Learning Scheduling Algorithms for Data Processing Clusters

Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, Mohammad Alizadeh

MIT Computer Science and Artificial Intelligence Laboratory
{hongzi, malte, bjjvnkt, zili, alizadeh}@csail.mit.edu

Abstract

Cluster schedulers today rely on generalized heuristics with extensive manual tuning. This can be costly: they may run computations inefficiently or unnecessarily leave resources idle. We introduce Decima, a system that uses reinforcement learning to automatically train scheduling policies for high-level objectives without encoding human-engineered heuristics. Off-the-shelf RL techniques, however, cannot handle the complexity and scale of the scheduling problem. To build Decima, we had to develop new representations for jobs' dependency graphs, design scalable RL models, and invent new RL training methods for continuous job arrivals. Our prototype integration with Spark shows that Decima outperforms existing heuristics by at least 21% and the default Spark scheduler by 3.1×.

1 Introduction

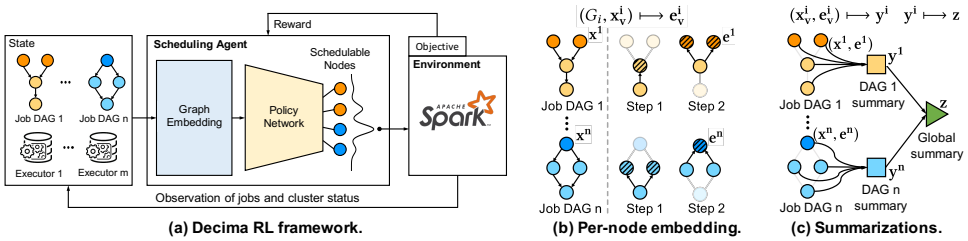
Efficient utilization of the expensive compute clusters matters for enterprises: even small improvements in utilization can save millions of dollars at scale [2, §1.2]. Cluster schedulers are key to realizing these savings. A good scheduling policy packs work tightly to reduce fragmentation [7, 8, 20], prioritizes jobs according to high-level metrics such as user-perceived latency [21], and avoids inefficiencies due to incorrect job configurations [6].

Current cluster schedulers, however, rely on heuristics that prioritize generality, ease of understanding, and straightforward implementation over achieving the ideal performance on a specific workload. In this paper, we show that modern deep reinforcement learning (RL) techniques can help side-step this trade-off by automatically learning highly efficient, workload-specific scheduling policies. We present Decima, a general-purpose scheduling service for data processing jobs with dependent stages. We focus on these jobs for two reasons: (*i*) many systems encode job stages and their dependencies as directed acyclic graphs (DAGs) [1, 3, 10, 22]; and (*ii*) scheduling DAGs is a hard algorithmic problem whose optimal solutions are intractable and difficult to capture in good heuristics [8].

Decima uses a neural network to encode its scheduling policy; it trains this policy network through a large number of simulated experiments, where it schedules a workload, observes the outcome, and gradually improves its policy (Figure 1a). To apply RL to complex cluster scheduling problems, however, we had to solve several key challenges.

First, standard neural networks require flat, numerical vectors as inputs, but the inputs to the scheduler are DAGs with attributes attached to nodes and edges. We developed a new embedding technique for mapping job DAGs with arbitrary size and shape to vectors that neural networks can process. Our approach builds upon recent work on learning graph embeddings [4, 5, 11], but is tailored to the scheduling domain. For example, existing embeddings cannot capture path-based properties such as a DAG's critical path, and we created new embedding methods for this purpose.

Second, cluster schedulers must scale to hundreds of jobs with thousands of machines. This makes for a significantly larger RL problem than typical game-play tasks, both in terms of the amount of information available to the scheduler (the state space), and the number of choices it must make (the action space). We therefore had to design a scalable RL formulation. Specifically, our neural network architecture processes any number of jobs with the same underlying parameters. We also



(a) Decima RL framework. (b) Per-node embedding. (c) Summarizations.

Figure 1: (a) In Decima's RL framework, a *scheduling agent* observes the *cluster state* to decide on a scheduling *action* to invoke on the *environment* (the cluster), and receives a *reward* based on a high-level objective. The agent uses a *graph embedding* to turn job DAGs into vectors for *policy neural networks*, which output actions. To process DAGs, Graph embedding transforms raw information each node of job DAGs into a vector representation. This example shows two steps of (b) local message passing and (c) two levels of summarizations.

factor actions into separate models for (*i*) picking a stage to schedule, and (*ii*) configuring the job's parallelism, which significantly reduces model complexity compared to a naïve action encoding.

