANFILIPPO, S. Redis: 오픈 소스 인 메모리 데이터 구조 가계. <https://redis.io/> , 2009.

[51] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. 근접 정책 최적화 알고리즘. arXiv 프리프린트 arXiv:1707.06347 (2017).

[52] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: 대규모 컴퓨팅 클러스터를 위한 유연하고 확장 가능한 스케줄러. 컴퓨터 시스템에 관한 제8회 ACM 유럽 회의(뉴욕, 뉴욕, 미국, 2013) 절차 , EuroSys '13, ACM, pp. 351–364.

[53] SERGEEV, A., AND DEL BALSIO, M. Horovod: tensorflow에서 빠르고 쉬운 분산 딥 러닝. arXiv 프리프린트 arXiv:1802.05799 (2018).

[54] SILVER, D., HUANG, A., MADDISON, CJ, GUEZ, A., SIFRE, L., VAN DEN DRIESCHE, G., SCHRITTWIESER, J. , ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., ET AL. 심층 신경망 및 트리 검색으로 바둑 게임 마스터하기 자연 529, 7587 (2016), 484–489.

[55] SILVER, D., LEVER, G., HEES, N., DEGRIS, T., WIERSTRA, D. 및 RIEDMILLER, M. 결정적 정책 기율기 알고리즘. ICML(2014)에서.

[56] SUTTON, RS 및 BARTO, AG 강화 학습: 소개. MIT 프레스 캠프리지, 1998.

[57] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. MPICH에서 집단 통신 작업의 최적화. 고성능 컴퓨팅 애플리케이션 국제 저널 19 , 1 (2005), 49–66.

[58] TIAN, Y., GONG, Q., SHANG, W., WU, Y. 및 ZITNICK, CL ELF: 실시간 전략 게임을 위한 광범위하고 가볍고 유연한 연구 플랫폼입니다. 신경 정보 처리 시스템(NIPS)의 발전(2017).

[59] TODOROV, E., EREZ, T. 및 TASSA, Y. Mujoco: 모델 기반 제어를 위한 물리 엔진. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on (2012), IEEE, pp. 5026–5033.

[60] VAN DEN BERG, J., MILLER, S., DUCKWORTH, D., HU, H., WAN, A., FU, X.-Y., GOLDBERG, K. 및 ABBEEL, P. 슈퍼 휴먼 인간 안내 데모에서 반복 학습을 사용하여 로봇에 의한 수술 작업 수행 . In Robotics and Automation (ICRA), 2010 IEEE 국제 회의(2010), IEEE, pp. 2074–2081.

[61] VAN RENESSE, R. 및 SCHNEIDER, 높은 처리량 및 가용성을 지원하는 FB 체인 복제. 운영 시스템 설계 및 구현에 관한 제6회 심포지엄 회의 진행 - 6권 (2004년 미국 캘리포니아 버클리), OSDI'04, USENIX 협회.

[62] VENKATARAMAN, S., PANDA, A., OUSTERHOUT, K., GHODSI, A., ARMBRUST, M., RECHT, B., FRANKLIN, M., STOICA, I. Drizzle: 빠르고 적응 가능한 스트림 처리 대규모로. 운영 체제 원칙에 관한 제26회 ACM 심포지엄 절차 ( 2017), SOSP '17, ACM.

[63] WHITE, T. Hadoop: 최종 가이드. 오라일리 미디어, Inc., 2012.

[64] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, MJ, SHENKER, S. 및 STOICA, I. Resilient 배포 datasets: 메모리 내 클러스터 컴퓨팅을 위한 내결함성 추상화입니다 . 네트워크 시스템 설계 및 구현(2012)에 관한 제9회 USENIX 회의 진행 , USENIX 협회, pp. 2–2.

[65] ZAHARIA, M., XIN, RS, WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, MJ , GHODSI, A., GONZALEZ, J., SHENKER, S. 및 STOICA, I. Apache Spark: 빅 데이터를 위한 통합 엔진 . 공동. ACM 59, 11(2016년 10월), 56–65.