Ray: A Distributed Framework for Emerging AI Applications

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

Abstract

The next generation of AI applications will continuously interact with the environment and learn from these interactions. These applications impose new and demanding systems requirements, both in terms of performance and flexibility. In this paper, we consider these requirements and present Ray—a distributed system to address them. Ray implements a unified interface that can express both task-parallel and actor-based computations, supported by a single dynamic execution engine. To meet the performance requirements, Ray employs a distributed scheduler and a distributed and fault-tolerant store to manage the system's control state. In our experiments, we demonstrate scaling beyond 1.8 million tasks per second and better performance than existing specialized systems for several challenging reinforcement learning applications.

1 Introduction

Over the past two decades, many organizations have been collecting—and aiming to exploit—ever-growing quantities of data. This has led to the development of a plethora of frameworks for distributed data analysis, including batch [? ? ?], streaming [? ? ?], and graph [? ? ?] processing systems. The success of these frameworks has made it possible for organizations to analyze large data sets as a core part of their business or scientific strategy, and has ushered in the age of "Big Data."

More recently, the scope of data-focused applications has expanded to encompass more complex artificial intelligence (AI) or machine learning (ML) techniques [?]. The paradigm case is that of supervised learning, where data points are accompanied by labels, and where the workhorse technology for mapping data points to labels is provided by deep neural networks. The complexity of these deep networks has led to another flurry of frameworks that focus on the training of deep neural networks and their use in prediction. These frameworks often leverage specialized hardware (e.g., GPUs and TPUs), with the goal of reducing training time in a batch setting. Examples include TensorFlow [?], MXNet [?], and PyTorch [?].

The promise of AI is, however, far broader than classical supervised learning. Emerging AI applications must increasingly operate in dynamic environments, react to changes in the environment, and take sequences of actions to accomplish long-term goals [? ?]. They must aim not only to exploit the data gathered, but also to explore the space of possible actions. These broader requirements are naturally framed within the paradigm of reinforcement learning (RL). RL deals with learning to operate continuously within an uncertain environment based on delayed and limited feedback [?]. RL-based systems have already yielded remarkable results, such as Google's AlphaGo beating a human world champion [?], and are beginning to find their way into dialogue systems, UAVs [?], and robotic manipulation [? ?].

The central goal of an RL application is to learn a policy—a mapping from the state of the environment to a choice of action—that yields effective performance over time, e.g., winning a game or piloting a drone. Finding effective policies in large-scale applications requires three main capabilities. First, RL methods often rely on simulation to evaluate policies. Simulations make it possible to explore many different choices of action sequences and to learn about the long-term consequences of those choices. Second, like their supervised learning counterparts, RL algorithms need to perform distributed training to improve the policy based on data generated through simulations or interactions with the physical environ-

[*]equal contribution

ment. Third, policies are intended to provide solutions to control problems, and thus it is necessary to serve the policy in interactive closed-loop and open-loop control scenarios.

These characteristics drive new systems requirements: a system for RL must support fine-grained computations (e.g., rendering actions in milliseconds when interacting with the real world, and performing vast numbers of simulations), must support heterogeneity both in time (e.g., a simulation may take milliseconds or hours) and in resource usage (e.g., GPUs for training and CPUs for simulations), and must support dynamic execution, as results of simulations or interactions with the environment can change future computations. Thus, we need a dynamic computation framework that handles millions of heterogeneous tasks per second at millisecond-level latencies.

Existing frameworks that have been developed for Big Data workloads or for supervised learning workloads fall short of satisfying these new requirements for RL. Bulk-synchronous parallel systems such as MapReduce [?], Apache Spark [?], and Dryad [?] do not support fine-grained simulation or policy serving. Task-parallel systems such as CIEL [?] and Dask [?] provide little support for distributed training and serving. The same is true for streaming systems such as Naiad [?] and Storm [?]. Distributed deep-learning frameworks such as TensorFlow [?] and MXNet [?] do not naturally support simulation and serving. Finally, model-serving systems such as TensorFlow Serving [?] and Clipper [?] support neither training nor simulation.

While in principle one could develop an end-to-end solution by stitching together several existing systems (e.g., Horovod [?] for distributed training, Clipper [?] for serving, and CIEL [?] for simulation), in practice this approach is untenable due to the tight coupling of these components within applications. As a result, researchers and practitioners today build one-off systems for specialized RL applications [? ? ? ? ? ?]. This approach imposes a massive systems engineering burden on the development of distributed applications by essentially pushing standard systems challenges like scheduling, fault tolerance, and data movement onto each application.

In this paper, we propose Ray, a general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications. The requirements of these workloads range from lightweight and stateless computations, such as for simulation, to long-running and stateful computations, such as for training. To satisfy these requirements, Ray implements a unified interface that can express both task-parallel and actor-based computations. Tasks enable Ray to efficiently and dynamically load balance simulations, process large inputs and state spaces (e.g., images, video), and recover from failures. In contrast, actors enable Ray to efficiently support stateful computations, such as model training, and expose shared mutable state to clients, (e.g., a parameter server). Ray implements the actor and the task abstractions on top of a single dynamic execution engine that is highly scalable and fault tolerant.

To meet the performance requirements, Ray distributes two components that are typically centralized in existing frameworks [? ? ?]: (1) the task scheduler and (2) a metadata store which maintains the computation lineage and a directory for data objects. This allows Ray to schedule millions of tasks per second with millisecond-level latencies. Furthermore, Ray provides lineage-based fault tolerance for tasks and actors, and replication-based fault tolerance for the metadata store.