Third, continuous, streaming job arrivals introduce undesirable variance that the conventional RL training approaches cannot tolerate [18, §3.7]. This variance exists because conventional RL training algorithms cannot tell whether two reward feedbacks differ due to different underlying job arrival patterns, or due to the quality of the learned scheduling policy's decisions. To counter this effect, we build upon recent work on new RL training techniques for variance reduction in settings with stochastic inputs [15]. By conditioning training feedback on the actual sequence of job arrivals experienced, we isolate the contributions of the scheduling policy in the overall feedback, making it feasible to learn policies that handle continuous job arrivals.

We integrated Decima with Spark [22] and evaluated it on both an experimental testbed and an industrial workload trace. Our experiments (§3) show that Decima outperforms existing heuristics on a 25-node Spark cluster, reducing average job completion time of TPC-H queries by 21% or more.

The rest of this paper will present an overview of Decima's design (§2) and highlights of experimental findings (§3). Further design details and experiments appear in the longer version of this paper [14].

2 Design

This section describes Decima's design, structured according to how it address the three aforementioned challenges: scalable graph embedding, encoding scheduling decisions as actions, and RL training with continuous stochastic job arrivals.

Spark scheduling problem. A Spark job consists of a DAG whose nodes are dependent *stages*. Each stage represents an operation that the system can run in parallel. A stage becomes runnable as soon as all parent stages have completed. How many *tasks* of a stage can run in parallel depends on the number of *executors* that the stage holds. A scheduler must therefore handle two kinds of actions: (*i*) deciding which stage to run next and (*ii*) deciding how many executors to give to each stage.

RL formulation. On scheduling events — e.g., a stage completion (which frees up executors), or a job arrival (which adds a DAG) — the agent takes as input the current state of the cluster and outputs a scheduling action. The reward is derived from the run time objective. For example, to minimize average job completion time (JCT), the corresponding reward is $-\tau \times J$ at each time step, where τ is the absolute time (in seconds) since last action and J is the number of jobs in the system [13]. To train, we use standard policy gradient methods (e.g., A3C [17]) with a high-fidelity Spark simulator.

Scalable graph embedding. On each state observation, Decima converts the DAGs (of arbitrary shapes and sizes) to vectors using graph embedding. Our method is based on graph convolutional neural networks [11], but it is customized for scheduling. Given the vectors $\mathbf{x}_v^i$ of raw features for the nodes in DAG G_i, Decima builds a per-node embedding $(G_i, \mathbf{x}_v^i) \longmapsto \mathbf{e}_v^i$. The result $\mathbf{e}_v^i$ captures information from all nodes reachable from v (i.e., v's child nodes, their children, etc.). To achieve this, Decima propagates information from children to parent nodes in a sequence of message passing steps (Figure 1b). In each message passing step, a node v whose children have aggregated messages from all of their children (shaded nodes in Figure 1b) computes its own embedding as: $\mathbf{e}_v = g\left[\sum_{w\in\xi(v)} f(\mathbf{e}_w)\right] + \mathbf{x}_v$, where $f(\cdot)$ and $g(\cdot)$ are non-linear transformations over vector inputs implemented as neural networks, and $\xi(v)$ denotes the set of v's children. The same non-linear

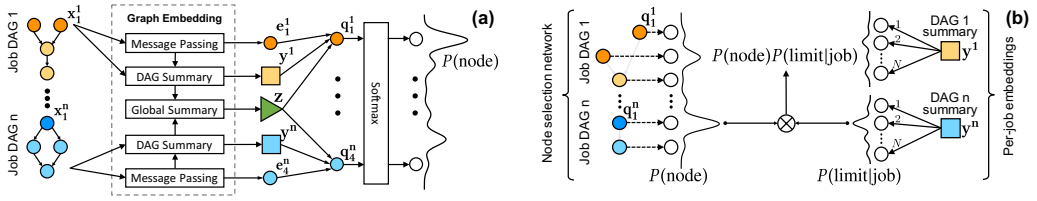


Figure 2: (a) For each node v in job i, the node selection network uses the message passing summary $\mathbf{e}_v^i$, DAG summary $\mathbf{y}^i$ and global summary $\mathbf{z}$ to compute a priority score $\mathbf{q}_v^i$ used to sample a node. (b) Decima's policy for jointly sampling a node and parallelism limit is implemented as the product of a node distribution, computed from graph embeddings, and a limit distribution, computed from the DAG summaries.

