Fulcrum : Scheduling dynamic tasks with low latency and high throughput

University of Illinois at Urbana-Champaign

Abstract

Modern machine learning systems are constantly evolving and have different and challenging requirements over traditional ones. Such systems entail computation with millisecond latency, high demand for throughput, and dynamic and heterogeneous task graphs. This poses unique challenges for making scheduling decisions which need to conform to latency and throughput needs. Previous approaches largely assume long running tasks, static computation patterns or homogeneous tasks. In this project, we propose to investigate scheduling algorithms for dynamically changing, millisecond latency, high throughput, heterogeneous tasks. For proof of concept, we plan to incorporate our ideas in Ray: a recent system for reinforcement learning.

1 Introduction

The world is generating data at a breakneck pace. In 2017, there were 8.4 billion connected devices, generating 2.5 quintillion bytes of data every single day and showing no signs of slowing down [1, 2]. This "big data" trend triggered the need for low latency, high throughput large-scale data analytics systems like Dremel [3], Spark [4], Drill [5] etc. capable of generating actionable insights from data quickly.

Advances in storage, cloud computing and networking have unlocked the potential of Machine Learning (ML) applications. ML systems like TensorFlow [6], MXNET [7], Pytorch [8] etc. go beyond analytics to learn the latent characteristics of big data. Such systems enable training on a large corpus of data to learn a highly accurate model which can then be used to serve inference on new unseen data.

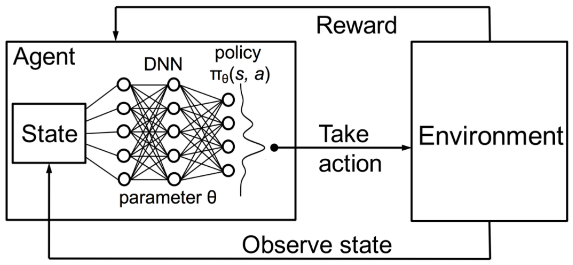

Figure 1: Reinforcement Learning setup with policy represented by deep neural network [9].

A recent shift in big data ML applications is towards making quick (near real time) decisions based on freshly observed data. Reinforcement Learning (RL) is one such area of ML. Figure 1 shows a typical setup for an RL application. An RL agent (for ex: self-driving car) observes the state of the environment (like road conditions), takes an action (turn left or right) and receives a reward (if the turn was accident-free or not). Over time it learns an optimal driving policy represented via a deep neural network.

System requirements for supporting such applications entail: a) small tasks (millisecond duration) and millions of tasks per second, b) nested parallel simulations, c) dynamic task graph with task dependencies determined at runtime, d) shared mutable state of tasks and e) support for heterogeneous resources (CPUs, GPUs, etc.). Frameworks like Ray [10], Dask [11] and Ciel [12] are nascent systems that aim to meet these requirements.

Designing a high throughput (millions of tasks/sec) and low latency scheduler for dynamically generated tasks is very challenging. To meet the throughput and latency requirements the scheduler needs to make good scheduling decisions in sub millisecond durations. We investigate the scheduling challenges in these systems by profiling different RL applications on Ray, including OpenAI RoboschoolHumanoid-v1 [13]. We use the observations from profiling Ray to guide the design of our Fulcrum scheduler.

The main contributions of the paper are:

- We present the task duration and scheduler runtime characteristics for several RL applications on Ray.
- We present initial high level design of Fulcrum scheduler and share the intuition behind our design.
- We integrate Fulcrum inside Ray and evaluate Fulcrum's performance.

2 Background

Latency and throughput requirements of big data systems continue to get stricter with recent systems [3, 4, 5] serving tasks over massive volumes of data. These trends have led to the design of schedulers that make millions of scheduling decisions per second. Mesos and Yarn [14, 15] are centralized schedulers that are designed for running batch jobs with long run times. Sparrow [16] is a stateless, distributed, sampling based scheduler designed for homogeneous, sub-second latency tasks with high throughput (million decisions per second). However, the latency requirements of systems we consider [10] are more stringent (sub-millisecond). Dominant Resource Fairness (DRF) [17] extends max-min fairness for heterogeneous tasks with different resource requirements. It is not applicable for low latency tasks with uniform resource needs.

