UIUC CS 425 Fall 22 - IDUNNO, a Distributed Learning Cluster

Ruipeng Han (ruipeng2), Tommy Kimura (tkimura4)

Design

For this Machine Problem (MP), we designed a distributed learning cluster that supports model training and inference across multiple machines. Our design is fault-tolerable, replicable, and fair with up to three simultaneous failures. In addition to our previous works on the SWIM style Failure Detector and the Simple Distributed File System (SDFS), we implemented a resource scheduler for Deep Learning (DL) inference that achieves both resource fairness and scalability

Our cluster is designed to have all machines to work on each job assigned in a **round-robin** manner. Every time a client initializes and runs a job, the request is sent to the coordinator/master/introducer (in our design, they are the same process and we will use the word coordinator here). Coordinator will then run the scheduler algorithm we designed that is similar to a round-robin.

Task assignment algorithm: Our scheduler algorithm communicates to each of the workers (all members in our membership list) concurrently and, for each worker, **it assigns the worker tasks from the job that has a lower query rate** to ensure the jobs has approximately equal query rate. We consider our implementation to be a variant of round-robin because it switches the next job assigned based on query rate, not time interval. Hence, if the first job comes in, all processes with work on it. If the second job comes in, all the processes will work on that second job whenever it has a lower query rate than the first job (and vice versa). If a job is done, all the processes will continue to consume the remaining job's tasks, until all jobs are finished.

Task assignment process: For each process/worker, concurrently, the coordinator would send a batch of filenames (of the test files of the job determined by our algorithm above) to the process, and then the machine would download and then run model inference on the files. The machines will store the inference results and upload it to the SDFS for the client to retrieve. This design is also replicable, since we send job information to other machines as well through piggybacking every inference batch message. When the coordinator fails, the newly elected coordinator (MP 3) will have the most up to date job information, and continue job scheduling. Since the inference job is idempotent, we would not lose any data with only a small cost of extra queries and time.

We have a loading function to load an ImageNet subset with about 5000 images. Our model has ResNet50 and Inception-V3 supporting this dataset. We chose these because of the high accuracy and the relatively small parameters for the VMs.

The programming languages we used are **GoLang and Python**. Our Failure Detector and SDFS are implemented with GoLang, and the medium of node communication is gRPC and UDP. However, due to lack of compatible DL packages (torch, tensorflow), we have decided to use Python to run the ML tasks for the jobs. Therefore, not only did we use gRPC for inter-node communication (between the node), we also used it for intra-node communication (within the node between GoLang scheduler and the Python ML server).

Data collection

1) Fair Time Inference

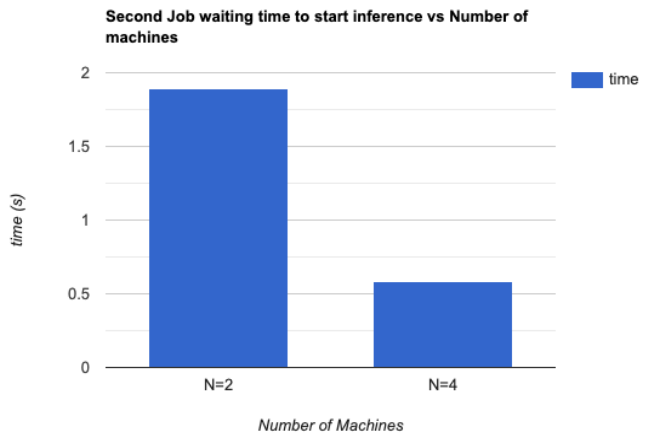
a) Ratio of resources

The ratio of resources of two different jobs will be 1:1. Within each pool, the scheduler runs the two jobs interchangeably. For this MP, we would like the two jobs to get the same amount of resources for fairness, therefore the scheduler would favor jobs with a temporary lower query rate to minimize the gap between the two jobs.

b) Time to begin the new job upon joining

Upon joining, the scheduler would immediately get notified and start scheduling two jobs interchangeably after the current job's batch is successfully done.

	T1 (s)	T2 (s)	T3 (s)	T4 (s)	T5 (s)	Avg (s)	STD
N=2	1.69	2.05	1.35	2.47	1.89	1.89	0.42
N=5	1.35	0.24	1.51	0.77	1.63	1.10	0.58



We can see that as the number of machines increases, the average time for job2 to begin decreases. This is reasonable. When job2 joins, its query rate is zero and less than job1's. This means that the next job assigned to each machine must be job2. As the number of machines increases, it is more likely for a machine to finish its previous jobs right after job2 joins and start inferencing on job2. This also explains why the standard deviation increases as well with more variation of the data.

2) Time to repair after one non-coordinator VM failure

	T1 (s)	T2 (s)	T3 (s)	T4 (s)	T5 (s)	Avg (s)	Std
N=5	0.37	0.30	1.41	0.09	0.91	0.61	0.54

We measured the time difference between first detecting the failure and when the next batch of queries is taken. Our design is really fast in repairing, because our SDFS has three replicas for each file, and we provide each worker/member a list of replicas to fetch the files for inferencing. This means that even if one worker is failing, other workers are still assigned tasks and inference independently. Similar to 1) part b, the repair time is fast if there is a worker ready right after the failure, and slower if all workers just started their tasks and will only start the next batch after a round of inference.

3) Time to repair after coordinator failure

	T1 (s)	T2 (s)	T3 (s)	T4 (s)	T5 (s)	Avg (s)	Std
N=5	1.03	1.04	1.03	1.14	1.04	1.05	0.04

The time to repair after coordinator failure is expected to be greater than a non-coordinator failure. This is because we have a new coordinator election after the failure detection. The system needs to wait for the new coordinator to get elected before continuing. This explains the lower standard deviation between each trial, since the time is now dependent on the election algorithm, which is relatively more consistent than the number of idle workers in the system.

Throughput & Latency

- We can see that the throughput increases while the latency remains relatively the same. This is because we have achieved parallelism for inference.