transformations $f(\cdot)$ and $g(\cdot)$ at all nodes, and in all message passing steps. Notably, combining two non-linear transforms $f(\cdot)$ and $g(\cdot)$ enables Decima to express a wide variety of aggregation functions. For example, if f and g are identity transformations, the aggregation sums the child node embeddings; if $f \sim \log(\cdot/n)$, $g \sim \exp(n \times \cdot)$ and $n \to \infty$, the aggregation takes the maximum of the child node embeddings. The graph embedding also includes a summary of all node features for each DAG, $\{(\mathbf{x}_v^i, \mathbf{e}_v^i), v \in G_i\} \longmapsto \mathbf{y}^i$; and a global summary across all DAGs, $\{\mathbf{y}^1, \mathbf{y}^2, \ldots\} \longmapsto \mathbf{z}$ (Figure 1c).

Encoding scheduling decisions as actions. Decima decomposes scheduling decisions into a series of two-dimensional actions, which output (*i*) a stage designated to be scheduled next, and (*ii*) a cap on the maximum allowed parallelism for that stage's job. At each scheduling event, Decima passes the processed vectors from graph embedding as input to the policy network (Figure 2a), which outputs of a composite action $\langle v, l_i \rangle$ of a stage v and a maximum level of parallelism l_i for v's job i. Here, we augmented Decima's node-selection action space to explicitly include a parallelism limit. The limit specifies a bound in $\{1, 2, \ldots, N\}$, where N is the total number of executors. However, naive augmentation would require $O(D \times N)$ possible actions, where D is total number of nodes, adding prohibitive complexity with tens of thousands of executors. We exploit a basic insight to achieve the same effective parallelism control with only $O(D + N)$ actions: parallelism levels of individual nodes can be unrestricted so long as the parallelism for the overall job is controlled (Figure 2b). Therefore, the probability of choosing a node and a limit at state s_t can be simplified as:

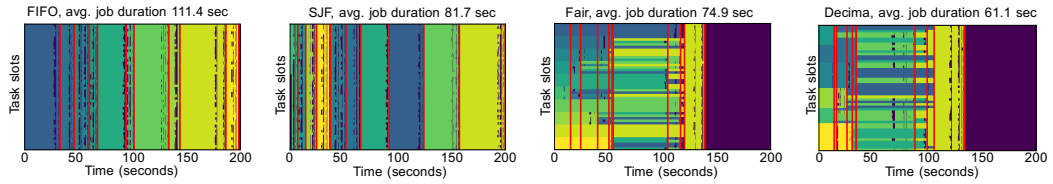
$$P(\text{node}, \text{limit}|s_t) = P(\text{node}|s_t) \times P(\text{limit}|\text{node}, s_t) = P(\text{node}|s_t) \times P(\text{limit}|\text{job}, s_t).$$

Handling continuous stochastic job arrivals. Training Decima for continuous job arrivals creates two challenges. First, the standard RL objective of maximizing the expected sum of rewards is not a good fit. For a set of N jobs $J_1, J_2, \ldots, J_N$, the standard objective minimizes $\mathbb{E}[\sum_{i=1}^{N} T(J_i)]$, where $T(\cdot)$ denotes the completion time of a job. However, with continuous job arrivals, our real objective is to minimize the average job completion time over a large time horizon, i.e., to minimize $\mathbb{E}[\lim_{N\to\infty} \sum_{i=1}^{N} T(J_i)/N]$. Thus, we use an alternative RL formulation that optimizes for the *average reward* in problems with an infinite time horizon [9, 18, §10.3, §13.6]. Operationally, this formulation replaces the standard reward with a *differential reward*: at every step t, the agent receives the reward $r_t - \hat{r}$, where r_t is the standard reward at time t and $\hat{r}$ is a moving average of the rewards r_t across a large number of previous time steps (across many training iterations). We refer the readers to Sutton and Barto [18, §10.3] for details on how this approach optimizes average reward.