While Ray supports serving, training, and simulation in the context of RL applications, this does not mean that it should be viewed as a replacement for systems that provide solutions for these workloads in other contexts. In particular, Ray does not aim to substitute for serving systems like Clipper [?] and TensorFlow Serving [?], as these systems address a broader set of challenges in deploying models, including model management, testing, and model composition. Similarly, despite its flexibility, Ray is not a substitute for generic data-parallel frameworks, such as Spark [?], as it currently lacks the rich functionality and APIs (e.g., straggler mitigation, query optimization) that these frameworks provide.

We make the following contributions:

- We design and build the first distributed framework that unifies training, simulation, and serving—necessary components of emerging RL applications.

- To support these workloads, we unify the actor and task-parallel abstractions on top of a dynamic task execution engine.

- To achieve scalability and fault tolerance, we propose a system design principle in which control state is stored in a sharded metadata store and all other system components are stateless.

- To achieve scalability, we propose a bottom-up distributed scheduling strategy.

Figure 1: Python code implementing the example in Figure ?? in Ray. Note that @ray.remote indicates remote functions and actors. Invocations of remote functions and actor methods return futures, which can be passed to subsequent remote functions or actor methods to encode task dependencies. Each actor has an environment object self.env shared between all of its methods.

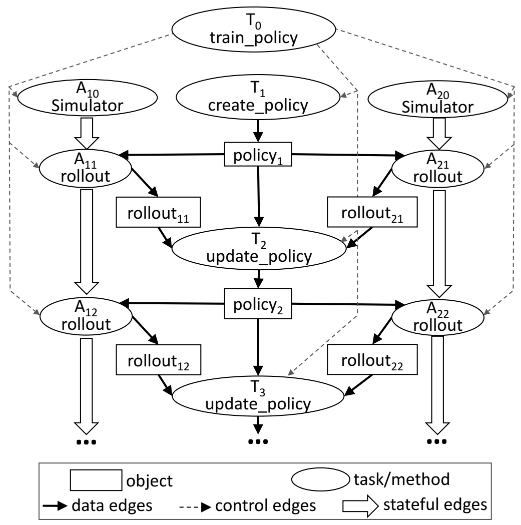


Figure 2: The task graph corresponding to an invocation of train_policy.remote() in Figure ??. Remote function calls and the actor method calls correspond to tasks in the task graph. The figure shows two actors. The method invocations for each actor (the tasks labeled A_{1i} and A_{2i}) have stateful edges between them indicating that they share the mutable actor state. There are control edges from train_policy to the tasks that it invokes. To train multiple policies in parallel, we could call train_policy.remote() multiple times.

2 Programming and Computation Model

Ray implements a dynamic task graph computation model, i.e., it models an application as a graph of dependent tasks that evolves during execution. On top of this model, Ray provides both an actor and a task-parallel programming abstraction. This unification differentiates Ray from related systems like CIEL, which only provides a task-parallel abstraction, and from Orleans [?] or Akka [?], which primarily provide an actor abstraction.

2.1 Programming Model

Tasks. A task represents the execution of a remote function on a stateless worker. When a remote function is invoked, a future representing the result of the task is returned immediately. Futures can be retrieved using **ray**.**get**() and passed as arguments into other remote functions without waiting for their result. This allows the user to express parallelism while capturing data dependencies. Table ?? shows Ray's API.

Remote functions operate on immutable objects and are expected to be stateless and side-effect free: their outputs are determined solely by their inputs. This implies idempotence, which simplifies fault tolerance through function re-execution on failure.

Actors. An actor represents a stateful computation. Each actor exposes methods that can be invoked remotely and are executed serially. A method execution is similar to a task, in that it executes remotely and returns a future, but differs in that it executes on a stateful worker. A handle to an actor can be passed to other actors or tasks, making it possible for them to invoke methods on that actor.

Table ?? summarizes the properties of tasks and actors. Tasks enable fine-grained load balancing through leveraging load-aware scheduling at task granularity, input data locality, as each task can be scheduled on the node storing its inputs, and low recovery overhead, as there is no need to checkpoint and recover intermediate state. In contrast, actors provide much more efficient fine-grained updates, as these updates are performed on internal rather than external state, which typically requires serialization and deserialization. For example, actors can be used to implement parameter servers [?] and GPU-based iterative computations (e.g., training). In addition, actors can be used to wrap third-party simulators and other opaque handles that are hard to serialize.

To satisfy the requirements for heterogeneity and flexibility (Section ??), we augment the API in three ways. First, to handle concurrent tasks with heterogeneous durations, we introduce **ray**.**wait**(), which waits for the first k available results, instead of waiting for all results like **ray**.**get**(). Second, to handle resource-heterogeneous tasks, we enable developers to specify resource requirements so that the Ray scheduler can efficiently manage resources. Third, to improve flexibility, we enable nested remote functions, meaning that remote functions can invoke other remote functions. This is also critical for achieving high scalability (Section ??), as it enables multiple processes to invoke remote functions in a distributed fashion.

2.2 Computation Model

Ray employs a dynamic task graph computation model [?], in which the execution of both remote functions and actor methods is automatically triggered by the system when their inputs become available. In this section, we describe how the computation graph (Figure ??) is constructed from a user program (Figure ??). This program uses the API in Table ?? to implement the pseudocode from Figure ??.

Ignoring actors first, there are two types of nodes in a computation graph: data objects and remote function invocations, or tasks. There are also two types of edges: data edges and control edges. Data edges capture the dependencies between data objects and tasks. More precisely, if data object D is an output of task T, we add a data edge from T to D. Similarly, if D is an input to T, we add a data edge from D to T. Control edges capture the computation dependencies that result from nested remote functions (Section ??): if task T_1 invokes task T_2, then we add a control edge from T_1 to T_2.