Chaos [18] is a graph processing system that works on the GAS (Gather-Apply-Scatter) programming model. Graph processing system typically target static graphs. RL application task graphs are more dynamic and have arbitrary dependencies which get resolved only during runtime. This makes them incompatible with Chaos like systems. Litz [19] is a recent distributed ML framework system that uses micro-tasks to achieve job elasticity. Litz is targeted towards parameter-server based applications and handles elasticity by fine-tuning data/model parallelism. It is unclear if Litz is compatible with un-structured dynamic task graphs that most RL applications need to support.

Several previous works have dealt with scheduling of task graphs, especially in parallel computing literature. Work stealing is a promising approach to schedule task graph based computations. One main challenge with work stealing is to efficiently deal with task heterogeneity. Agarwal et. al [20] showed that a work stealing approach achieves asymptotically optimal time for static task graphs. Maglalang et. al [21] take advantage of data locality in the context of NUMA architectures and target highly multicore machines. More recently, Kurt et. al [22] presented an approach to handle failures of tasks using work stealing.

To the best of our knowledge, none of the previous works have addressed scheduling for task graph based computations having the following characteristics - a) low latency tasks (few milliseconds) b) high throughput (millions of decisions per second) c) heterogeneous tasks d) dynamic task graphs e) large-scale distributed data processing systems.

3 Scheduler Design Goals

A scheduler for applications discussed in section 2 needs to meet the following high-level design goals:

- For scalability, the scheduler needs to be distributed and stateless. Keeping a scheduler stateless simplifies design and allows for horizontal scale-out.

- The scheduler should be able to make a large number of decisions (of the order of millions) per second to support the throughput needs of the system.

- The scheduler should make extremely quick decisions on a task. The tasks themselves might have running times of the order of milliseconds and hence the scheduling overhead should be in the order of tens of microseconds for it to be acceptable.

- The scheduler needs to handle heterogeneous tasks and should ideally perform better than a scheduler originally designed to only handle homogeneous tasks.

4 Algorithm and Implementation

Owing to the small task running times, the scheduler needs to be extremely lightweight. The current scheduling in Ray consists of a simple two-leveled scheduler consisting of a local scheduler at every node and a global scheduler. The system tries to schedule a task at the same node where it originated (via the local scheduler). If none of the local workers are free, the local scheduler forwards the task to the global scheduler for it to be scheduled on other workers in the system. Currently, the global scheduler picks a worker at random amongst the workers which have enough resources to execute that task (these resources need not be free). Additionally, the local scheduler periodically spills some of its tasks from its task queue to the global scheduler. This helps in keeping the system load balanced.

Algorithm 1 Fulcrum v1.0

```
 1: procedure REFRESH(double decayrate)
 2:     for i in [0, nworkers) do
 3:         load[i] = load[i] * decayrate
 4: procedure UPDATELOAD(worker i)
 5:     load[i] = load[i] + 1
 6: procedure GETWEIGHTEDRANDOM
 7:     maxLoad ← 0
 8:     for i in [0, nworkers) do
 9:         maxLoad = max(maxLoad, load[i])
10:     totalwt ← 0
11:     for i in [0, nworkers) do
12:         totalwt += maxLoad - load[i]
13:     rand = getRandom(0.0, totalwt)
14:     currwt ← 0
15:     for i in [0, nworkers) do
16:         currwt = currwt + (maxLoad - load[i])
17:         if currwt ≥ rand then return i
```

