



Figure 13: Images per second reached when distributing the training of a ResNet-101 TensorFlow model (from the official TF benchmark). All experiments were run on p3.16xlarge instances connected by 25Gbps Ethernet, and workers allocated 4 GPUs per node as done in Horovod [53]. We note some measurement deviations from previously reported, likely due to hardware differences and recent TensorFlow performance improvements. We used OpenMPI 3.0, TF 1.8, and NCCL2 for all runs.

5.2.1 Distributed Training

We implement data-parallel synchronous SGD leveraging the Ray actor abstraction to represent model replicas. Model weights are synchronized via allreduce (5.1) or parameter server, both implemented on top of the Ray API.

In Figure 13, we evaluate the performance of the Ray (synchronous) parameter-server SGD implementation against state-of-the-art implementations [53], 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 13 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.

5.2.2 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 [19] focus on serving predictions to external clients.

In this setting, low latency is critical for achieving high utilization. To show this, in Table 3 we compare the

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

Table 3: Throughput comparisons for Clipper [19], 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.

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.

5.2.3 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 4 shows that both systems scale well, yet Ray achieves up to $1.8\times$ throughput. This motivates a programming model that can dynamically spawn and collect the results of fine-grained simulation tasks.

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 4: Timesteps per second for the Pendulum-v0 simulator in OpenAI Gym [13]. Ray allows for better utilization when running heterogeneous simulations at scale.

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