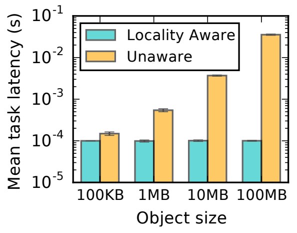
Actor method invocations are also represented as nodes in the computation graph. They are identical to tasks with one key difference. To capture the state dependency across subsequent method invocations on the same actor, we add a third type of edge: a stateful edge. If method M_j is called right after method M_i on the same actor, then we add a stateful edge from M_i to M_j. Thus, all methods invoked on the same actor object form a chain that is connected by stateful edges (Figure ??). This chain captures the order in which these methods were invoked.

Stateful edges help us embed actors in an otherwise stateless task graph, as they capture the implicit data dependency between successive method invocations sharing the internal state of an actor. Stateful edges also enable us to maintain lineage. As in other dataflow systems [?], we track data lineage to enable reconstruction. By explicitly including stateful edges in the lineage graph, we can easily reconstruct lost data, whether produced by remote functions or actor methods (Section ??).

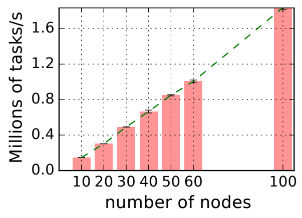
3 Evaluation

In our evaluation, we study the following questions:

1. How well does Ray meet the latency, scalability, and fault tolerance requirements listed in Section ??? (Section ??)
2. What overheads are imposed on distributed primitives (e.g., allreduce) written using Ray's API? (Section ??)
3. In the context of RL workloads, how does Ray compare against specialized systems for training, serving, and simulation? (Section ??)
4. What advantages does Ray provide for RL applications, compared to custom systems? (Section ??)



(a) Ray locality scheduling



(b) Ray scalability

Figure 3: (a) Tasks leverage locality-aware placement. 1000 tasks with a random object dependency are scheduled onto one of two nodes. With locality-aware policy, task latency remains independent of the size of task inputs instead of growing by 1-2 orders of magnitude. (b) Near-linear scalability leveraging the GCS and bottom-up distributed scheduler. Ray reaches 1 million tasks per second throughput with 60 nodes. $x \in \{70, 80, 90\}$ omitted due to cost.

All experiments were run on Amazon Web Services. Unless otherwise stated, we use m4.16xlarge CPU instances and p3.16xlarge GPU instances.

3.1 Microbenchmarks

Locality-aware task placement. Fine-grain load balancing and locality-aware placement are primary benefits of tasks in Ray. Actors, once placed, are unable to move their computation to large remote objects, while tasks can. In Figure ??, tasks placed without data locality awareness (as is the case for actor methods), suffer 1-2 orders of magnitude latency increase at 10-100MB input data sizes. Ray unifies tasks and actors through the shared object store, allowing developers to use tasks for e.g., expensive postprocessing on output produced by simulation actors.

End-to-end scalability. One of the key benefits of the Global Control Store (GCS) and the bottom-up distributed scheduler is the ability to horizontally scale the system to support a high throughput of fine-grained tasks, while maintaining fault tolerance and low-latency task scheduling. In Figure ??, we evaluate this ability on an embarrassingly parallel workload of empty tasks, increasing the cluster size on the x-axis. We observe near-perfect linearity in progressively increasing task throughput. Ray exceeds 1 million tasks per second throughput at 60 nodes and continues to scale linearly beyond 1.8 million tasks per second at 100 nodes. The rightmost datapoint shows that Ray can process 100 million tasks in less than a minute (54s), with minimum variability. As

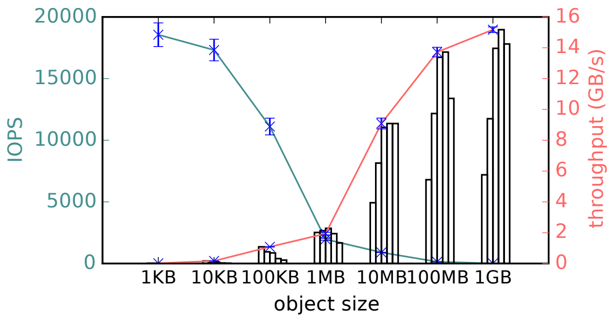
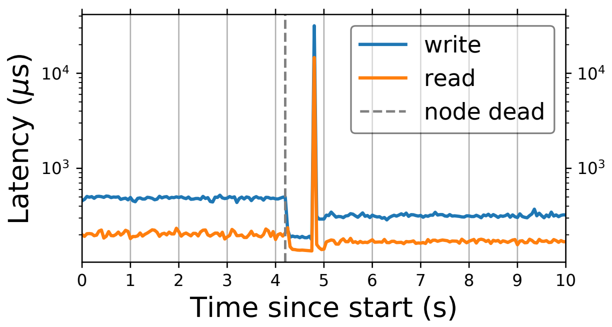
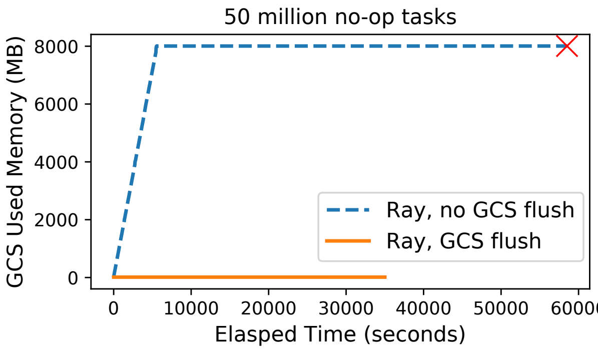


Figure 4: Object store write throughput and IOPS. From a single client, throughput exceeds 15GB/s (red) for large objects and 18K IOPS (cyan) for small objects on a 16 core instance (m4.4xlarge). It uses 8 threads to copy objects larger than 0.5MB and 1 thread for small objects. Bar plots report throughput with 1, 2, 4, 8, 16 threads. Results are averaged over 5 runs.