Second, different job arrival patterns have a large impact on the reward feedbacks. Since this variance is due to randomness in the job arrival process, *not* the quality of scheduling decisions, it adds significant noise to the reward and distorts the policy gradient estimation. To account for the variance, we build upon recently-proposed variance reduction techniques for "input-driven" environments [15]. Specifically, we use "input-dependent" baselines that are customized for each instance of the job arrival sequence used in training. To implement these input-dependent baselines, the RL agent schedules the same job arrival sequence multiple times (i.e., multiple rollouts) during training. For the same job arrival sequence, we synchronously terminate all rollouts at the same step τ, where we draw τ randomly from a (memoryless) geometric distribution. We then compute the baseline by averaging over the rollouts of that particular job arrival sequence. This method accounts for the variance caused by different job arrival sequences, significantly improving the quality of policy gradients.

3 Experiments

We first analyze and understand how Decima compares with other heuristic based scheduler by visualizing a small set of job schedules. We illustrate this by running a mix of 10 randomly chosen



(a) FIFO scheduling. **(b)** SJF scheduling. **(c)** Fair scheduling. **(d)** Decima.

Figure 3: Decima improves average JCT of 10 randomly sample TPC-H queries by 45% over Spark's naïve FIFO scheduler, and by 19% over a fair scheduler on a cluster with 50 task slots (executors). Different queries in different colors; vertical red lines are job completions; purple indicates the idle period.

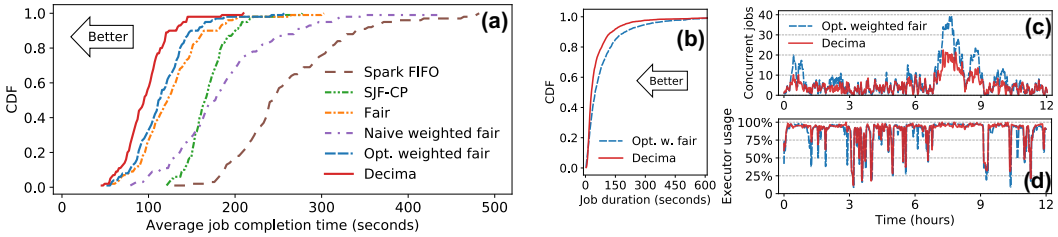


Figure 4: (a) Decima's learned scheduling policy achieves 21%–3.1× better average JCT than baseline algorithms for 100 batches of 20 concurrent TPC-H jobs in a real Spark cluster. (b) Streaming job arrival of 1,000 TPC-H jobs over 12 hours, Decima achieves 29% better average JCT than the best heuristic (other heuristics are not stable) and (c) has fewer active jobs at most points in time while (d) maintaining similar resource utilization.

TPC-H [19] queries on a Spark cluster with 50 parallel task slots. Figure 3 visualizes the schedules imposed by (3a) Spark's default FIFO scheduling; (3b) a shortest-job-first (SJF) policy that strictly prioritizes short jobs; (3c) a more realistic, fair scheduler that dynamically divides task slots between jobs; and (3d) a scheduling policy learned by Decima. We measure average job completion time (JCT). Decima achieves speedup over all comparing schemes (*i*) by completing short jobs quickly, as five jobs finish in the first 40 seconds; and (*ii*) by maximizing parallel-processing efficiency. Specifically, SJF dedicates all task slots to the smallest job to finish it early (but inefficiently). By better controlling parallelism, Decima reduces the JCT by 30% compared to SJF. Further, unlike fair scheduling, Decima partitions task slots non-uniformly across jobs, improving the JCT by 19%.

We then run a large set of TPC-H jobs with two arrival processes: (*i*) *batched*, in which a batch of multiple jobs arrives, and (*ii*) *continuous*, in which jobs arrivals follow a stochastic process. For batch arrivals, we randomly sample 20 jobs from six different input sizes (2, 5, 10, 20, 50, and 100 GB), producing a heavy-tailed distribution: 23% of the jobs contain 82% of the total work. The comparing heuristics are variants of SJF-based and fair-sharing based schedulers that are optimized for this experiment (details in [14]). Figure 4a shows a cumulative distribution of the average JCT achieved over 100 experiments. Decima outperforms all baseline algorithms and improves the average JCT by 21% over the closest heuristic ("opt. weighted fair"). This comes because Decima prioritizes jobs better, assigns efficient executor shares to different jobs, and leverages the job DAG structure.