We propose a simple modification to the two tiered scheduler of ray. Whenever a task gets spilled from the local scheduler to the global scheduler, it can be seen as a signal that the local worker is busy at that point. The global scheduler simply maintains a counter of the number of tasks received from each local scheduler (procedure UPDATELOAD in Algorithm 1). For scheduling, it chooses a random worker accounting the weight of each worker and this weight is a function of the load on that worker (procedure GETWEIGHTEDRANDOM in Algorithm 1). A more loaded worker will have a smaller weight, and less loaded worker will have a larger weight. Currently, we have experimented with Algorithm 1. We plan to explore different strategies of setting

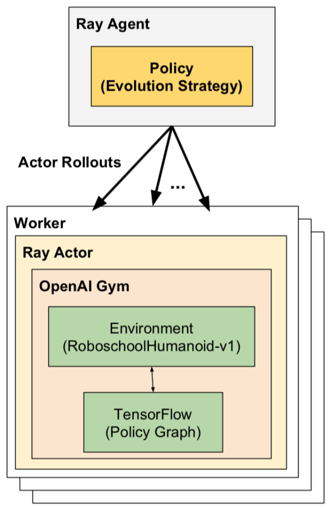
Ray Agent
Policy (Evolution Strategy)
Actor Rollouts
...
Worker
Ray Actor
OpenAI Gym
Environment (RoboschoolHumanoid-v1)
TensorFlow (Policy Graph)

Figure 2: Distributed deployment of 3D bipedal robot RL training using Ray in our experiments

the weight based on the load. This is a simple change in the implementation. More specifically, the weight in the update of *currwt* in line 16 of Algorithm 1 can be replaced with any decreasing function $g : g(load)$.

Additionally, to eliminate stale information the global scheduler invokes the REFRESH procedure of Algorithm 1 periodically to decay the load of every node. In our experiments, we picked a decay rate of 0.5, but it needs to be tuned.

5 Evaluation

5.1 Methodology

We implemented Fulcrum scheduler inside Ray library. We evaluate Fulcrum over a distributed RL application that trains a 3D bipedal robot to walk. Figure 2 depicts the implementation of the RoboschoolHumanoid-v1 application using Ray. Deployment consists of a Ray agent which uses Evolution Strategy policy to rollout multiple Ray actors (or simulations) across a set of workers. Each Ray actor connects to a 3D bipedal robot environment in OpenAI gym and executes the policy. Upon completion of simulation, the Ray actor shares the results (reward) with the Ray agent. This procedure repeats until convergence.

Setup: Our experiments are conducted on 5 Standard B2s (2 VCPUs, 4GB memory) virtual machine instances on Azure [23]. Experiments are designed to run for 150 iterations (rollout events) and result in 47715 tasks (actor simulations).

Baseline: We compare the performance of Fulcrum against Sparrow [16] and Ray's existing scheduler. Both these schedulers will be our baseline.

Evaluation Metrics: Our evaluation criteria is a) overall application execution time and b) scheduling decision latency.

5.2 Evaluation Goals

Through our evaluations, we want to address the following.

1. Justify the premise that reinforcement learning applications spawn tasks whose runtimes are of the order of milliseconds. This in turn strongly supports the case for a scheduler with sub millisecond latencies or else the scheduling time will be a major overhead.

2. Profile the task scheduling performance and overhead of Ray's scheduler.

3. Compare Fulcrum and Ray's scheduler based on evaluation metrics

5.3 Observations

1. Figure 3 shows the Cumulative Distribution Function (CDF) of the spawned tasks which are created during the run of the RoboschoolHumanoid application. As per the CDF, 99% of tasks finish under 600 milliseconds. This observation is strongly consistent with our premise that the runtime of most tasks spawned during a run of