(a) A timeline for GCS read and write latencies as viewed from a client submitting tasks. The chain starts with 2 replicas. We manually trigger reconfiguration as follows. At $t \approx 4.2$s, a chain member is killed; immediately after, a new chain member joins, initiates state transfer, and restores the chain to 2-way replication. The maximum client-observed latency is under 30ms despite reconfigurations.



(b) The Ray GCS maintains a constant memory footprint with GCS flushing. Without GCS flushing, the memory footprint reaches a maximum capacity and the workload fails to complete within a predetermined duration (indicated by the red cross).

Figure 5: Ray GCS fault tolerance and flushing.

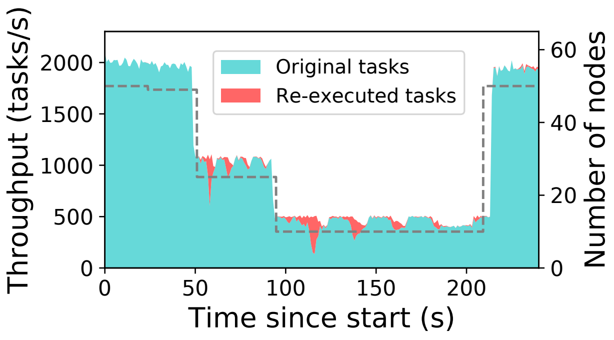
expected, increasing task duration reduces throughput proportionally to mean task duration, but the overall scalability remains linear. While many realistic workloads may exhibit more limited scalability due to object dependencies and inherent limits to application parallelism, this demonstrates the scalability of our overall architecture under high load.

Object store performance. To evaluate the performance of the object store (Section ??), we track two metrics: IOPS (for small objects) and write throughput (for large objects). In Figure ??, the write throughput from a single client exceeds 15GB/s as object size increases. For larger objects, memcpy dominates object creation time. For smaller objects, the main overheads are in serialization and IPC between the client and object store.

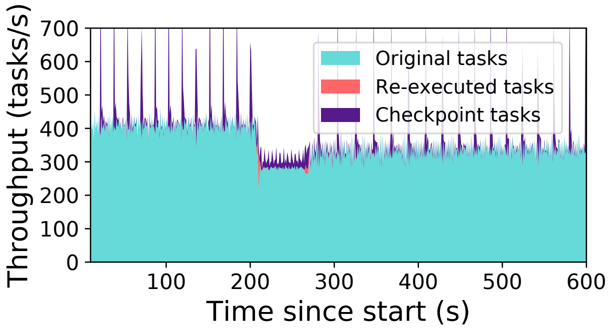
GCS fault tolerance. To maintain low latency while providing strong consistency and fault tolerance, we build a lightweight chain replication [?] layer on top of Redis. Figure ?? simulates recording Ray tasks to and reading tasks from the GCS, where keys are 25 bytes and values are 512 bytes. The client sends requests as fast as it can, having at most one in-flight request at a time. Failures are reported to the chain master either from the client (having received explicit errors, or timeouts despite retries) or from any server in the chain (having received explicit errors). Overall, reconfigurations caused a maximum client-observed delay of under 30ms (this includes both failure detection and recovery delays).

GCS flushing. Ray is equipped to periodically flush the contents of GCS to disk. In Figure ?? we submit 50 million empty tasks sequentially and monitor GCS memory consumption. As expected, it grows linearly with the number of tasks tracked and eventually reaches the memory capacity of the system. At that point, the system becomes stalled and the workload fails to finish within a reasonable amount of time. With periodic GCS flushing, we achieve two goals. First, the memory footprint is capped at a user-configurable level (in the microbenchmark we employ an aggressive strategy where consumed memory is kept as low as possible). Second, the flushing mechanism provides a natural way to snapshot lineage to disk for long-running Ray applications.

Recovering from task failures. In Figure ??, we demonstrate Ray's ability to transparently recover from worker node failures and elastically scale, using the durable GCS lineage storage. The workload, run on m4.xlarge instances, consists of linear chains of 100ms tasks submitted by the driver. As nodes



(a) Task reconstruction



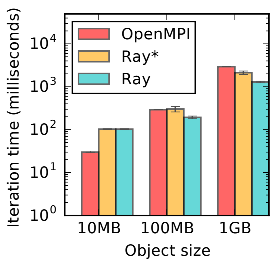
(b) Actor reconstruction

Figure 6: Ray fault-tolerance. (a) Ray reconstructs lost task dependencies as nodes are removed (dotted line), and recovers to original throughput when nodes are added back. Each task is 100ms and depends on an object generated by a previously submitted task. (b) Actors are reconstructed from their last checkpoint. At $t = 200$s, we kill 2 of the 10 nodes, causing 400 of the 2000 actors in the cluster to be recovered on the remaining nodes ($t = 200$–270s).

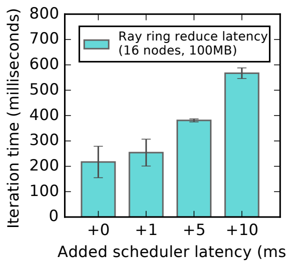
are removed (at 25s, 50s, 100s), the local schedulers reconstruct previous results in the chain in order to continue execution. Overall per-node throughput remains stable throughout.

Recovering from actor failures. By encoding actor method calls as stateful edges directly in the dependency graph, we can reuse the same object reconstruction mechanism as in Figure ?? to provide transparent fault tolerance for stateful computation. Ray additionally leverages user-defined checkpoint functions to bound the reconstruction time for actors (Figure ??). With minimal overhead, checkpointing enables only 500 methods to be re-executed, versus 10k re-executions without checkpointing. In the future, we hope to further reduce actor reconstruction time, e.g., by allowing users to annotate methods that do not mutate state.