For continuous job arrivals, We randomly sample 1,000 TPC-H jobs, and model their arrival as a Poisson process with an average inter-arrival time of 25 seconds. The resulting cluster load is about 85%. We record all job durations, and, in 10-second intervals, the concurrent number of jobs and the executor usage. We train Decima using both the average reward and synchronized termination techniques (§2) and evaluate with an unseen sequence of job arrivals. A busy period 8 hours into the experiment causes some scheduling policies to fall behind as they cannot finish jobs fast enough. Figure 4b shows the results for Decima and the only baseline algorithm that can keep up ("opt. weighted fair"). Decima achieves a 29% better JCT than the carefully-tuned weighted fair scheduler. Moreover, Decima uses the executors more efficiently: it has a higher executor usage (Figure 4d) and consistently maintains a lower active job count (Figure 4c) as it completes jobs sooner.

4 Conclusion

Decima demonstrates that automatically learning complex cluster scheduling policies using reinforcement learning is feasible, and that the learned policies are flexible and efficient. Decima's learning innovations, such as its graph embedding technique and the training framework for streaming, may be applicable to other systems that involves processing DAGs (e.g., query optimization [12], device placements [16]). We will open-source Decima, our models, and our experimental infrastructure.

Acknowledgement. We thank the anonymous NeurIPS reviewers for their valuable feedback. This work was funded in part by NSF grants CNS-1751009, CNS-1617702, a Google Faculty Research Award and an AWS Machine Learning Research Award.

References

[1] *Apache Tez*. https://tez.apache.org/.

[2] L. A. Barroso, J. Clidaras, and U. Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition". In: *Synthesis Lectures on Computer Architecture* 8.3 (July 2013), pp. 1–154.

[3] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. "FlumeJava: Easy, Efficient Data-parallel Pipelines". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Ontario, Canada, June 2010, pp. 363–375.

[4] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. "Learning Combinatorial Optimization Algorithms over Graphs". In: *NIPS*. 2017.

[5] M. Defferrard, X. Bresson, and P. Vandergheynst. "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: *CoRR* (2016).

[6] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. "Jockey: guaranteed job latency in data parallel clusters". In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012.

[7] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. "Multi-resource Packing for Cluster Schedulers". In: *SIGCOMM*. 2014.

[8] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. "Graphene: Packing and dependency-aware scheduling for data-parallel clusters". In: *OSDI*. USENIX Association. 2016, pp. 81–97.

[9] E. Greensmith, P. L. Bartlett, and J. Baxter. "Variance reduction techniques for gradient estimates in reinforcement learning". In: *Journal of Machine Learning Research* 5.Nov (2004), pp. 1471–1530.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72.

[11] T. N. Kipf and M. Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *CoRR* abs/1609.02907 (2016).

[12] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. "Learning to optimize join queries with deep reinforcement learning". In: *arXiv preprint arXiv:1808.03196* (2018).

[13] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. "Resource Management with Deep Reinforcement Learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets)*. Atlanta, GA, 2016.

[14] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. "Learning Scheduling Algorithms for Data Processing Clusters". In: *arXiv preprint arXiv:1810.01963* (2018).

[15] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh. "Variance Reduction for Reinforcement Learning in Input-Driven Environments". In: *arXiv preprint arXiv:1807.02264* (2018).

[16] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, et al. "Device Placement Optimization with Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. 2017.

[17] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, et al. "Asynchronous methods for deep reinforcement learning". In: *Proceedings of the International Conference on Machine Learning*. 2016, pp. 1928–1937.

[18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction, Second Edition*. MIT Press, 2017.

[19] *The TPC-H Benchmarks*. www.tpc.org/tpch/.

[20] A. Verma, M. Korupolu, and J. Wilkes. "Evaluating job packing in warehouse-scale computing". In: *Proceedings of the 2014 IEEE International Conference on Cluster Computing (CLUSTER)*. Madrid, Spain, Sept. 2014, pp. 48–56.

[21] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: cluster computing with working sets." In: *HotCloud* 10 (2010), pp. 10–10.