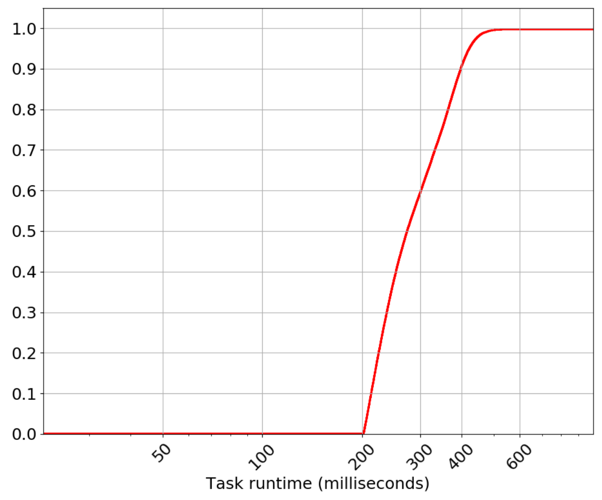

Figure 3: Cumulative Distribution Function of runtimes of tasks generated over 150 iterations of Roboschool Humanoid in Ray.

a reinforcement learning application is of the order of milliseconds. Thus, this application serves as a good example to benchmark our scheduler.

2. Figure 4 describes how Ray distributes the spawned tasks across 5 workers. We observe that 3 workers have a balanced distribution of tasks however the other 2 are imbalanced. Tasks in Ray can be either invoked on actors or as stateless remote functions. We believe this imbalance is because Ray naively assigns the tasks invoked on actors by hashing.

3. In Figure 5, we consider a special balls and bins case where the number of tasks is equal to the number of workers. It is well known that randomly distributing tasks would result in a maximum load $\theta(\log n / \log \log n)$. Using load based weights to randomly choose a worker would decrease the chances of collision, which in turn would improve the running time of the application. The application in Figure 5 is an RL agent trying to learn a pong game. In every iteration, n runs of the game are spawned with slightly perturbed policies. Once all tasks get finished, the policy is updated using the reward of each run and the next set of tasks are rolled out. Note that the task graph in this example is similar to a BSP model and hence does not necessarily capture the entire range of scenarios. As show in Figure 5, Fulcrum does marginally better than Ray's random scheduling. We found out that our algorithm suffers slightly in cases where the number of tasks is much higher than the number of workers. We suspect that this could be due to inefficient scheduling or imperfect weight functions. We plan to investigate both of these directions.

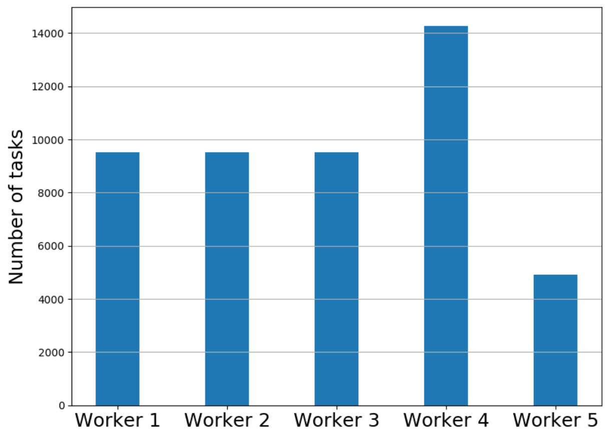

Figure 4: Task distribution across workers over 150 iterations of Roboschool Humanoid in Ray.

5.4 Future Strategies for Scheduler Algorithm Design

We plan to experiment with different weight functions based on the load. This could be also determined analytically and is work in progress. The problem could be formulated as follows:

Consider n workers, each with it's own local scheduler and one global scheduler. Assume that task sizes are drawn independently

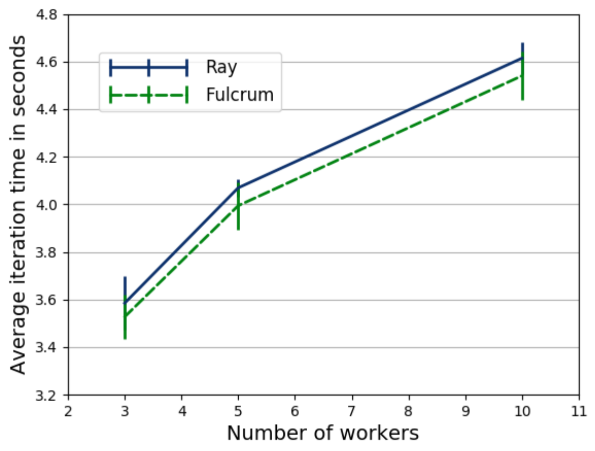

Figure 5: For the balls and bins case (tasks = workers), our first cut solution provides marginal benefits. Note that Fulcrum's advantage might improve with weight engineering and parameter tuning, which we believe are promising directions. Each data point is averaged over 5 runs.