Allreduce. Allreduce is a distributed communication primitive important to many machine learning workloads. Here, we evaluate whether Ray can na-



(a) Ray vs OpenMPI



(b) Ray scheduler ablation

Figure 7: (a) Mean execution time of allreduce on 16 m4.16xl nodes. Each worker runs on a distinct node. Ray* restricts Ray to 1 thread for sending and 1 thread for receiving. (b) Ray's low-latency scheduling is critical for allreduce.

tively support a ring allreduce [?] implementation with low enough overhead to match existing implementations [?]. We find that Ray completes allreduce across 16 nodes on 100MB in ∼200ms and 1GB in ∼1200ms, surprisingly outperforming OpenMPI (v1.10), a popular MPI implementation, by 1.5× and 2× respectively (Figure ??). We attribute Ray's performance to its use of multiple threads for network transfers, taking full advantage of the 25Gbps connection between nodes on AWS, whereas OpenMPI sequentially sends and receives data on a single thread [?]. For smaller objects, OpenMPI outperforms Ray by switching to a lower overhead algorithm, an optimization we plan to implement in the future.

Ray's scheduler performance is critical to implementing primitives such as allreduce. In Figure ??, we inject artificial task execution delays and show that performance drops nearly 2× with just a few ms of extra latency. Systems with centralized schedulers like Spark and CIEL typically have scheduler overheads in the tens of milliseconds [? ?], making such workloads impractical. Scheduler throughput also becomes a bottleneck since the number of tasks required by ring reduce scales quadratically with the number of participants.

3.2 Building blocks

End-to-end applications (e.g., AlphaGo [?]) require a tight coupling of training, serving, and simulation. In this section, we isolate each of these workloads to a setting that illustrates a typical RL application's requirements. Due to a flexible programming model targeted to RL, and a system designed to support this programming model, Ray matches and sometimes exceeds the performance of dedicated systems for these individual workloads.

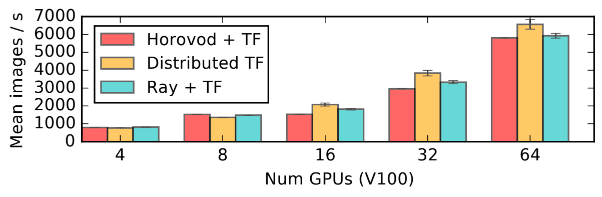


Figure 8: Images per second reached when distributing the training of a ResNet-101 TensorFlow model (from the official TF benchmark). All experiments were run on p3.16xl instances connected by 25Gbps Ethernet, and workers allocated 4 GPUs per node as done in Horovod [?]. We note some measurement deviations from previously reported, likely due to hardware differences and recent TensorFlow performance improvements. We used Open-MPI 3.0, TF 1.8, and NCCL2 for all runs.

3.2.1 Distributed Training

We implement data-parallel synchronous SGD leveraging the Ray actor abstraction to represent model replicas. Model weights are synchronized via allreduce (??) or parameter server, both implemented on top of the Ray API.

In Figure ??, we evaluate the performance of the Ray (synchronous) parameter-server SGD implementation against state-of-the-art implementations [?], using the same TensorFlow model and synthetic data generator for each experiment. We compare only against TensorFlow-based systems to accurately measure the overhead imposed by Ray, rather than differences between the deep learning frameworks themselves. In each iteration, model replica actors compute gradients in parallel, send the gradients to a sharded parameter server, then read the summed gradients from the parameter server for the next iteration.

Figure ?? shows that Ray matches the performance of Horovod and is within 10% of distributed TensorFlow (in `distributed_replicated` mode). This is due to the ability to express the same application-level optimizations found in these specialized systems in Ray's general-purpose API. A key optimization is the pipelining of gradient computation, transfer, and summation within a single iteration. To overlap GPU computation with network transfer, we use a custom TensorFlow operator to write tensors directly to Ray's object store.

3.2.2 Serving

Model serving is an important component of end-to-end applications. Ray focuses primarily on the embedded serving of models to simulators running within the same dynamic task graph (e.g., within an RL application on Ray). In contrast, systems like Clipper [?] focus on serving predictions to external clients.

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

Table 1: Throughput comparisons for Clipper [?], a dedicated serving system, and Ray for two embedded serving workloads. We use a residual network and a small fully connected network, taking 10ms and 5ms to evaluate, respectively. The server is queried by clients that each send states of size 4KB and 100KB respectively in batches of 64.

In this setting, low latency is critical for achieving high utilization. To show this, in Table ?? we compare the server throughput achieved using a Ray actor to serve a policy versus using the open source Clipper system over REST. Here, both client and server processes are co-located on the same machine (a p3.8xlarge instance). This is often the case for RL applications but not for the general web serving workloads addressed by systems like Clipper. Due to its low-overhead serialization and shared memory abstractions, Ray achieves an order of magnitude higher throughput for a small fully connected policy model that takes in a large input and is also faster on a more expensive residual network policy model, similar to one used in AlphaGo Zero, that takes smaller input.

3.2.3 Simulation

Simulators used in RL produce results with variable lengths ("timesteps") that, due to the tight loop with training, must be used as soon as they are available. The task heterogeneity and timeliness requirements make simulations hard to support efficiently in BSP-style systems. To demonstrate, we compare (1) an MPI implementation that submits $3n$ parallel simulation runs on n cores in 3 rounds, with a global barrier between rounds*, to (2) a Ray program that issues the same $3n$ tasks while concurrently gathering simulation results back to the driver. Table ?? shows that both systems scale well, yet Ray achieves up to 1.8× throughput. This motivates a programming model that can dynamically spawn and collect the results of fine-grained simulation tasks.

*Note that experts can use MPI's asynchronous primitives to get around barriers—at the expense of increased program complexity —we nonetheless chose such an implementation to simulate BSP.

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

Table 2: Timesteps per second for the Pendulum-v0 simulator in OpenAI Gym [?]. Ray allows for better utilization when running heterogeneous simulations at scale.

3.3 RL Applications

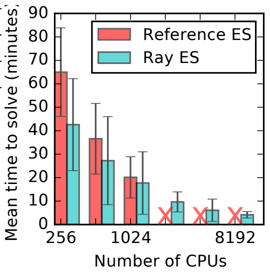
Without a system that can tightly couple the training, simulation, and serving steps, reinforcement learning algorithms today are implemented as one-off solutions that make it difficult to incorporate optimizations that, for example, require a different computation structure or that utilize different architectures. Consequently, with implementations of two representative reinforcement learning applications in Ray, we are able to match and even outperform custom systems built specifically for these algorithms. The primary reason is the flexibility of Ray's programming model, which can express application-level optimizations that would require substantial engineering effort to port to custom-built systems, but are transparently supported by Ray's dynamic task graph execution engine.

3.3.1 Evolution Strategies

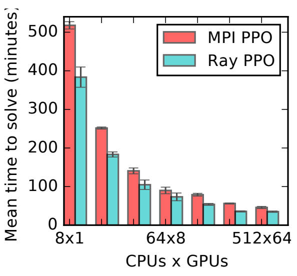
To evaluate Ray on large-scale RL workloads, we implement the evolution strategies (ES) algorithm and compare to the reference implementation [?]—a system specially built for this algorithm that relies on Redis for messaging and low-level multiprocessing libraries for data-sharing. The algorithm periodically broadcasts a new policy to a pool of workers and aggregates the results of roughly 10000 tasks (each performing 10 to 1000 simulation steps).

As shown in Figure ??, an implementation on Ray scales to 8192 cores. Doubling the cores available yields an average completion time speedup of 1.6×. Conversely, the special-purpose system fails to complete at 2048 cores, where the work in the system exceeds the processing capacity of the application driver. To avoid this issue, the Ray implementation uses an aggregation tree of actors, reaching a median time of 3.7 minutes, more than twice as fast as the best published result (10 minutes).

Initial parallelization of a serial implementation using Ray required modifying only 7 lines of code. Performance improvement through hierarchical aggregation was easy to realize with Ray's support for nested tasks and actors. In contrast, the reference



(a) Evolution Strategies



(b) PPO

Figure 9: Time to reach a score of 6000 in the Humanoid-v1 task [?]. (a) The Ray ES implementation scales well to 8192 cores and achieves a median time of 3.7 minutes, over twice as fast as the best published result. The special-purpose system failed to run beyond 1024 cores. ES is faster than PPO on this benchmark, but shows greater runtime variance. (b) The Ray PPO implementation outperforms a specialized MPI implementation [?] with fewer GPUs, at a fraction of the cost. The MPI implementation required 1 GPU for every 8 CPUs, whereas the Ray version required at most 8 GPUs (and never more than 1 GPU per 8 CPUs).

implementation had several hundred lines of code dedicated to a protocol for communicating tasks and data between workers, and would require further engineering to support optimizations like hierarchical aggregation.

3.3.2 Proximal Policy Optimization

We implement Proximal Policy Optimization (PPO) [?] in Ray and compare to a highly-optimized reference implementation [?] that uses OpenMPI communication primitives. The algorithm is an asynchronous scatter-gather, where new tasks are assigned to simulation actors as they return rollouts to the driver. Tasks are submitted until 320000 simulation steps are collected (each task produces between 10 and 1000 steps). The policy update performs 20 steps of SGD with a batch size of 32768. The model parameters in this example are roughly 350KB. These experiments were run using p2.16xlarge (GPU) and m4.16xlarge (high CPU) instances.

As shown in Figure ??, the Ray implementation outperforms the optimized MPI implementation in all experiments, while using a fraction of the GPUs. The reason is that Ray is heterogeneity-aware and allows the user to utilize asymmetric architectures by expressing resource requirements at the granularity of a task or actor. The Ray implementation can then leverage TensorFlow's single-process multi-GPU support and can pin objects in GPU memory when possible. This optimization cannot be easily ported to

MPI due to the need to asynchronously gather rollouts to a single GPU process. Indeed, [?] includes two custom implementations of PPO, one using MPI for large clusters and one that is optimized for GPUs but that is restricted to a single node. Ray allows for an implementation suitable for both scenarios.

Ray's ability to handle resource heterogeneity also decreased PPO's cost by a factor of 4.5 [?], since CPU-only tasks can be scheduled on cheaper high-CPU instances. In contrast, MPI applications often exhibit symmetric architectures, in which all processes run the same code and require identical resources, in this case preventing the use of CPU-only machines for scale-out. Furthermore, the MPI implementation requires on-demand instances since it does not transparently handle failure. Assuming 4× cheaper spot instances, Ray's fault tolerance and resource-aware scheduling together cut costs by 18×.

4 Related Work

Dynamic task graphs. Ray is closely related to CIEL [?] and Dask [?]. All three support dynamic task graphs with nested tasks and implement the futures abstraction. CIEL also provides lineage-based fault tolerance, while Dask, like Ray, fully integrates with Python. However, Ray differs in two aspects that have important performance consequences. First, Ray extends the task model with an actor abstraction. This is necessary for efficient stateful computation in distributed training and serving, to keep the model data collocated with the computation. Second, Ray employs a fully distributed and decoupled control plane and scheduler, instead of relying on a single master storing all metadata. This is critical for efficiently supporting primitives like allreduce without system modification. At peak performance for 100MB on 16 nodes, allreduce on Ray (Section ??) submits 32 rounds of 16 tasks in 200ms. Meanwhile, Dask reports a maximum scheduler throughput of 3k tasks/s on 512 cores [?]. With a centralized scheduler, each round of allreduce would then incur a minimum of ∼5ms of scheduling delay, translating to up to 2× worse completion time (Figure ??). Even with a decentralized scheduler, coupling the control plane information with the scheduler leaves the latter on the critical path for data transfer, adding an extra roundtrip to every round of allreduce.