from a given distribution (this is not always completely true in practice since there could be bursts, but might approximate a large enough system with several tasks). Finally, the task inter-arrival times on each worker are also drawn from a distribution with different means μ_i on every worker. We explicitly assume different means on every worker to model the load imbalance and the resource heterogeneity. For simplicity, we assume a constant network latency t_l from every local scheduler to the global scheduler. The goal of the global scheduler is to minimize the expected latency of each task.

Apart from this, we also wish to explore certain other approaches:

1. A scheduling algorithm can take advantage of the fact that sibling tasks (rolled out from the same parent task) can have similar running times. As an example, consider a reinforcement learning setting, where a robot is maneuvering across a maze. While training, the robot can make random local decisions from its current state and then update its model policy based on the rewards it gets. Since the states resulting from these random local decisions are minor perturbations of the current state, they might involve similar running times in many cases. We plan to explore sampling based approaches to estimate running times of a set of sibling tasks in the task graph. Such an approach takes advantage of some degree of homogeneity within sibling tasks while at the same time accounting for heterogeneity across tasks from different parents.

2. During the training phase, a parent task rolls out more tasks as soon as all its children tasks are finished. Thus, it is advantageous to finish all sibling tasks together. The scheduling algorithm can prioritize tasks whose completion will lead to further roll-out of tasks. This approach is related to co-flow scheduling of network flows [24].

3. We intend to profile individual iterations of the training process and observe any patterns in the execution times. Based on the observations, it might be possible to draw useful insights for improving scheduling of tasks associated with future iterations.

6 Conclusion

Our approach to come up with a high throughput and low latency scheduler for dynamic tasks looks promising. Even with our crude and untuned algorithm we are able to perform slightly better than the existing Ray scheduler. It is possible that we might be able to beat the existing schedulers with a higher margin after tuning the parameters or more sophisticated weight engineering that we wish to explore.

References

[1] Connected devices in 2017. https://www.gartner.com/newsroom/id/3598917. visited on 2018-04-01.

[2] Data never sleeps. https://www.domo.com/learn/data-never-sleeps-5. visited on 2018-04-01.

[3] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.

[4] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[5] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 225–238, New York, NY, USA, 2017. ACM.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv e-prints*, December 2015.

[8] PyTorch. http://pytorch.org. visited on 2018-04-01.

[9] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 197–210, New York, NY, USA, 2017. ACM.

[10] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. *ArXiv e-prints*, December 2017.

[11] Dask parallel computing library. http://dask.pydata.org. visited on 2018-04-01.

[12] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[13] OpenAI Roboschool. https://blog.openai.com/roboschool/. visited on 2018-04-01.

[14] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 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*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[15] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[16] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[17] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[18] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM.

[19] Litz: An Elastic Framework for High-Performance Distributed Machine Learning technical report. http://www.pdl.cmu.edu/PDL-FTP/BigLearning/CMU-PDL-17-103.pdf. visited on 2018-04-01.

[20] Kunal Agrawal, Charles E Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[21] J. Maglalang, S. Krishnamoorthy, and K. Agrawal. Locality-aware dynamic task graph scheduling. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 70–80, Aug 2017.

[22] Mehmet Can Kurt, Sriram Krishnamoorthy, Kunal Agrawal, and Gagan Agrawal. Fault-tolerant dynamic task graph scheduling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 719–730. IEEE Press, 2014.

[23] Azure. https://azure.microsoft.com/. visited on 2018-04-01.

[24] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. *SIGCOMM Comput. Commun. Rev.*, 44(4):443–454, August 2014.