Dataflow systems. Popular dataflow systems, such as MapReduce [?], Spark [?], and Dryad [?] have widespread adoption for analytics and ML workloads, but their computation model is too restrictive for a fine-grained and dynamic simulation workload. Spark and MapReduce implement the BSP execution model, which assumes that tasks within the same stage perform the same computation and take roughly the same amount of time. Dryad relaxes this restriction but lacks support for dynamic task graphs. Furthermore, none of these systems provide an actor abstraction, nor implement a distributed scalable control plane and scheduler. Finally, Naiad [?] is a dataflow system that provides improved scalability for some workloads, but only supports static task graphs.

Machine learning frameworks. TensorFlow [?] and MXNet [?] target deep learning workloads and efficiently leverage both CPUs and GPUs. While they achieve great performance for training workloads consisting of static DAGs of linear algebra operations, they have limited support for the more general computation required to tightly couple training with simulation and embedded serving. TensorFlow Fold [?] provides some support for dynamic task graphs, as well as MXNet through its internal C++ APIs, but neither fully supports the ability to modify the DAG during execution in response to task progress, task completion times, or faults. TensorFlow and MXNet in principle achieve generality by allowing the programmer to simulate low-level message-passing and synchronization primitives, but the pitfalls and user experience in this case are similar to those of MPI. OpenMPI [?] can achieve high performance, but it is relatively hard to program as it requires explicit coordination to handle heterogeneous and dynamic task graphs. Furthermore, it forces the programmer to explicitly handle fault tolerance.

Actor systems. Orleans [?] and Akka [?] are two actor frameworks well suited to developing highly available and concurrent distributed systems. However, compared to Ray, they provide less support for recovery from data loss. To recover stateful actors, the Orleans developer must explicitly checkpoint actor state and intermediate responses. Stateless actors in Orleans can be replicated for scale-out, and could therefore act as tasks, but unlike in Ray, they have no lineage. Similarly, while Akka explicitly supports persisting actor state across failures, it does not provide efficient fault tolerance for stateless computation (i.e., tasks). For message delivery, Orleans provides at-least-once and Akka provides at-most-once semantics. In contrast, Ray provides transparent fault tolerance and exactly-once semantics, as each method call is logged in the GCS and both arguments and results are immutable. We find that in practice these limitations do not affect the performance of our applications. Erlang [?] and C++ Actor Framework [?], two other actor-based systems, have similarly limited support for fault tolerance.

Global control store and scheduling. The concept of logically centralizing the control plane has been previously proposed in software defined networks (SDNs) [?], distributed file systems (e.g., GFS [?]), resource management (e.g., Omega [?]), and distributed frameworks (e.g., MapReduce [?], BOOM [?]), to name a few. Ray draws inspiration from these pioneering efforts, but provides significant improvements. In contrast with SDNs, BOOM, and GFS, Ray decouples the storage of the control plane information (e.g., GCS) from the logic implementation (e.g., schedulers). This allows both storage and computation layers to scale independently, which is key to achieving our scalability targets. Omega uses a distributed architecture in which schedulers coordinate via globally shared state. To this architecture, Ray adds global schedulers to balance load across local schedulers, and targets ms-level, not second-level, task scheduling.

Ray implements a unique distributed bottom-up scheduler that is horizontally scalable, and can handle dynamically constructed task graphs. Unlike Ray, most existing cluster computing systems [? ? ?] use a centralized scheduler architecture. While Sparrow [?] is decentralized, its schedulers make independent decisions, limiting the possible scheduling policies, and all tasks of a job are handled by the same global scheduler. Mesos [?] implements a two-level hierarchical scheduler, but its top-level scheduler manages frameworks, not individual tasks. Canary [?] achieves impressive performance by having each scheduler instance handle a portion of the task graph, but does not handle dynamic computation graphs.

Cilk [?] is a parallel programming language whose work-stealing scheduler achieves provably efficient load-balancing for dynamic task graphs. However, with no central coordinator like Ray's global scheduler, this fully parallel design is also difficult to extend to support data locality and resource heterogeneity in a distributed setting.

5 Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1730628 and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware as well as by NSF grant DGE-1106400. We are grateful to our anonymous reviewers and our shepherd, Miguel Castro, for thoughtful feedback, which helped improve the quality of this paper.

References

[1] Akka. `https://akka.io/`.

[2] Apache Arrow. `https://arrow.apache.org/`.

[3] Dask Benchmarks. `http://matthewrocklin.com/blog/work/2017/07/03/scaling`.

[4] EC2 Instance Pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`.

[5] OpenAI Baselines: high-quality implementations of reinforcement learning algorithms. `https://github.com/openai/baselines`.

[6] TensorFlow Serving. `https://www.tensorflow.org/serving/`.

[7] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA (2016).

[8] Agarwal, A., Bird, S., Cozowicz, M., Hoang, L., Langford, J., Lee, S., Li, J., Melamed, D., Oshri, G., Ribas, O., Sen, S., and Slivkins, A. A multiworld testing decision service. arXiv preprint arXiv:1606.03966 (2016).

[9] Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J. M., and Sears, R. BOOM Analytics: exploring data-centric, declarative programming for the cloud. In Proceedings of the 5th European conference on Computer systems (2010), ACM, pp. 223–236.

[10] Armstrong, J., Virding, R., Wikström, C., and Williams, M. Concurrent programming in ERLANG.

[11] Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., et al. DeepMind Lab. arXiv preprint arXiv:1612.03801 (2016).

[12] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. J. ACM 46, 5 (Sept. 1999), 720–748.

[13] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI gym. arXiv preprint arXiv:1606.01540 (2016).

[14] Bykov, S., Geller, A., Kliot, G., Larus, J. R., Pandya, R., and Thelin, J. Orleans: Cloud computing for everyone. In Proceedings of the 2nd ACM Symposium on Cloud Computing (2011), ACM, p. 16.

[15] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., and Tzoumas, K. State management in Apache Flink: Consistent stateful distributed stream processing. Proc. VLDB Endow. 10, 12 (Aug. 2017), 1718–1729.

[16] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking control of the enterprise. SIGCOMM Comput. Commun. Rev. 37, 4 (Aug. 2007), 1–12.

[17] Charousset, D., Schmidt, T. C., Hiesgen, R., and Wählisch, M. Native actors: A scalable software platform for distributed, heterogeneous environments. In Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control (2013), ACM, pp. 87–96.

[18] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In NIPS Workshop on Machine Learning Systems (LearningSys'16) (2016).

[19] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) (Boston, MA, 2017), USENIX Association, pp. 613–627.

[20] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. Commun. ACM 51, 1 (Jan. 2008), 107–113.

[21] Dennis, J. B., and Misunas, D. P. A preliminary architecture for a basic data-flow processor. 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, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting (Budapest, Hungary, September 2004), pp. 97–104.

[23] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. 29–43.

[24] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 599–613.

[25] Gu*, S., Holly*, E., Lillicrap, T., and Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In IEEE International Conference on Robotics and Automation (ICRA 2017) (2017).

[26] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[27] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. Distributed prioritized experience replay. International Conference on Learning Representations (2018).

[28] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (New York, NY, USA, 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: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014).

[30] Jordan, M. I., and Mitchell, T. M. Machine learning: Trends, perspectives, and prospects. Science 349, 6245 (2015), 255–260.

[31] Leibiusky, J., Eisbruch, G., and Simonassi, D. Getting Started with Storm. O'Reilly Media, Inc., 2012.

[32] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2014), OSDI'14, pp. 583–598.

[33] Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. arXiv preprint arXiv:1702.02181 (2017).

[34] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. GraphLab: A new framework for parallel machine learning. In Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (Arlington, Virginia, United States, 2010), UAI'10, pp. 340–349.

[35] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[36] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning (2016).

[37] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. Nature 518, 7540 (2015), 529–533.

[38] Murray, D. A Distributed Execution Engine Supporting Data-dependent Control Flow. University of Cambridge, 2012.

[39] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[40] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. CIEL: A universal execution engine for distributed data-flow computing. In Proceedings of the 8th USENIX Conference on

Networked Systems Design and Implementation (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 113–126.

[41] Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., Maria, A. D., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., and Silver, D. Massively parallel methods for deep reinforcement learning, 2015.

[42] Ng, A., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. Autonomous inverted helicopter flight via reinforcement learning. Experimental Robotics IX (2006), 363–372.

[43] Nishihara, R., Moritz, P., Wang, S., Tumanov, A., Paul, W., Schleier-Smith, J., Liaw, R., Niknami, M., Jordan, M. I., and Stoica, I. Real-time machine learning: The missing pieces. In Workshop on Hot Topics in Operating Systems (2017).

[44] OpenAI. OpenAI Dota 2 1v1 bot. `https://openai.com/the-international/`, 2017.

[45] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: Distributed, low latency scheduling. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (New York, NY, USA, 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., and Lerer, A. Automatic differentiation in PyTorch.

[47] Qu, H., Mashayekhi, O., Terei, D., and Levis, P. Canary: A scheduling architecture for high performance cloud computing. arXiv preprint arXiv:1602.01412 (2016).

[48] Rocklin, M. Dask: Parallel computation with blocked algorithms and task scheduling. In Proceedings of the 14th Python in Science Conference (2015), K. Huff and J. Bergstra, Eds., pp. 130 – 136.

[49] Salimans, T., Ho, J., Chen, X., and Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864 (2017).

[50] Sanfilippo, S. Redis: An open source, in-memory data structure store. `https://redis.io/`, 2009.

[51] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017).

[52] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: Flexible, scalable schedulers for large compute clusters. In Proceedings of the 8th ACM European Conference on Computer Systems (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 351–364.

[53] Sergeev, A., and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. arXiv preprint arXiv:1802.05799 (2018).

[54] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 7587 (2016), 484–489.

[55] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. Deterministic policy gradient algorithms. In ICML (2014).

[56] Sutton, R. S., and Barto, A. G. Reinforcement Learning: An Introduction. MIT press Cambridge, 1998.

[57] Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in MPICH. The International Journal of High Performance Computing Applications 19, 1 (2005), 49–66.

[58] Tian, Y., Gong, Q., Shang, W., Wu, Y., and Zitnick, C. L. ELF: An extensive, lightweight and flexible research platform for real-time strategy games. Advances in Neural Information Processing Systems (NIPS) (2017).

[59] Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. 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., and Abbeel, P. Superhuman performance of surgical tasks by robots using iterative learning from human-guided demonstrations. In Robotics and Automation (ICRA), 2010 IEEE International Conference on (2010), IEEE, pp. 2074–2081.

[61] van Renesse, R., and Schneider, F. B. Chain replication for supporting high throughput and availability. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association.

[62] Venkataraman, S., Panda, A., Ousterhout, K., Ghodsi, A., Armbrust, M., Recht, B., Franklin, M., and Stoica, I. Drizzle: Fast and adaptable stream processing at scale. In Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles (2017), SOSP '17, ACM.

[63] White, T. Hadoop: The Definitive Guide. O'Reilly Media, Inc., 2012.

[64] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. 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), USENIX Association, pp. 2–2.

[65] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache Spark: A unified engine for big data processing. Commun. ACM 59, 11 (Oct. 2016), 56–65.